4+1 — Desde los casos de uso a la arquitectura estructurada

Una buena arquitectura no puede ser descrita por un solo modelo. Necesitamos diferentes perspectivas para responder a diferentes preocupaciones.

— Philippe Kruchten, The Rational Unified Process, 1995

Objetivos de la Sesión

Al finalizar esta sesión, los estudiantes serán capaces de:

- **Comprender** la necesidad de múltiples vistas para describir sistemas complejos desde diversas perspectivas.
- **Explicar** el modelo 4+1 propuesto por Kruchten y sus componentes: lógica, proceso, desarrollo, despliegue y casos de uso.
- **Relacionar** los casos de uso previamente definidos con las vistas arquitectónicas que permiten implementarlos.
- **Identificar** los stakeholders clave que consumen cada vista y el propósito de comunicación técnica detrás de cada una.
- **Analizar críticamente** un caso de uso y esbozar su representación preliminar en al menos dos de las vistas del modelo 4+1.
- **Valorar** la trazabilidad entre necesidades funcionales (escenarios) y decisiones técnicas mediante un enfoque multivista.

¿Qué es el modelo 4+1?

¿Basta con los casos de uso para modelar una arquitectura completa?

"Una buena arquitectura no puede ser descrita por un solo modelo." — *Kruchten, 1995*

Una arquitectura de software necesita representar múltiples aspectos del sistema.

No existe una única vista que lo diga todo.

¿Dónde corre esto?

¿Hay concurrencia?

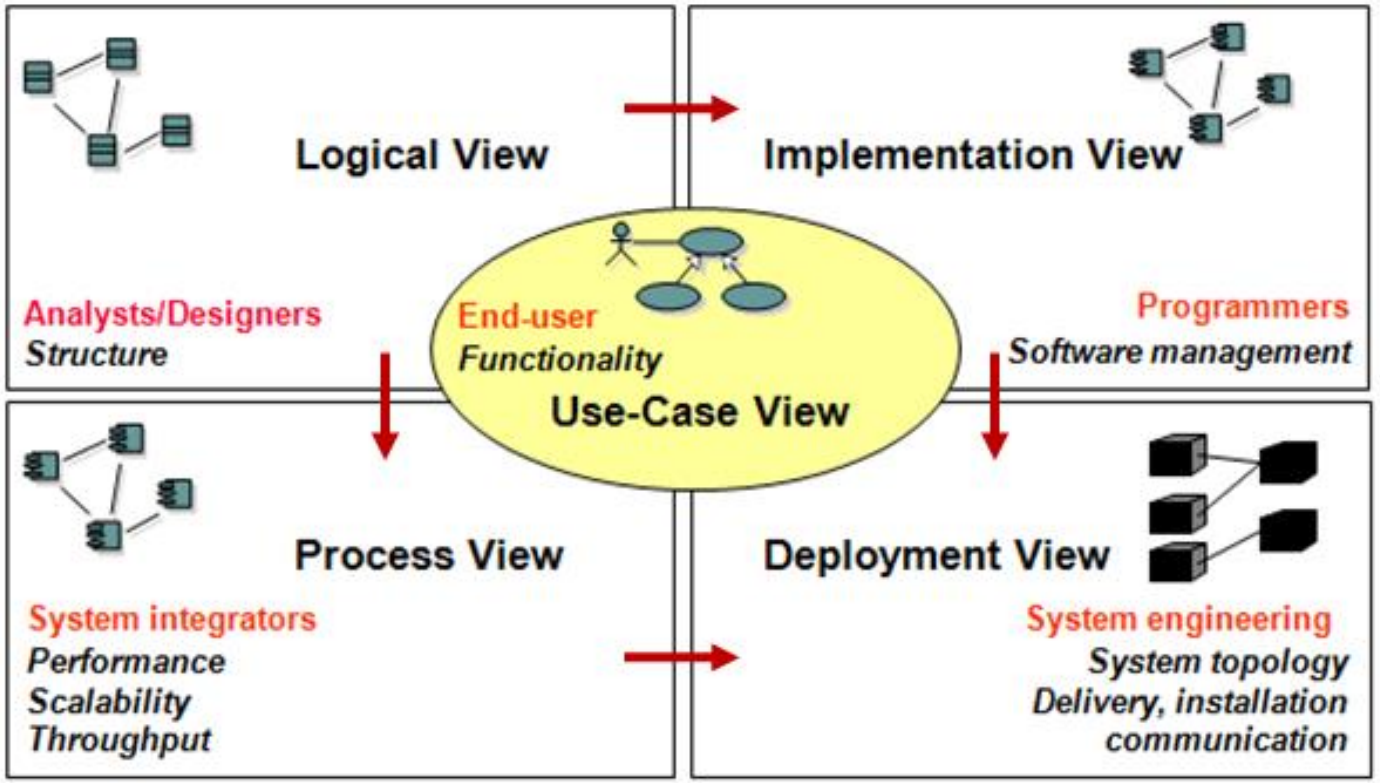
¿Cómo se despliega?

¿Está claro qué módulo lo implementa?

El modelo **4+1 vistas** fue propuesto por Philippe Kruