# Patrones Arquitectónicos y Estilos

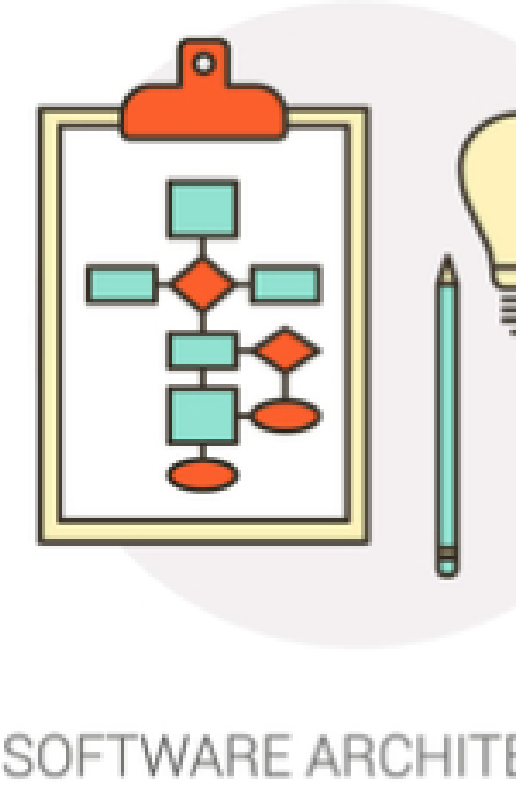
Comprendiendo la estructura y el estilo en el desarrollo de software

# Objetivos de la clase

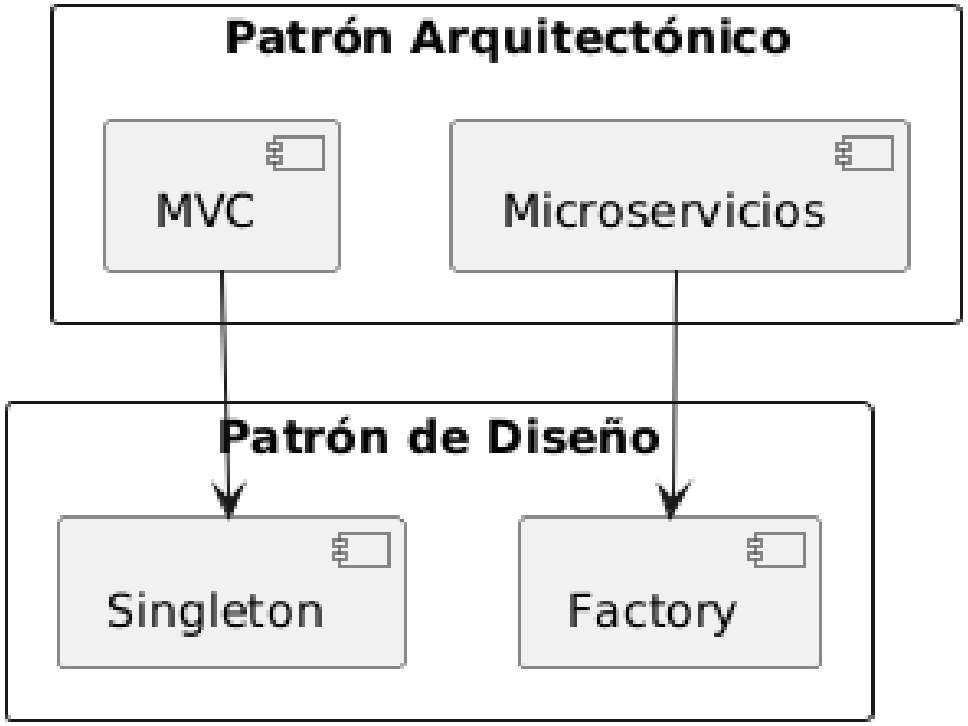
- Comprender qué son los patrones arquitectónicos.
- Identificar patrones comunes: MVC, MVVM, Layered.
- Reconocer estilos arquitectónicos: monolítico, microservicios, event-driven, SOA.
- Diferenciar cuándo aplicar cada uno.

# Introducción a los Patrones Arquitectónicos

- Los **Patrones Arquitectónicos** son **soluciones probadas y reutilizables** para resolver problemas comunes en el diseño de la **estructura global** de un sistema de software.



## Diferencia entre Patrón Arquitectónico y Patrón de Diseño

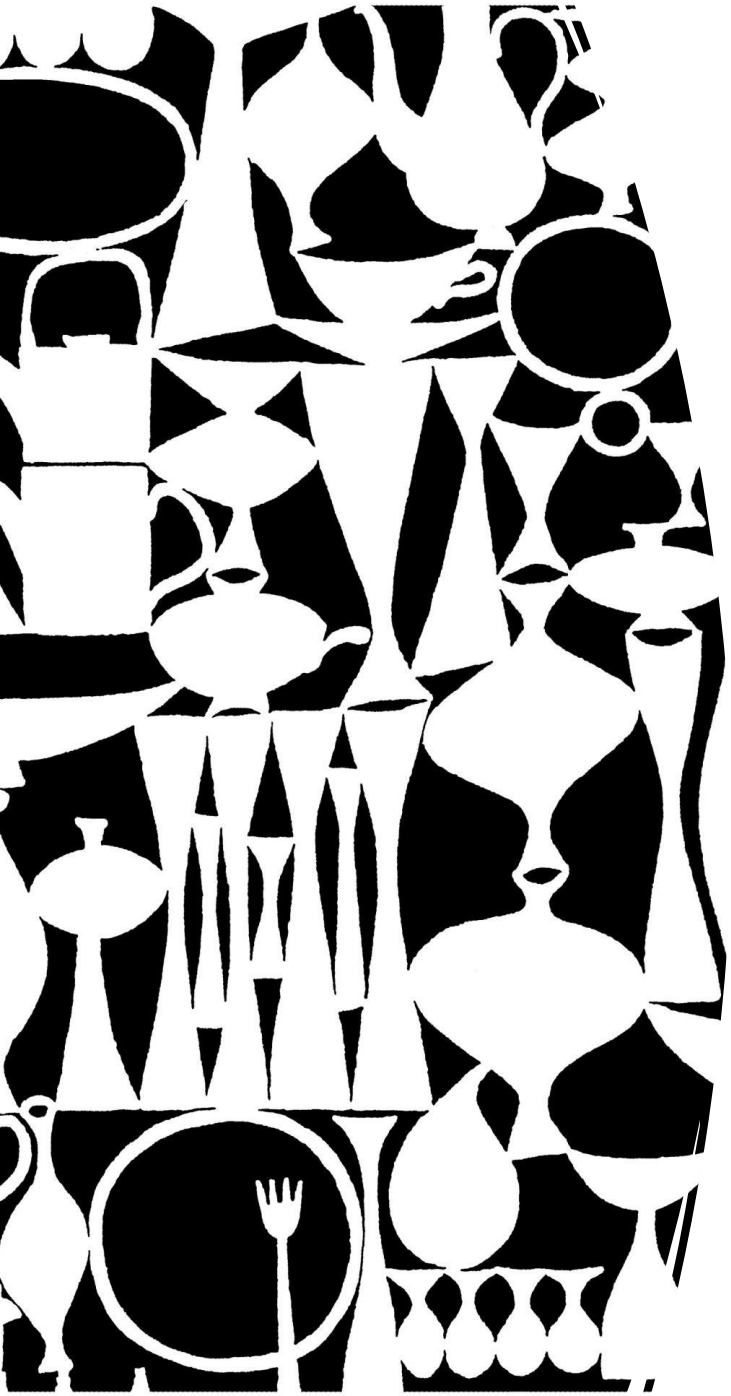


**Patrón Arquitectónico**

Define la **estructura global del sistema**. Establece cómo se organizan los módulos principales, cómo se comunican entre sí y qué responsabilidades tiene cada capa o componente.

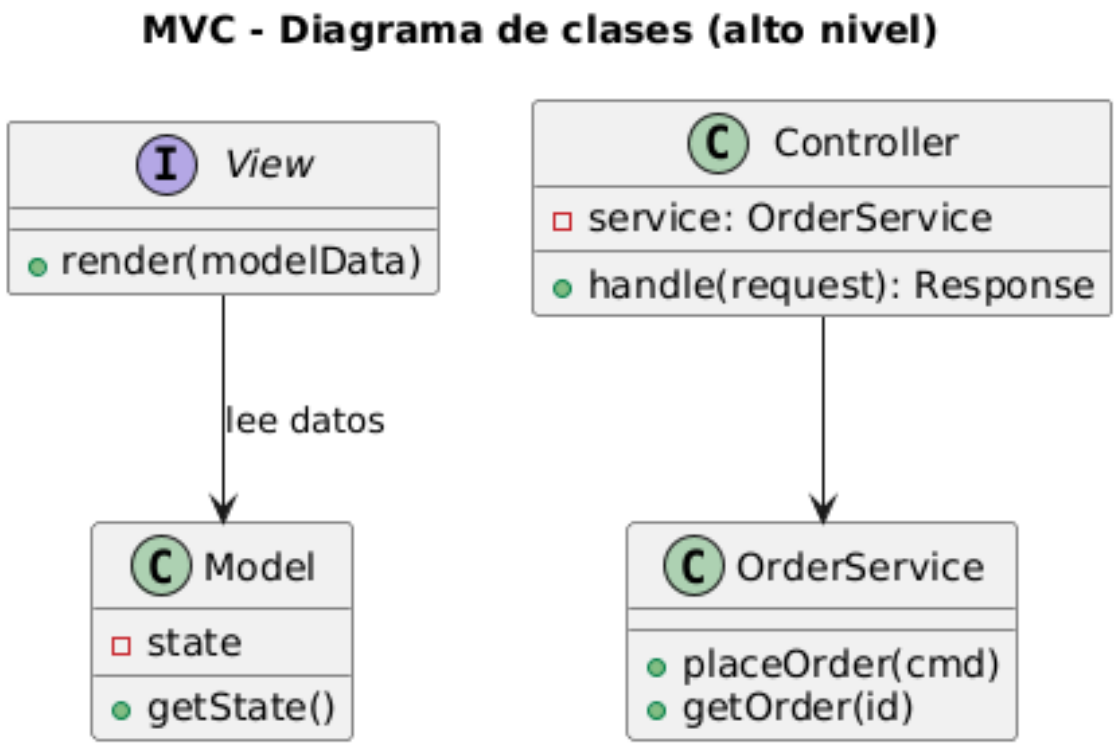
**Patrón de Diseño**

Se aplica a un nivel **más bajo y concreto**, resolviendo problemas recurrentes en la implementación de clases y objetos. Se centra en la reutilización, flexibilidad y buenas prácticas de codificación



# Patrones Arquitectonicos Comunes

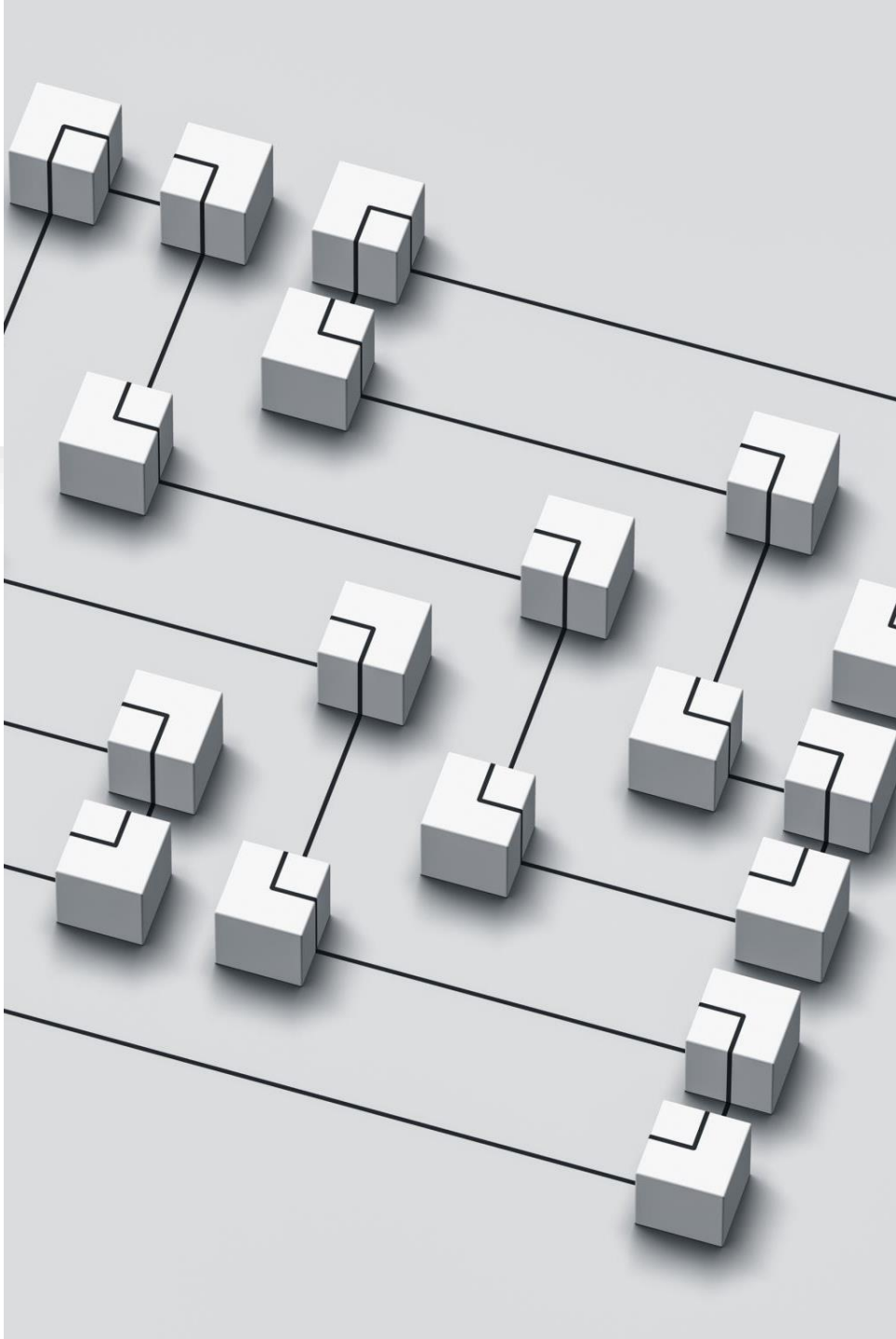
# MVC


MVC - Diagrama de clases (alto nivel)
I View
render(modelData)
C Controller
service: OrderService
handle(request): Response
lee datos
C Model
state
getState()
C OrderService
placeOrder(cmd)
getOrder(id)

# MVC

- **¿Qué es MVC?**
- MVC (Model–View–Controller) es un **patrón arquitectónico** que separa una aplicación en tres componentes principales:
- **Model**: gestiona los datos y la lógica de negocio.
- **View**: muestra la información al usuario.
- **Controller**: actúa como intermediario entre el usuario, el modelo y la vista.

- Esta separación permite organizar mejor el código y facilitar la escalabilidad y mantenimiento de las aplicaciones.



# MVC

**Metáfora**

- Imagina un **restaurante**:
- El **Cliente (usuario)** pide un plato.
- El **Mesero (Controller)** recibe la orden y la lleva a la cocina.
- El **Chef (Model)** prepara el plato con los ingredientes (datos).
- El **Mesero** trae el plato ya preparado.
- El **Comedor (View)** es donde el cliente ve y disfruta el plato servido.
- Así, cada uno tiene un rol definido y no se mezcla con el trabajo de los demás.

**Objetivo**

- Separar las responsabilidades de la aplicación para:
- Reducir la complejidad.
- Aumentar la reutilización de componentes.
- Permitir que equipos trabajen en paralelo en **interfaz**, **lógica** y **datos**.

# MVC

# MVC

**Ventajas**

- Favorece la **separación de responsabilidades**.
- **Escalable**: es fácil extender cada capa sin afectar las demás.
- **Reutilizable**: vistas diferentes pueden compartir el mismo modelo.
- Mejora la **mantenibilidad** del código.

**Desventajas**

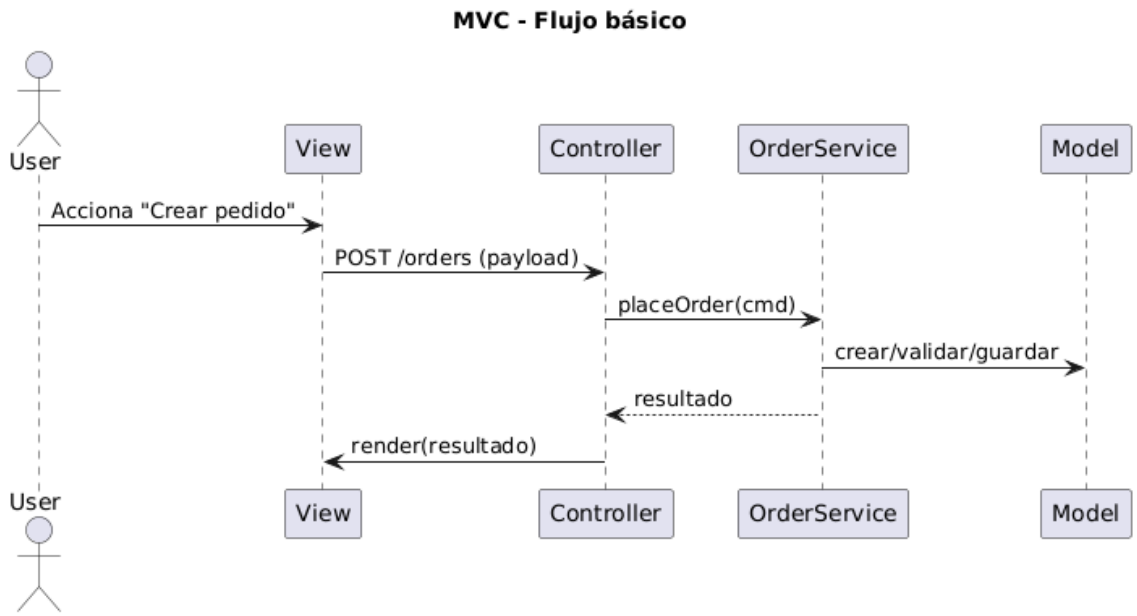
- Puede incrementar la **complejidad inicial** del proyecto.
- La comunicación entre capas puede generar **sobrecarga**.
- Requiere **disciplina en el desarrollo**, pues mezclar responsabilidades rompe el patrón.

# MVC

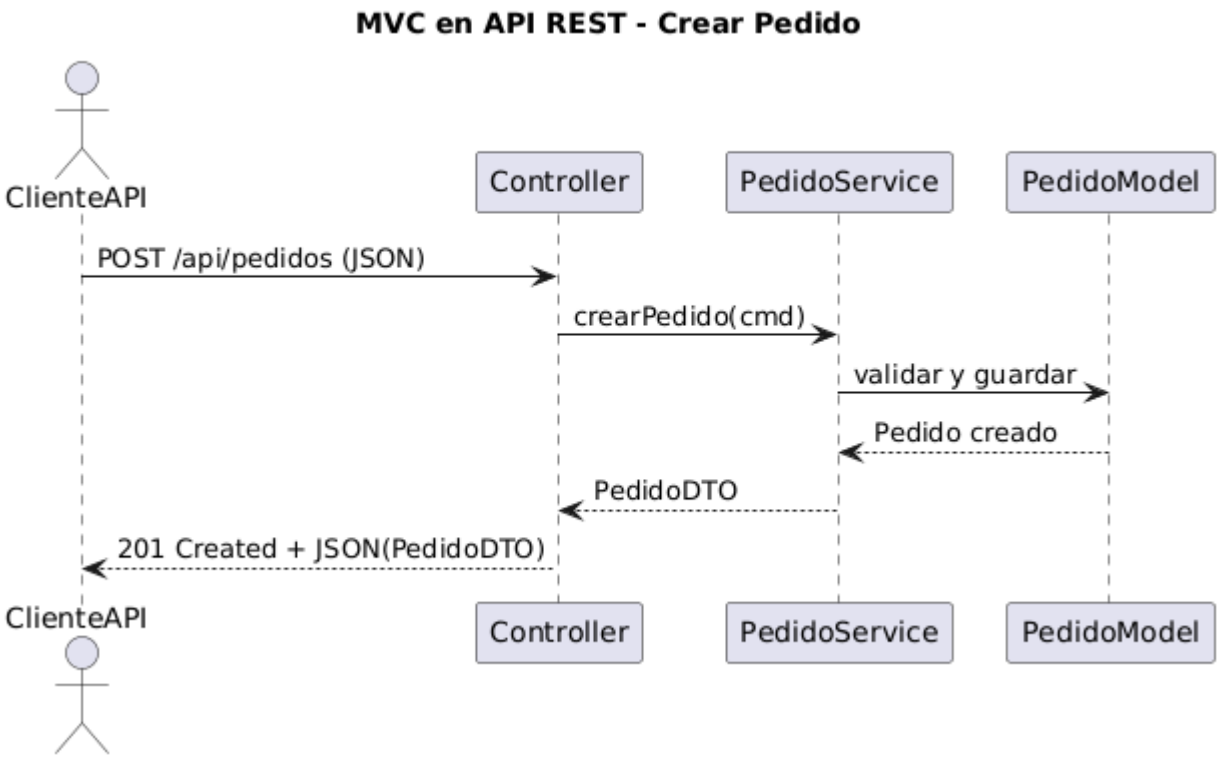
## Casos recomendados donde aplicar

- Aplicaciones **web** con interfaces dinámicas (ej: Spring MVC, ASP.NET MVC, Angular).
- Sistemas donde la **interfaz** debe cambiar constantemente pero la lógica de negocio se mantiene.
- Proyectos con **equipos grandes**, donde diseñadores, programadores backend y frontend trabajan en paralelo.

# MVC

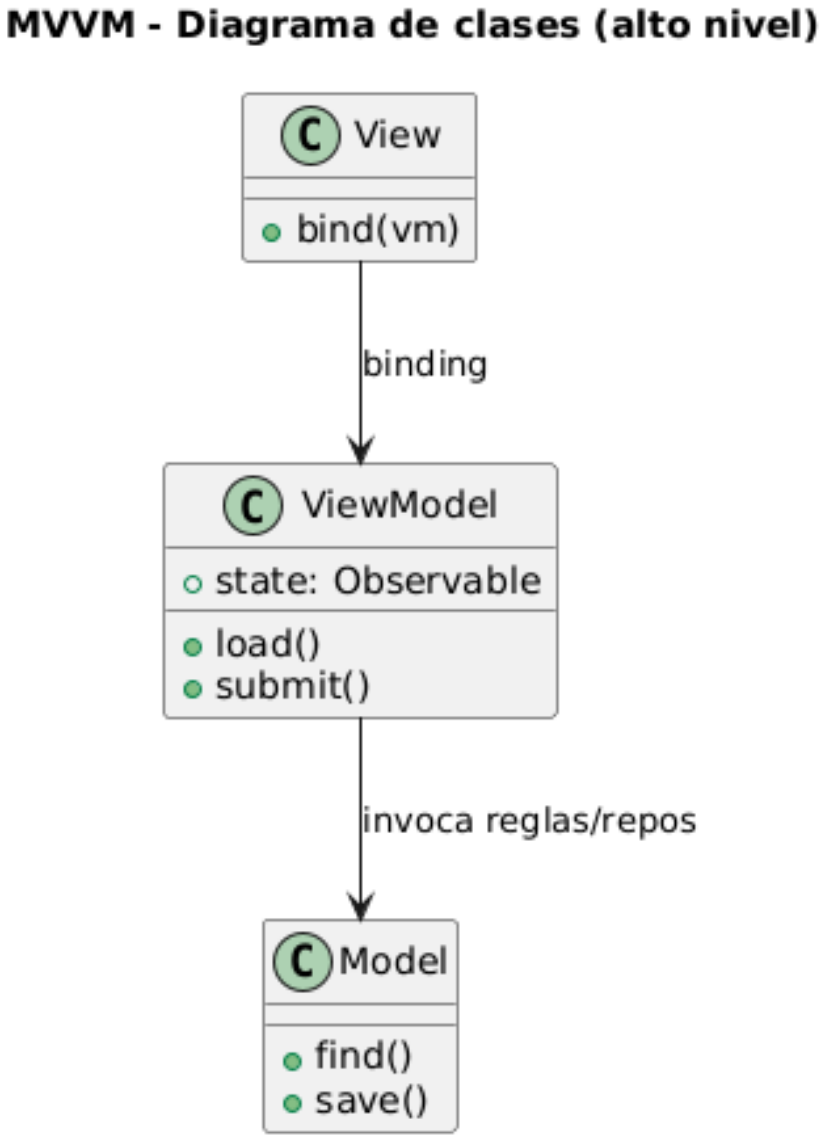

MVC - Flujo básico
User
View
Controller
OrderService
Model
Acciona "Crear pedido"
POST /orders (payload)
placeOrder(cmd)
crear/validar/guardar
resultado
render(resultado)
User
View
Controller
OrderService
Model

# MVC


MVC en API REST - Crear Pedido
ClienteAPI
Controller
PedidoService
PedidoModel
POST /api/pedidos (JSON)
crearPedido(cmd)
validar y guardar
Pedido creado
PedidoDTO
201 Created + JSON(PedidoDTO)
ClienteAPI
Controller
PedidoService
PedidoModel

Model View ViewModel

MVVM



# MVVM

Model View ViewModel

**¿Qué es?**

- MVVM es un patrón de arquitectura de software que separa la interfaz de usuario (View) de la lógica de negocio (Model), utilizando una capa intermedia llamada **ViewModel** que actúa como puente mediante un mecanismo de **data binding** (enlace automático de datos).
- Esto permite que los cambios en el modelo se reflejen automáticamente en la vista, y viceversa, sin necesidad de que estén acoplados directamente.

Model View ViewModel

# MVVM

**Metáfora**

- Imagina una **obra de teatro**:
- El **Modelo** es el **guion** con la historia completa.
- La **Vista** son los **actores en escena**, mostrando lo que ocurre al público.
- El **ViewModel** es el **director de escena**, que no aparece en el escenario, pero asegura que los actores representen correctamente el guion y coordina lo que se debe mostrar según las reglas.

# MVVM

**Objetivo**

- Facilitar la separación de responsabilidades en aplicaciones donde la **UI cambia constantemente**.
- Permitir que la lógica de presentación (ViewModel) esté desacoplada de la vista, facilitando pruebas unitarias y mantenimiento.
- Mejorar la **reactividad** entre vista y modelo a través del binding.

# MVVM

**Ventajas**

- **Separación clara de responsabilidades**: La lógica de negocio no está mezclada con la vista.
- **Reactividad automática**: Con data binding, la vista se actualiza cuando cambia el estado en el ViewModel.
- **Testabilidad**: El ViewModel puede probarse fácilmente sin necesidad de una interfaz gráfica.
- **Escalabilidad**: Ideal para proyectos donde la UI cambia mucho o se deben manejar múltiples estados.
- **Reutilización**: El mismo ViewModel puede servir para diferentes vistas.

**Desventajas**

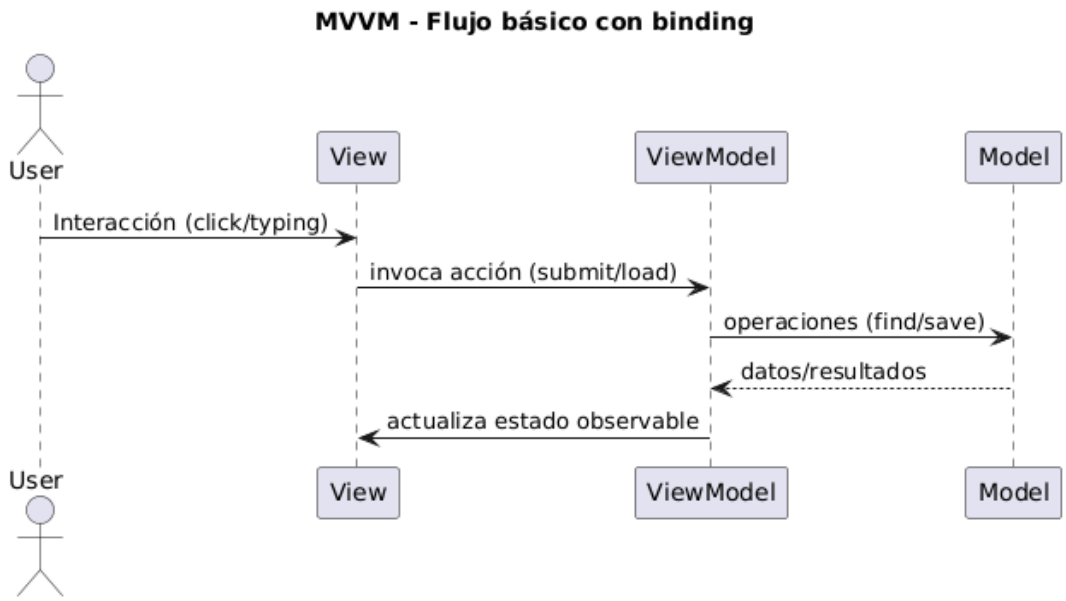
- **Complejidad inicial**: Requiere mayor curva de aprendizaje, sobre todo en proyectos pequeños.
- **Sobrecoste en apps simples**: Puede ser demasiado para un CRUD básico.
- **Dependencia de frameworks de binding**: En muchos lenguajes, se depende de librerías o frameworks para implementar correctamente el binding (ej. Angular, JavaFX, React con Hooks, etc.).
- **Debugging más difícil**: El binding automático puede ocultar errores si no se controlan adecuadamente.

# MVVM

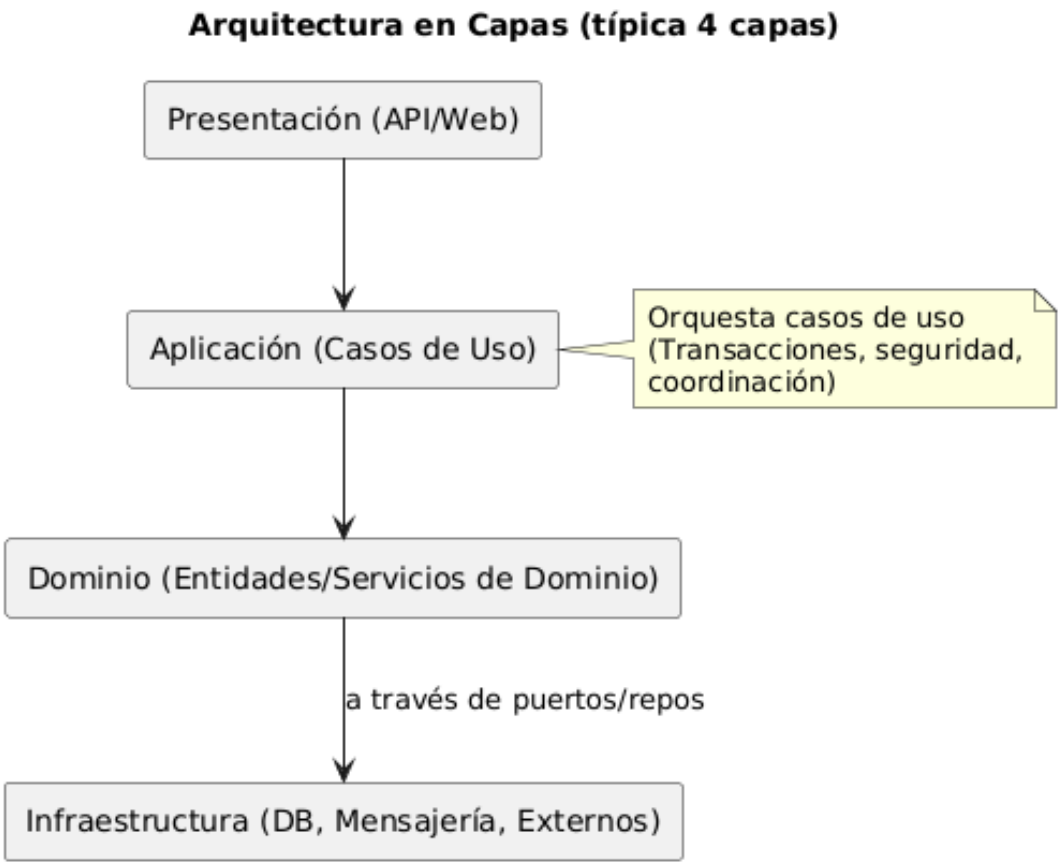
**Casos recomendados de uso**

- Aplicaciones con **interfaces gráficas complejas** (ejemplo: aplicaciones de escritorio con JavaFX o .NET WPF).
- Aplicaciones móviles donde los datos cambian dinámicamente (ejemplo: Android con Jetpack ViewModel).
- Frontends web modernos con **frameworks reactivoss** (ejemplo: Angular, React con hooks, Vue.js).
- Situaciones donde se requiere **sincronización automática** entre modelo y vista (formularios dinámicos, dashboards).

# MVVM


MVVM - Flujo básico con binding
User
View
ViewModel
Model
Interacción (click/typing)
invoca acción (submit/load)
operaciones (find/save)
datos/resultados
actualiza estado observable
User
View
ViewModel
Model

# Layered


Arquitectura en Capas (típica 4 capas)
Presentación (API/Web)
Aplicación (Casos de Uso)
Orquesta casos de uso (Transacciones, seguridad, coordinación)
Dominio (Entidades/Servicios de Dominio)
a través de puertos/repos
Infraestructura (DB, Mensajería, Externos)

# Layered

**¿Qué es?**

- La arquitectura en capas es un estilo arquitectónico clásico donde el sistema se organiza en niveles bien definidos.

Cada capa tiene una **responsabilidad específica** y se comunica solo con la capa inmediatamente inferior.
En el caso típico de 4 capas:

- **Presentación (API/Web)** → Interfaz de entrada del usuario o cliente.
- **Aplicación (Casos de uso)** → Orquesta reglas de negocio y coordina transacciones.
- **Dominio (Entidades/Servicios de Dominio)** → Modela la lógica y reglas de negocio.
- **Infraestructura (DB, mensajería, externos)** → Conecta con persistencia y servicios externos.

# Layered

**Metáfora**

- Imagina un **restaurante**:
- **Presentación (Meseros)**: Reciben la orden del cliente y la muestran claramente.
- **Aplicación (Chef principal)**: Organiza el pedido, define qué se necesita y asegura la secuencia correcta.
- **Dominio (Recetas de cocina)**: Contienen el conocimiento culinario (reglas de negocio).
- **Infraestructura (Ingredientes, cocina, proveedores)**: Recursos físicos y externos que permiten cocinar.
- El cliente no entra a la cocina ni trata con proveedores: todo fluye por capas ordenadas.

# Layered

**Objetivo**

Separar responsabilidades de manera clara y jerárquica.

Garantizar que los cambios en una capa tengan un **impacto mínimo** en las demás.

Mejorar la **mantenibilidad**, **modularidad** y la **claridad** del sistema.

# Layered

**Ventajas**

- **Simplicidad conceptual**: Fácil de entender y enseñar.
- **Separación de responsabilidades**: Cada capa tiene un rol definido.
- **Mantenibilidad**: Facilita modificaciones en una capa sin afectar las demás.
- **Reutilización**: El dominio puede ser usado con diferentes interfaces (ejemplo: web, móvil).
- **Testabilidad**: Posibilita probar capas de forma aislada.

**Desventajas**

- **Acoplamiento vertical:** Cambios en infraestructura pueden "filtrarse" hacia arriba.
- **Rendimiento:** Excesivo paso de datos entre capas puede ralentizar el sistema.
- **Rigidez:** Si no se diseña bien, puede volverse muy burocrática (mucho "boilerplate code").
- **Difícil evolución:** Para casos de integración moderna (eventos, microservicios), puede quedarse corta.

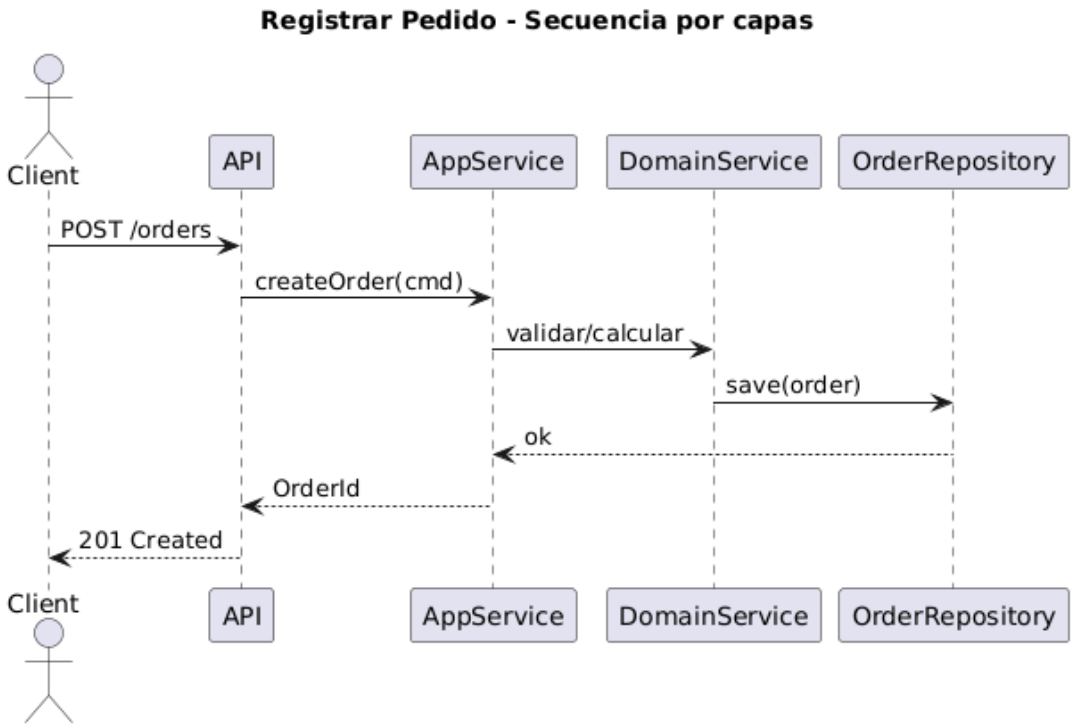
# Layered

**Casos recomendados donde aplicar**

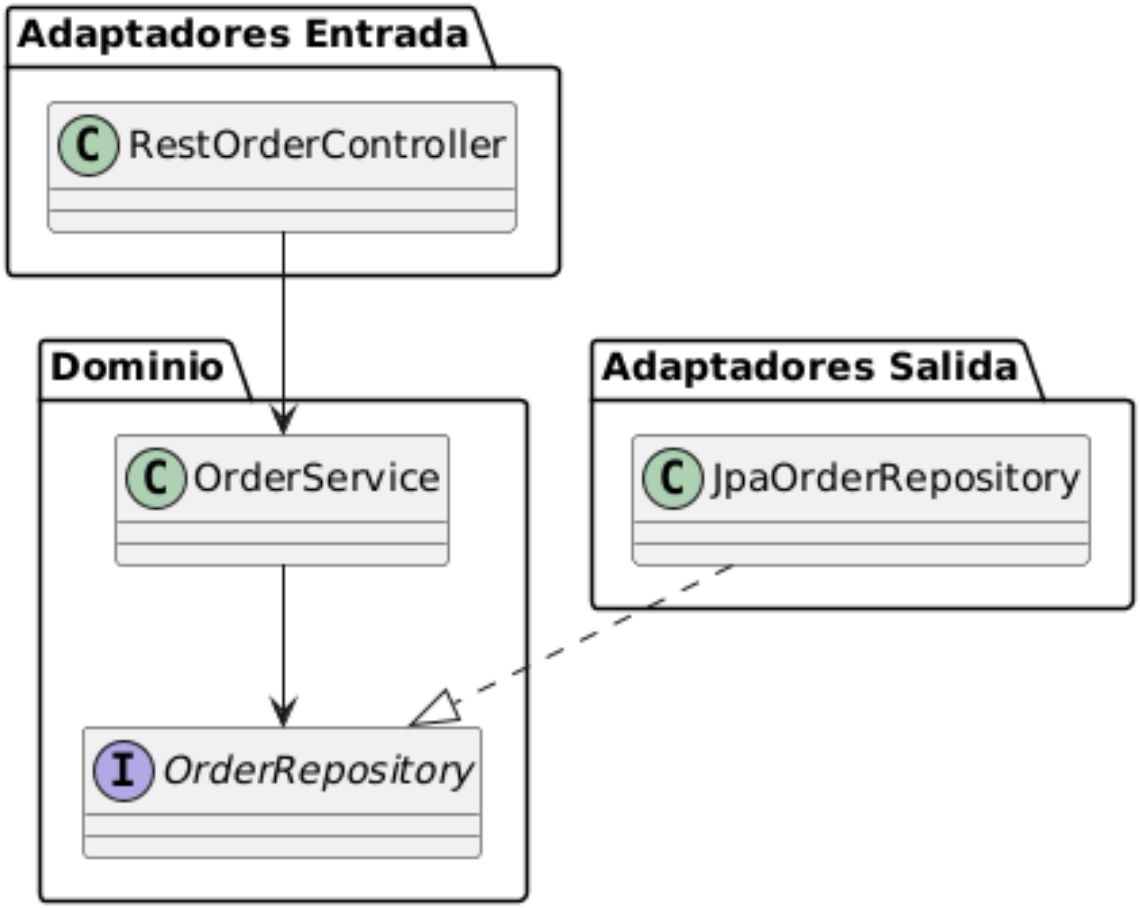
- **Aplicaciones empresariales tradicionales**: CRUD, ERPs, sistemas administrativos.
- **Sistemas donde la lógica de negocio es clara y estable**.
- **Proyectos con equipos grandes**: Cada equipo puede enfocarse en una capa.
- **Aplicaciones monolíticas bien organizadas**.

**No es tan recomendable** en proyectos altamente reactivos, de mensajería intensiva o microservicios modernos, donde patrones como **Hexagonal** o **Clean** dan más flexibilidad.

# Layered


Registrar Pedido - Secuencia por capas
Client
API
AppService
DomainService
OrderRepository
POST /orders
createOrder(cmd)
validar/calcular
save(order)
ok
OrderId
201 Created
Client
API
AppService
DomainService
OrderRepository

# Hexagonal


Adaptadores Entrada
RestOrderController
Dominio
OrderService
OrderRepository
Adaptadores Salida
JpaOrderRepository

# Hexagonal

**¿Qué es?**

Patrón arquitectónico que organiza una aplicación con un **núcleo de dominio independiente** rodeado de **puertos** (interfaces que define el dominio) y **adaptadores** (implementaciones tecnológicas de esos puertos: REST, DB, colas, CLI). Distingue **adaptadores de entrada** (driving) y **de salida** (driven).

# Hexagonal

**Metáfora**

Una **isla** (negocio) con **puertos** de embarque/desembarque. Los **barcos** (adaptadores) pueden cambiar —ferry, carguero, velero— sin modificar la isla ni sus reglas de atraque

# Hexagonal

**Objetivo**

- Proteger la **lógica de negocio** de frameworks y detalles técnicos.
- Permitir múltiples **formas de entrada** (REST, eventos, CLI) y **salida** (BD, APIs externas) intercambiables.
- Facilitar pruebas del dominio **en aislamiento**.

**¿Qué problema soluciona?**

- **Acoplamiento** del negocio a tecnologías (frameworks/ORM/HTTP).
- Dificultad para **cambiar infraestructura** (DB, proveedor externo).
- Baja **testabilidad** por dependencias duras.
- Necesidad de **varias interfaces** para el mismo caso de uso.

# Hexagonal

**Ventajas**

- **Independencia tecnológica** real (sustituir adaptadores sin tocar el dominio).
- **Testabilidad**: fakes/stubs de puertos para pruebas rápidas.
- **Extensibilidad**: añadir nuevos canales de entrada/salida sin reescribir reglas.
- **Claridad** de límites: contratos explícitos (puertos).

**Desventajas**

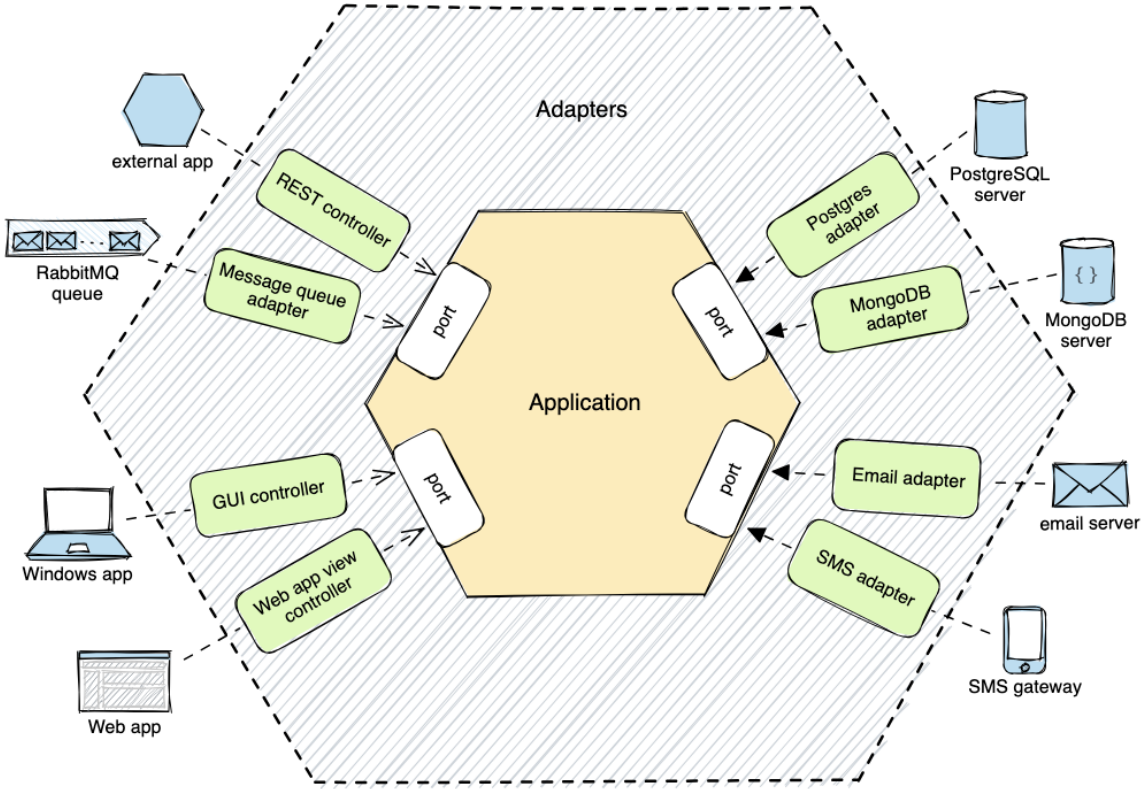
- **Más código** “ceremonial” (interfaces, mapeos, DTOs).
- Puede sentirse **complejo** en proyectos pequeños.
- Riesgo de **adaptadores “gordos”** si se cuela lógica de negocio.

# Hexagonal

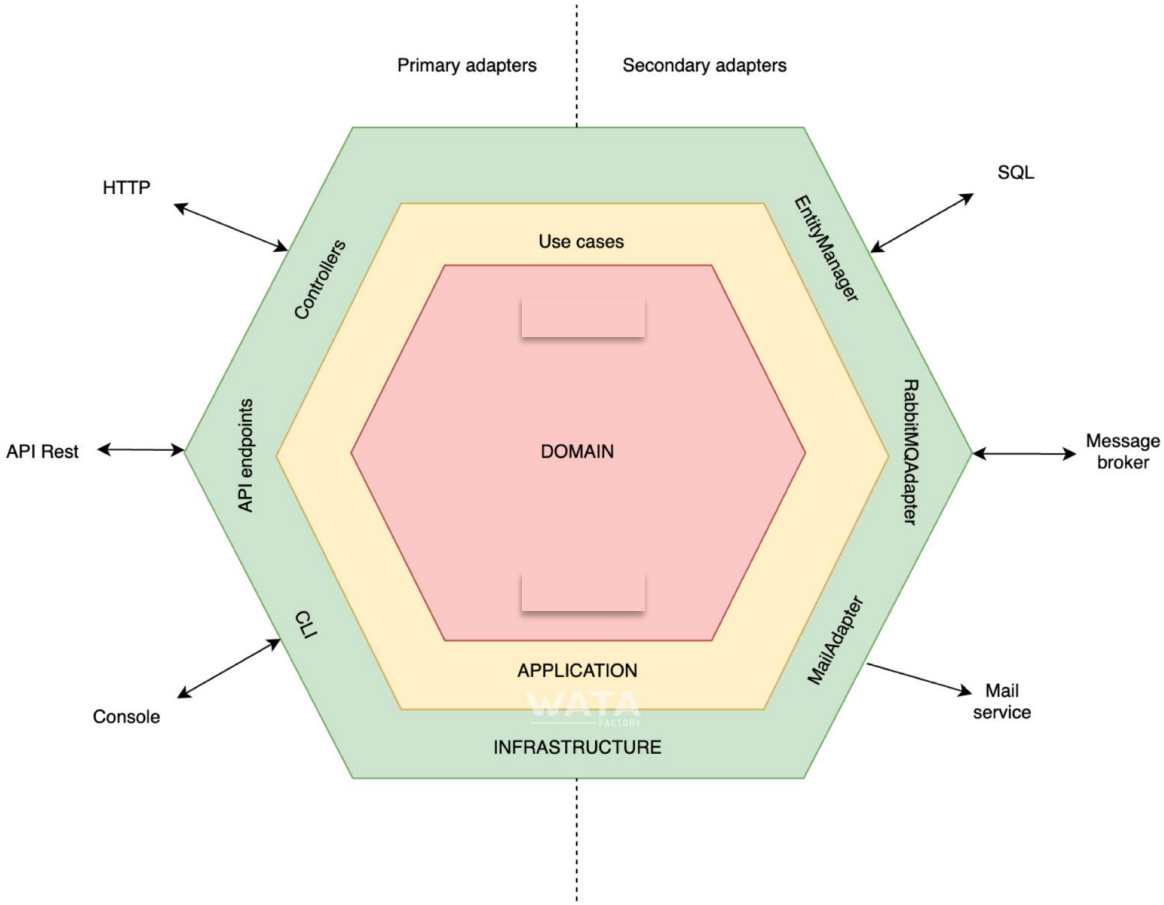
**Casos recomendados**

- Servicios con **múltiples interfaces** (REST + eventos + batch).
- Sistemas que integran **proveedores externos** cambiantes (pagos, identidad, RENIEC).
- Proyectos con **larga vida** y probables migraciones de tecnología (cambiar BD/cola/framework).
- **Microservicios**: cada servicio con dominio limpio y adaptadores propios.
- Necesidad de **pruebas unitarias** del negocio sin levantar infra.

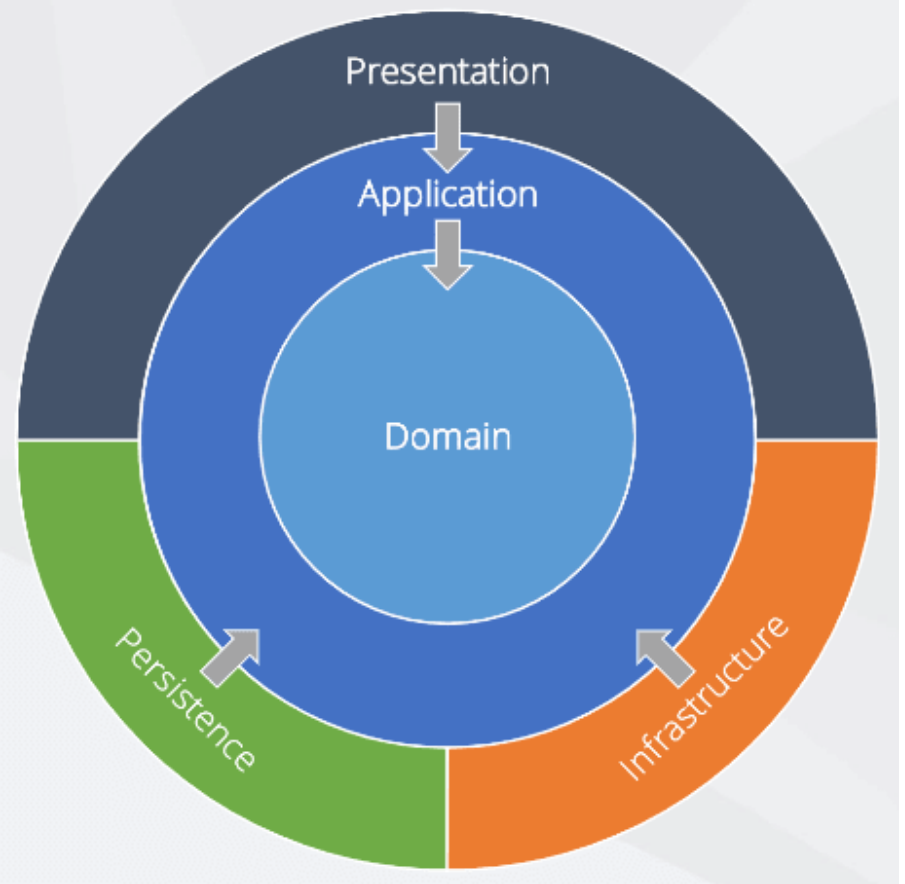
# Hexagonal

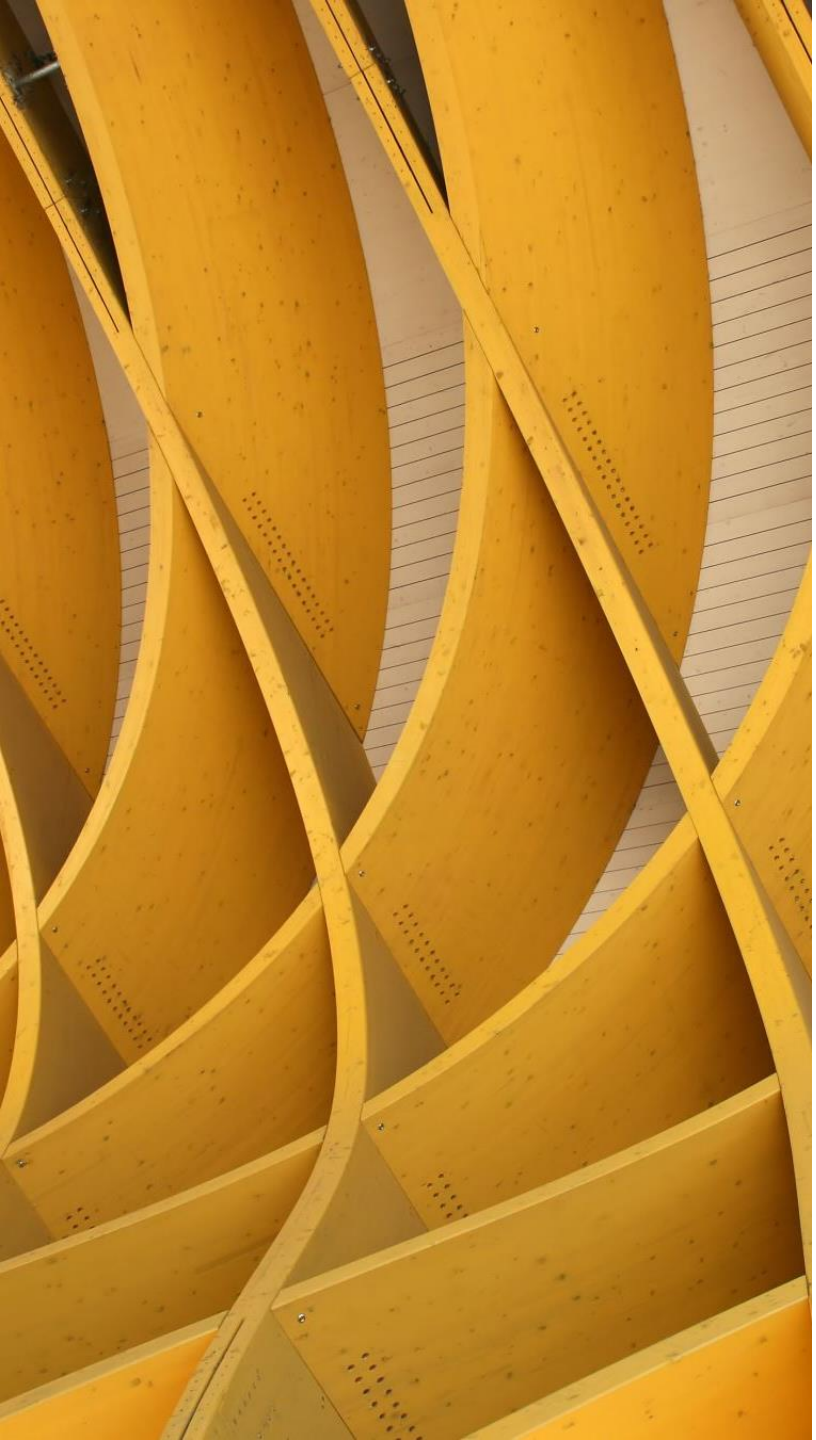

Adapters
external app
REST controller
RabbitMQ queue
Message queue adapter
port
Application
port
Postgres adapter
PostgreSQL server
MongoDB adapter
MongoDB server
port
GUI controller
Windows app
Web app view controller
Web app
port
Email adapter
email server
SMS adapter
SMS gateway

# Hexagonal


Primary adapters
Secondary adapters
HTTP
SQL
Controllers
EntityManager
Use cases
API Rest
API endpoints
DOMAIN
RabbitMQAdapter
Message broker
CLI
MailAdapter
APPLICATION
Console
Mail service
INFRASTRUCTURE

# ONION


Presentation
Application
Domain
Persistence
Infrastructure



# ONION

**¿Qué es?**

- La **Onion Architecture** (Arquitectura de Cebolla) es un estilo de arquitectura de software propuesto por Jeffrey Palermo. Su propósito es organizar el sistema en capas concéntricas donde la **lógica de negocio (Domain)** está en el centro y las dependencias fluyen hacia adentro. Esto significa que las capas externas (infraestructura, persistencia, presentación) dependen de las capas internas, pero nunca al revés.

# ONION

- **Metáfora**

Imagina una **cebolla**:

- En el **centro** está el corazón: la **lógica de negocio** que nunca debería verse afectada por detalles externos.
- Cada capa alrededor protege el núcleo. Puedes quitar o reemplazar una capa externa (por ejemplo, cambiar de base de datos o framework web) sin dañar el núcleo interno.
- Así como al pelar una cebolla llegas al centro protegido, en este modelo todo está diseñado para proteger el dominio de cambios externos.

# ONION

**Objetivo**

- **Proteger la lógica de negocio** de los detalles tecnológicos.
- **Asegurar independencia** de frameworks, bases de datos y librerías externas.
- **Facilitar pruebas unitarias** aislando el dominio de dependencias.
- **Permitir evolución** del sistema reemplazando infraestructura sin alterar la lógica central.

# ONION

**Problema que soluciona**

- La **alta dependencia** de frameworks, bases de datos o UI que suele ocurrir en arquitecturas tradicionales.
- La **dificultad para mantener y escalar** sistemas donde los cambios en infraestructura rompen la lógica de negocio.
- La **mala separación de responsabilidades**, donde reglas del negocio quedan mezcladas con detalles técnicos.

# ONION

**Ventajas**

- **Alta mantenibilidad**: el núcleo del negocio no depende de cambios externos.
- **Testabilidad**: se puede probar la lógica de negocio sin necesidad de levantar bases de datos o frameworks.
- **Flexibilidad tecnológica**: puedes cambiar la base de datos, UI o framework sin afectar la lógica.
- **Claridad en capas**: cada módulo tiene un propósito claro (Domain, Application, Infrastructure, Presentation).

**Desventajas**

- **Mayor complejidad inicial**: requiere más disciplina y diseño desde el inicio.
- **Curva de aprendizaje**: no todos los equipos están acostumbrados a pensar en capas concéntricas.
- **Sobrecarga en proyectos pequeños**: puede ser excesivo si el proyecto es muy simple.
- **Necesidad de patrones extra**: se suelen usar interfaces, inyección de dependencias y DTOs, lo que añade complejidad.

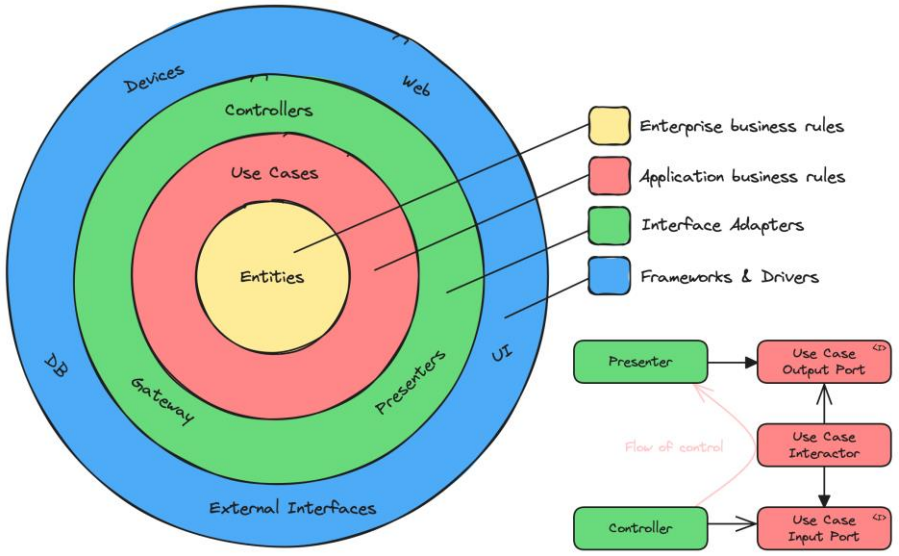
# ONION

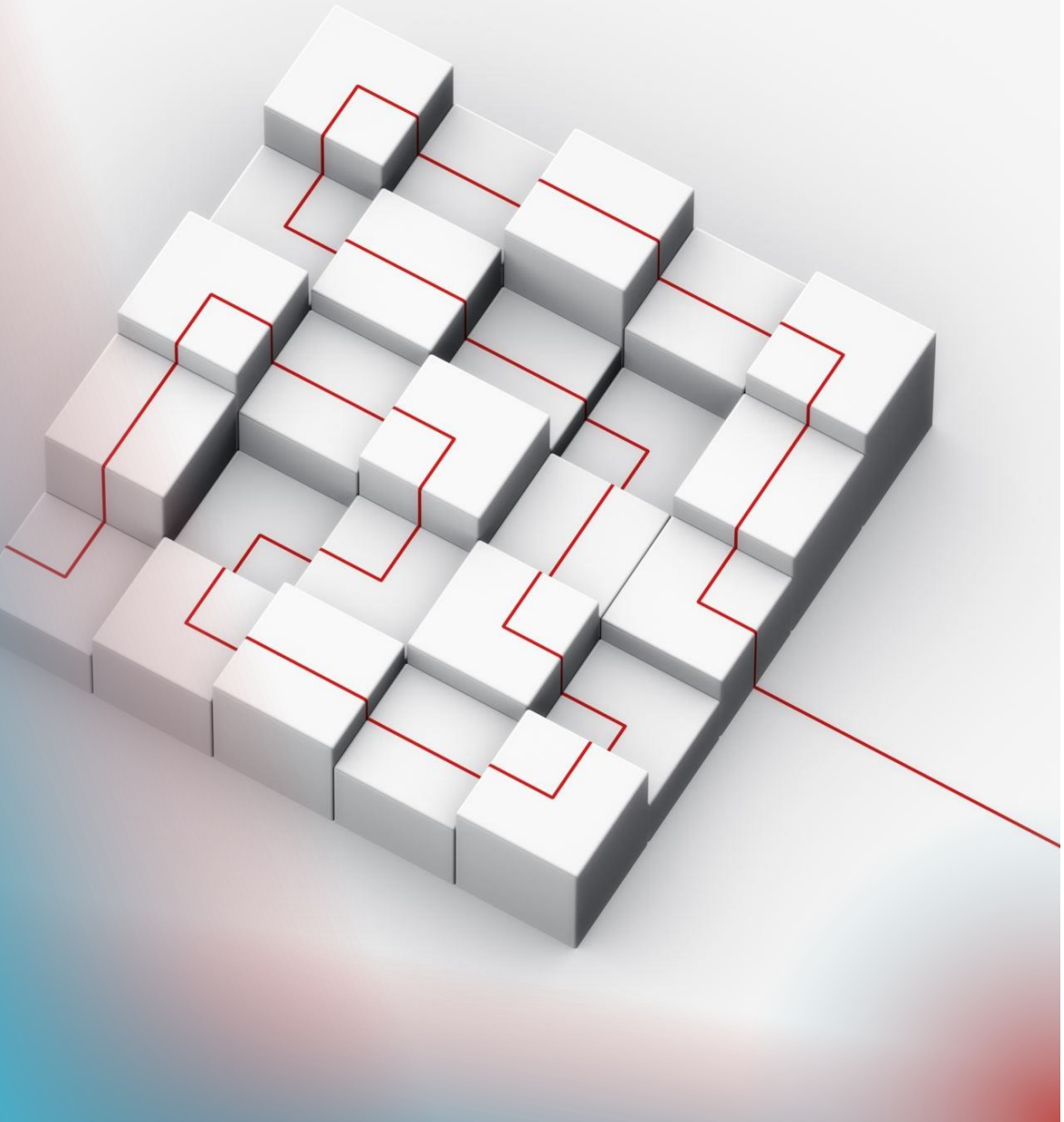
**Casos recomendados donde aplicar**

- Sistemas de **larga vida útil**, donde la evolución tecnológica es esperada.
- Aplicaciones empresariales con **reglas de negocio complejas**.
- Proyectos donde se planea **migrar infraestructura** (ej: pasar de Oracle a PostgreSQL, de REST a GraphQL).
- Aplicaciones que requieren **alta testabilidad** (finanzas, banca, salud).
- Arquitecturas orientadas a **DDD (Domain Driven Design)**.

No se recomienda en **proyectos muy pequeños o prototipos rápidos**, porque la sobrecarga de capas no se justifica.

# Clean


Devices
Web
Controllers
Use Cases
Entities
DB
Gateway
Presenters
UI
External Interfaces
Enterprise business rules
Application business rules
Interface Adapters
Frameworks & Drivers
Presenter
Use Case Output Port
Flow of control
Use Case Interactor
Controller
Use Case Input Port



# Clean

**¿Qué es?**

- Clean Architecture es un estilo arquitectónico propuesto por Robert C. Martin (Uncle Bob) que organiza el código en capas concéntricas donde **las reglas de negocio son independientes de frameworks, bases de datos y detalles externos**. La dependencia siempre fluye hacia el centro, es decir, el **núcleo (entidades y casos de uso)** no conoce nada de las capas externas.

# Clean

**Metáfora**

- Imagina un **castillo medieval**:
- El **núcleo** (entidades) es el **rey y sus leyes**, lo más importante y protegido.
- Los **casos de uso** son los **consejeros y ministros**, que organizan cómo aplicar esas leyes en la práctica.
- Los **adaptadores de interfaz** son las **puertas, puentes y embajadores**, que permiten la comunicación con el exterior (UI, controladores, gateways).
- Los **frameworks y drivers** son las **murallas externas y soldados**, reemplazables y solo cumplen una función de soporte.
- Si las murallas caen (framework cambia, DB cambia, UI cambia), el **núcleo sigue intacto y funcionando**.

# Clean

**Objetivo**

- Separar reglas de negocio del detalle técnico.
- Facilitar cambios de frameworks o tecnologías sin tocar el core.
- Crear sistemas mantenibles, testeables y con bajo acoplamiento.

# Clean

**Problema que soluciona**

- En muchos proyectos, el código de negocio queda acoplado a frameworks (Spring, Angular, Django), bases de datos o librerías externas. Esto genera **rigidez, alto costo de mantenimiento y dificultad para probar el sistema sin infraestructura real**.
- Clean Architecture soluciona esto al **independizar el dominio** de cualquier detalle de infraestructura.

# Clean

**Ventajas**

- **Independencia de frameworks**: puedes reemplazar Spring, Angular o Hibernate sin reescribir la lógica de negocio.
- **Alta testabilidad**: puedes probar las reglas de negocio sin necesitar base de datos ni UI.
- **Escalabilidad y mantenibilidad**: se adapta mejor a proyectos grandes y de larga vida.
- **Reutilización**: las reglas de negocio pueden reutilizarse en diferentes interfaces (ejemplo: web, móvil, APIs).
- **Seguridad del núcleo**: la parte más valiosa (dominio) está protegida del caos de cambios tecnológicos.

**Desventajas**

- **Curva de aprendizaje alta**: requiere disciplina y entendimiento profundo.
- **Sobre-ingeniería en proyectos pequeños**: puede ser excesivo si el sistema es simple.
- **Mayor esfuerzo inicial**: se necesita tiempo para estructurar bien las capas y definir contratos (puertos y adaptadores).
- **Más clases y capas**: puede parecer complejo y "verboso" para equipos novatos.

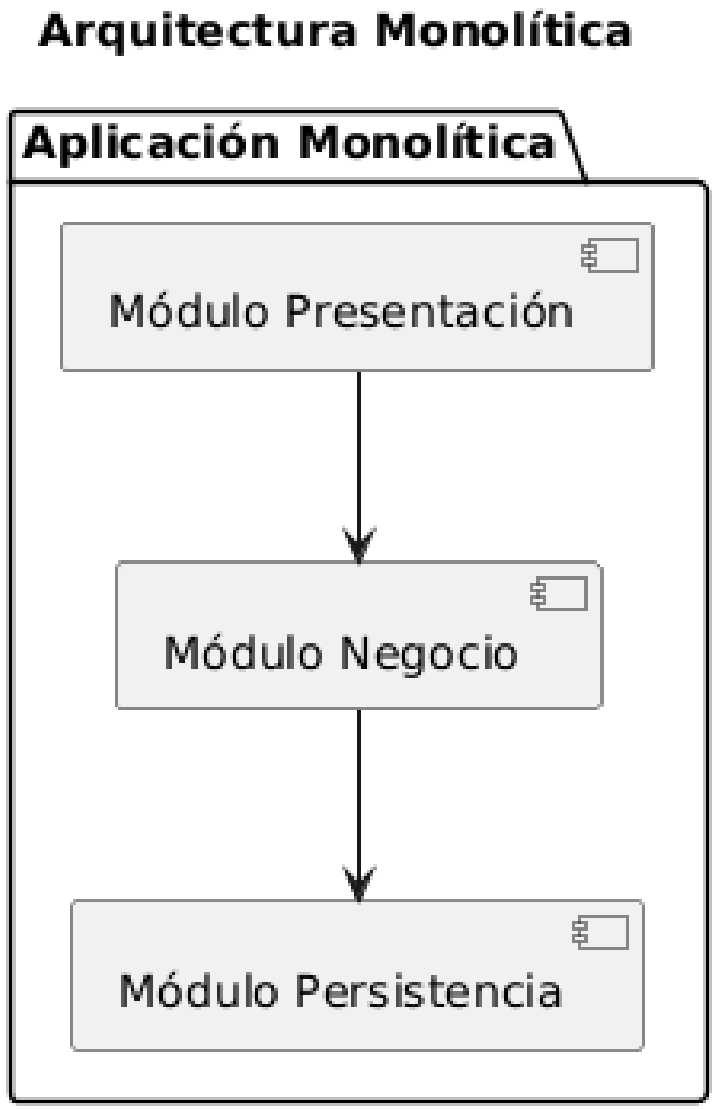
# Clean

- **Casos recomendados donde aplicar**

  - **Sistemas empresariales grandes** con lógica compleja y larga vida útil (banca, salud, gobierno).
  - **Aplicaciones multiplataforma** donde la lógica de negocio debe usarse en web, móvil, API, etc.
  - **Microservicios** que necesitan independencia tecnológica y alta testabilidad.
  - **Proyectos donde la base de datos o frameworks pueden cambiar** en el tiempo.

No se recomienda para **proyectos muy pequeños, prototipos rápidos o MVPs**, donde la simplicidad y velocidad importan más que la arquitectura.

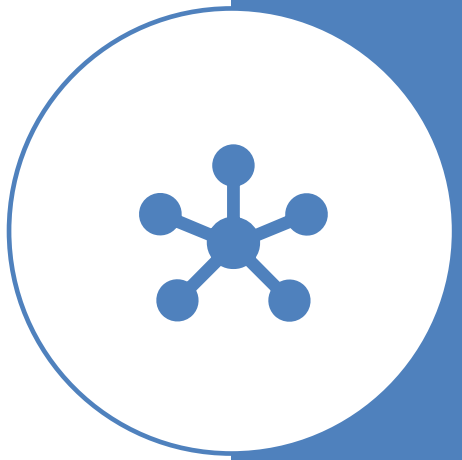
# Estilo Monolítico



# Estilo Monolítico

**¿Qué es?**

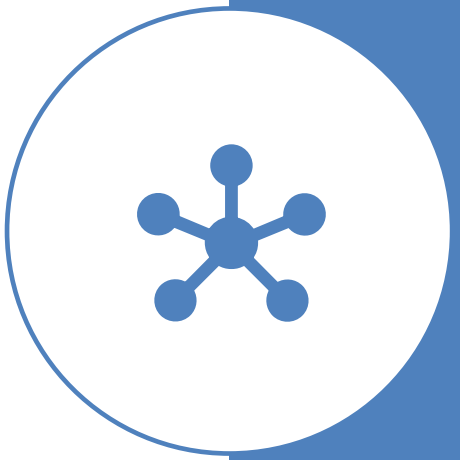
El estilo monolítico es una forma de desarrollar aplicaciones donde **todo el sistema se construye y despliega como una única unidad**. Dentro de ese bloque se incluyen la capa de presentación, la lógica de negocio y la persistencia de datos, todas interconectadas y empaquetadas juntas.



# Estilo Monolítico

**Metáfora**

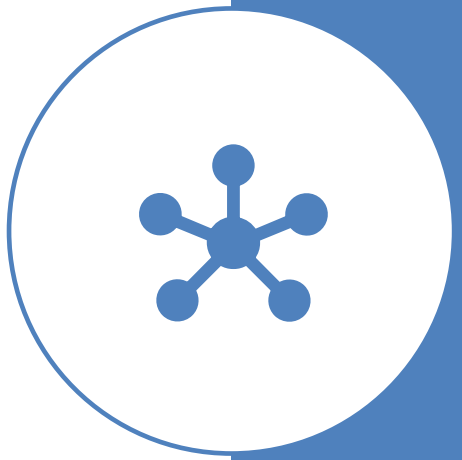
Imagina un **edificio de una sola pieza de concreto**: todo está unido, desde los cimientos hasta el techo. Si necesitas cambiar una ventana o una puerta, debes trabajar sobre toda la estructura, porque no está pensada para separar sus partes.



# Estilo Monolítico

**Objetivo**

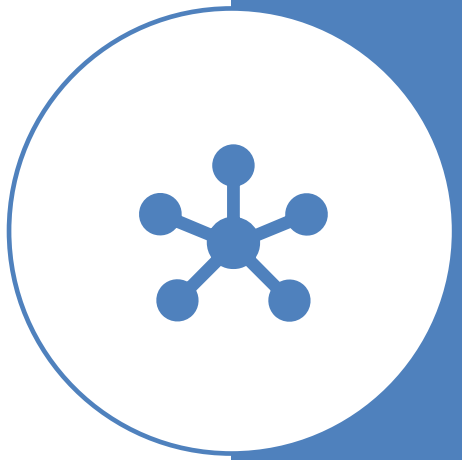
El objetivo principal de este estilo es **simplificar el desarrollo y despliegue** en sus fases iniciales, concentrando todo en un solo artefacto ejecutable, lo que facilita la puesta en marcha rápida de proyectos pequeños o medianos.



# Estilo Monolítico

**Qué problema solucionan**

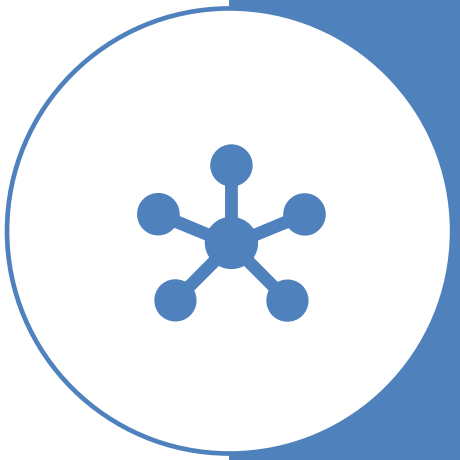
- **Rapidez inicial**: Permite crear un sistema funcional en menos tiempo y con menor complejidad técnica.
- **Simplicidad**: Los equipos pequeños pueden manejar todo el sistema en un solo proyecto sin necesidad de orquestar múltiples servicios.
- **Coherencia**: La lógica de negocio, la persistencia y la interfaz están fuertemente integradas, lo que hace que todo se comunique de manera directa.



# Estilo Monolítico

**Ventajas**

- **Desarrollo más rápido al inicio**: No hay necesidad de configurar arquitecturas distribuidas.
- **Despliegue sencillo**: Un solo artefacto que se despliega en un servidor.
- **Menor curva de aprendizaje**: Más fácil para desarrolladores junior o equipos pequeños.
- **Facilidad de testing local**: Al tener todo en un mismo paquete, las pruebas pueden ejecutarse en un único entorno.



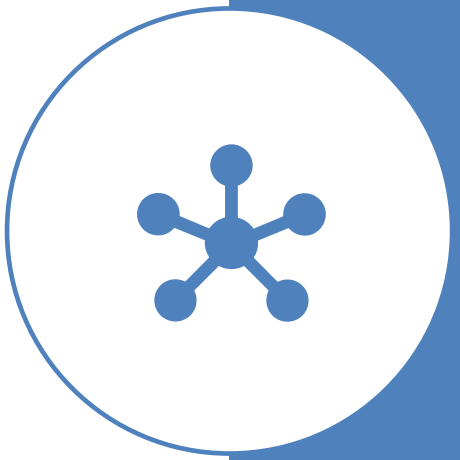
**Desventajas**

- **Escalabilidad limitada**: Todo el sistema debe escalarse junto, incluso si solo una parte necesita más recursos.
- **Mantenimiento complejo**: Con el crecimiento, el código tiende a convertirse en un “monstruo” difícil de entender y modificar.
- **Baja resiliencia**: Un fallo en un módulo puede tumbar toda la aplicación.
- **Dificultad para adoptar nuevas tecnologías**: Cambiar la base tecnológica requiere modificar todo el bloque.
- **Ciclo de despliegue lento**: Incluso pequeños cambios obligan a recompilar y desplegar toda la aplicación.

# Estilo Monolítico

**Casos recomendados donde aplicar**

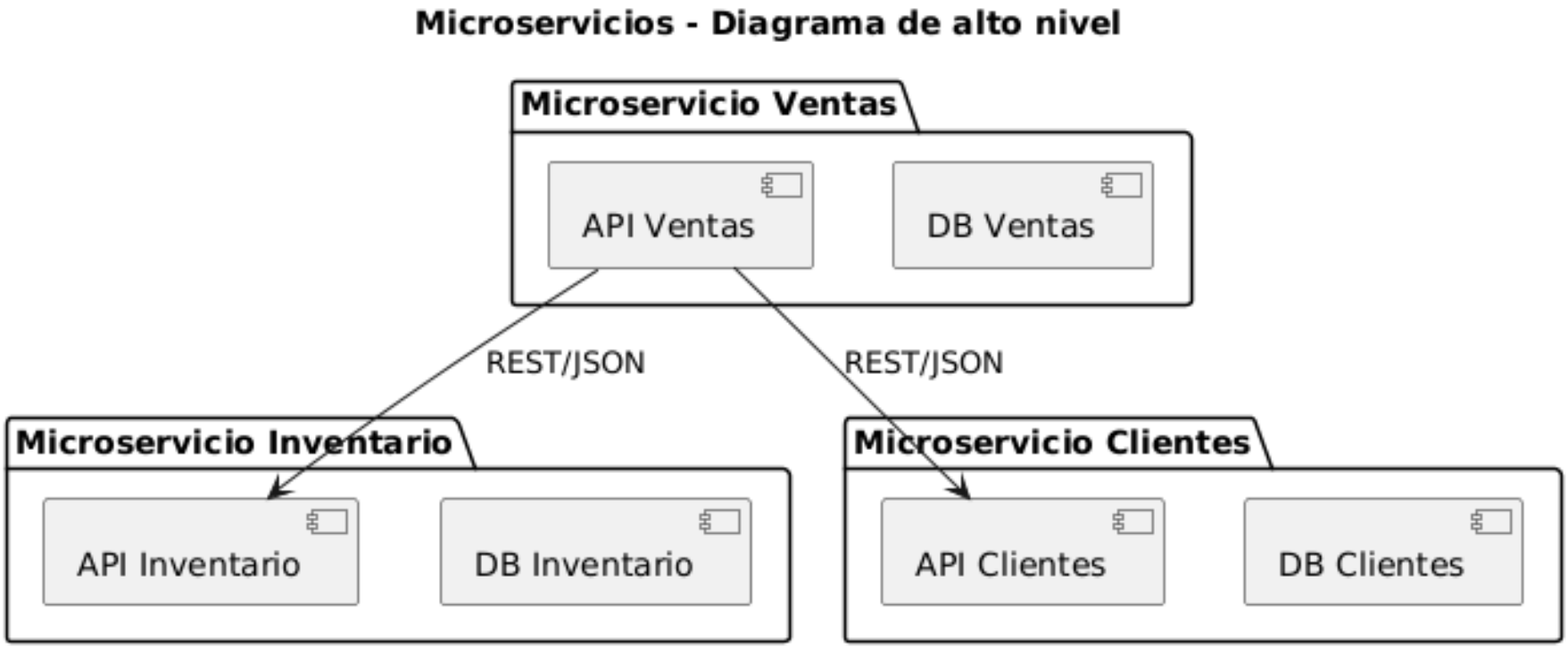
- **Proyectos pequeños o prototipos**: Donde el foco está en validar una idea rápida.
- **Equipos reducidos**: Donde mantener microservicios sería demasiado costoso.
- **Aplicaciones internas** con bajo tráfico y poco crecimiento esperado.
- **Sistemas académicos o de aprendizaje**: Para enseñar conceptos básicos de capas (presentación, negocio, persistencia).



**¡Algo importante a considerar!**

El estilo monolítico **no es obsoleto**, simplemente tiene un alcance distinto. Para empresas en fase inicial o sistemas con pocas necesidades de escalabilidad, es **más económico y eficiente**. El reto es saber cuándo migrar a un estilo más distribuido (como microservicios) antes de que el monolito se convierta en un obstáculo.

# Estilo Microservicios



# Estilo Microservicios

**¿Qué es?**

El estilo de **Microservicios** es una forma de diseñar aplicaciones como un conjunto de pequeños servicios independientes, cada uno con su propia lógica y base de datos.

Cada microservicio se comunica con los demás mediante APIs ligeras (generalmente **REST/JSON** o mensajería). En lugar de una gran aplicación monolítica, tenemos **varias piezas que trabajan juntas**, pero de forma autónoma.

# Estilo Microservicios

**Metáfora**

Imagina un **restaurante**:

El **chef de pizzas** solo cocina pizzas.
El **chef de postres** solo hace postres.
El **chef de bebidas** solo prepara bebidas.

Cada uno tiene su propio espacio, ingredientes y especialidad.

Cuando llega un pedido, trabajan de manera independiente pero coordinada para entregar un menú completo.

Eso es exactamente lo que hacen los microservicios en un sistema.

# Estilo Microservicios

**Objetivo**

- Dividir un sistema grande en **módulos independientes y manejables**.
- Permitir que cada equipo pueda desarrollar, desplegar y escalar su parte sin afectar a los demás.
- Mejorar la **agilidad** y la **resiliencia** del sistema.

# Estilo Microservicios

**Problema que solucionan**

- Evitan la rigidez del **monolito**, donde un pequeño cambio obliga a desplegar toda la aplicación.
- Reducen el **riesgo de caídas masivas**, ya que si falla un microservicio, el resto puede seguir funcionando.
- Facilitan la **escalabilidad selectiva**, permitiendo dar más recursos solo a los módulos más demandados.

# Estilo Microservicios

**Ventajas**

- **Independencia tecnológica**: cada microservicio puede desarrollarse con diferentes lenguajes o bases de datos.
- **Despliegue autónomo**: se actualizan sin necesidad de tocar toda la aplicación.
- **Escalabilidad selectiva**: puedes escalar solo lo que más tráfico recibe.
- **Mantenibilidad**: los equipos trabajan de forma más autónoma y ágil.
- **Resiliencia**: un fallo no necesariamente tumba todo el sistema.

# Estilo Microservicios

**Desventajas**

- **Complejidad en la comunicación**: necesitas manejar latencias, timeouts, fallos de red.
- **Gestión de datos distribuida**: cada servicio tiene su DB, lo que complica transacciones globales.
- **DevOps más exigente**: requiere CI/CD, monitoreo y logging distribuidos.
- **Sobrecoste inicial**: más difícil de implementar que un monolito simple.

# Estilo Microservicios

**Casos recomendados donde aplicar**

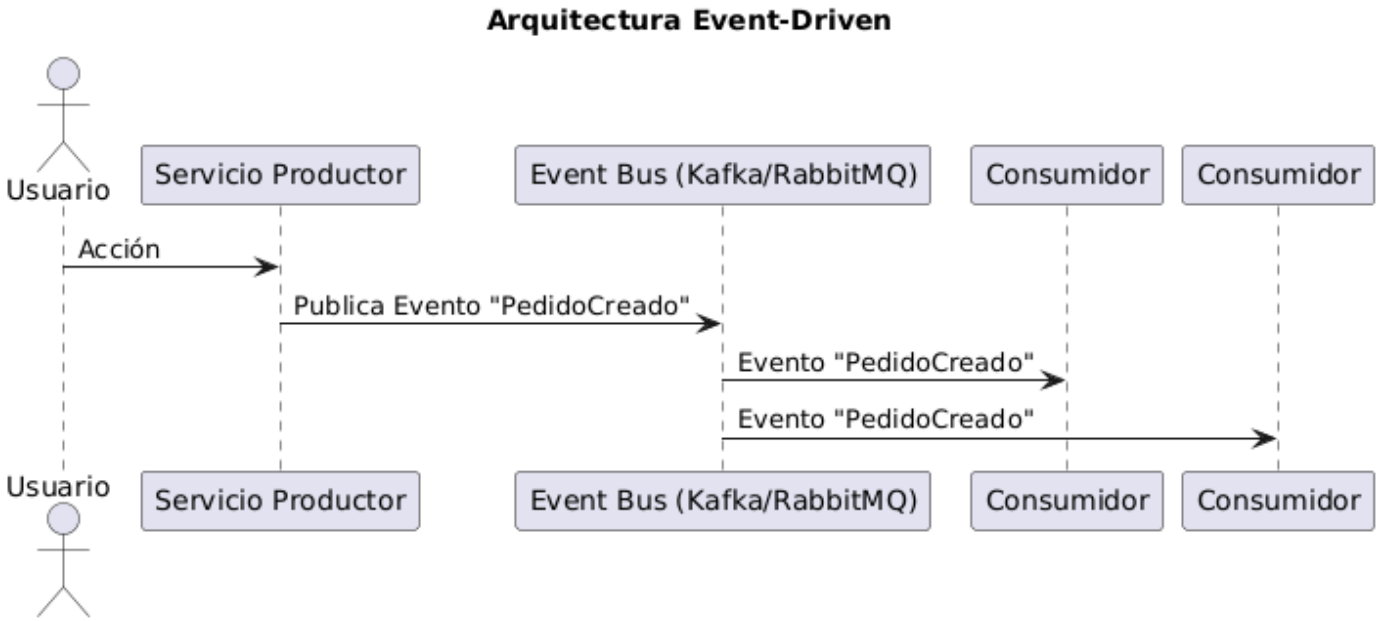
- Aplicaciones con **alto crecimiento esperado** en tráfico y funcionalidades.
- Sistemas que requieren **escalabilidad** por partes (ejemplo: módulo de pagos que crece más que el resto).
- Organizaciones con **equipos grandes y distribuidos**, donde cada equipo puede ser dueño de un microservicio.
- Proyectos que necesitan **alta disponibilidad y resiliencia** (e-commerce, banca digital, telecomunicaciones).

# Estilo Microservicios

**Algo que debes saber**

- Los microservicios casi siempre se apoyan en un **API Gateway** para centralizar el acceso, seguridad y balanceo.
- Son aliados naturales de la **nube** y **contenedores** (Docker + Kubernetes).
- Requieren buenas prácticas de **observabilidad** (logs, métricas, trazas distribuidas).
- No siempre son la mejor opción: para proyectos pequeños, un monolito modular puede ser más eficiente.

# Event-Driven Architecture



# Event-Driven Architecture

**¿Qué es la Arquitectura Event-Driven?**

Es un **estilo arquitectónico basado en eventos**, donde los sistemas reaccionan a sucesos (eventos) en lugar de seguir un flujo rígido de peticiones y respuestas.
En lugar de que un servicio llame directamente a otro, lo que hace es **publicar un evento** (ej. "PedidoCreado") en un **Event Bus** (Kafka, RabbitMQ, etc.), y los **consumidores** interesados reaccionan cuando reciben ese evento.

# Event-Driven Architecture

**Metáfora**

Imagina un **periódico**:

- El **periodista** (servicio productor) escribe una noticia (evento).
- El **periódico** (event bus) publica esa noticia.
- Los **lectores** (consumidores) que compran el periódico reciben la noticia y reaccionan según su interés.
  El periodista **no sabe quién leerá su artículo ni qué hará con él**, simplemente lo publica.

# Event-Driven Architecture

**Objetivo**

- Desacoplar los sistemas y permitir que se comuniquen de forma **asíncrona, escalable y flexible**.
- Facilitar la reacción en tiempo real ante cambios o sucesos en el sistema.

**Problemas que Soluciona**

- **Acoplamiento fuerte**: en lugar de que un servicio dependa directamente de otro, solo publica un evento.
- **Escalabilidad**: múltiples consumidores pueden escuchar un mismo evento sin sobrecargar al productor.
- **Extensibilidad**: si mañana agrego un nuevo consumidor, no tengo que modificar el productor.
- **Procesamiento en tiempo real**: ideal para sistemas que requieren reacción inmediata.

# Event-Driven Architecture

**Ventajas**

- **Desacoplamiento**: productores y consumidores no dependen directamente entre sí.
- **Escalabilidad horizontal**: más fácil añadir consumidores según la carga.
- **Asincronía**: mejora la velocidad y resiliencia del sistema.
- **Extensibilidad**: agregar nuevas funcionalidades sin tocar lo existente.
- **Resiliencia**: si un consumidor falla, el evento puede persistir en la cola y procesarse después.

**Desventajas**

- **Complejidad**: la arquitectura es más difícil de diseñar y monitorear.
- **Depuración complicada**: seguir el recorrido de un evento puede ser difícil.
- **Consistencia eventual**: los datos no siempre están sincronizados al instante.
- **Sobrecarga en infraestructura**: requiere herramientas como Kafka o RabbitMQ, que implican mantenimiento adicional.

# Event-Driven Architecture

**Casos Recomendados de Uso**

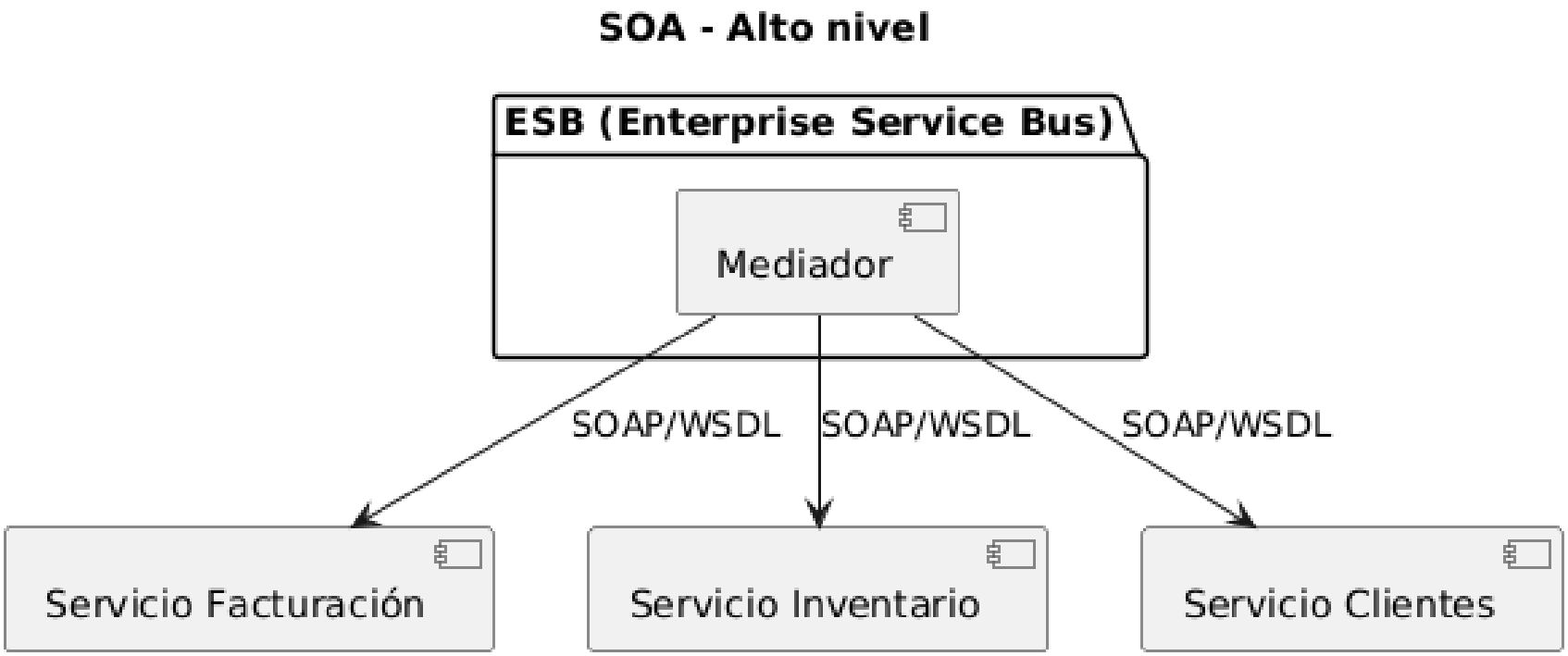
- **E-commerce**
  Evento: "PedidoCreado" → servicios de facturación, inventario y logística reaccionan automáticamente.
- **Banca y Finanzas**
  Evento: "TransacciónRealizada" → genera auditorías, envía notificaciones y actualiza balances.
- **IoT (Internet of Things)**
  Evento: "SensorTemperaturaAlerta" → activa alarmas, guarda en logs y dispara procesos.
- **Redes sociales**
  Evento: "NuevoPost" → notificaciones, recomendaciones y feeds se actualizan automáticamente.
- **Streaming de datos en tiempo real**
  Ejemplo: monitoreo de fraudes, detección de anomalías, analítica en tiempo real.

# Event-Driven Architecture

**¡Algo que es necesario saber!**

- **No todo sistema debe ser 100% Event-Driven**: en muchos casos se combinan **EDA + Request-Response** (ej. microservicios).
- **El bus de eventos es el corazón**: si falla Kafka/RabbitMQ, puede afectar a todo el sistema.
- **Buenas prácticas**:
  - Definir contratos claros para los eventos (nombre, estructura, payload).
  - Manejar idempotencia (evitar procesar el mismo evento varias veces).
  - Diseñar con trazabilidad y observabilidad para entender qué pasó con cada evento.

# Arquitectura SOA



# Arquitectura SOA

**¿Qué es?**

La **Arquitectura SOA** es un enfoque de diseño de software donde las funcionalidades de un sistema se dividen en **servicios independientes**, cada uno con una tarea específica (ejemplo: facturación, inventario, clientes). Estos servicios se comunican a través de estándares abiertos (SOAP, WSDL, XML) y se integran mediante un **ESB (Enterprise Service Bus)** que actúa como mediador.

**Metáfora**

Imagina una **empresa de mensajería**:

- El **ESB** es el **centro logístico** (almacén de distribución) que recibe, clasifica y redirige los paquetes.
- Los **servicios (facturación, inventario, clientes)** son como diferentes **oficinas especializadas**: cada una resuelve un problema en particular.
- El **SOAP/WSDL** serían las **guías de envío estandarizadas** que garantizan que todos los paquetes (mensajes) tengan el mismo formato y se entiendan entre oficinas.

# Arquitectura SOA

**Objetivo**

- Garantizar **interoperabilidad** entre sistemas heterogéneos.
- Permitir **reutilización de servicios** en distintos procesos de negocio.
- Centralizar la comunicación a través de un **bus empresarial** (ESB) que controla, transforma y enruta los mensajes.

**Problema que soluciona**

- La **comunicación entre aplicaciones diferentes** (ERP, CRM, sistemas legados).
- Evitar el desarrollo repetitivo de funciones (ejemplo: no crear facturación en cada sistema).
- Reducir la **complejidad de integración** en organizaciones grandes con muchos sistemas aislados.

# Arquitectura SOA

**Ventajas**

- **Reutilización**: los servicios pueden usarse en múltiples aplicaciones.
- **Escalabilidad**: es fácil agregar nuevos servicios.
- **Estandarización**: usa protocolos abiertos (SOAP, WSDL, XML).
- **Interoperabilidad**: conecta aplicaciones en distintos lenguajes y plataformas.
- **Flexibilidad**: permite reorganizar procesos de negocio cambiando la orquestación de servicios.

**Desventajas**

- **Complejidad**: requiere un ESB, lo que puede volver el sistema más complejo de mantener.
- **Costo**: implementación inicial alta (infraestructura y capacitación).
- **Rendimiento**: SOAP/XML es más pesado en comparación con REST/JSON.
- **Acoplamiento al ESB**: si falla, todo el ecosistema se ve afectado (punto único de fallo).

# Arquitectura SOA

**Casos recomendados donde aplicar**

- Grandes empresas con **sistemas heterogéneos** que necesitan comunicarse.
- Bancos, aseguradoras, gobiernos y corporaciones con sistemas legados.
- Proyectos donde la **estandarización y robustez** son más importantes que la velocidad de respuesta.
- Cuando se requiere una **alta capacidad de integración** entre diferentes plataformas y tecnologías.

**¡Algo necesario que saber¡**

- Aunque SOA fue muy usado en los 2000, hoy en día muchas empresas migran hacia **Microservicios** porque son más ligeros y usan **REST/JSON** o **gRPC** en lugar de SOAP.
- Sin embargo, **SOA sigue vigente** en entornos donde la **estabilidad y compatibilidad** con sistemas antiguos (legados) es crítica.

# Comparativa de Estilos Arquitectónicos

Monolítico: simple, pero poco escalable.

Microservicios: escalable, pero complejo.

Event-Driven: desacoplado, pero difícil de rastrear.

SOA: integrador, pero más pesado.

# Actividad Práctica Grupal

# Cierre y Conclusiones

Los patrones arquitectónicos son guías para estructurar sistemas.

Los estilos arquitectónicos definen cómo se organiza y comunica el sistema.

No existe un estilo único: depende del contexto y requisitos.

La elección correcta impacta en el mantenimiento, escalabilidad y éxito del proyecto.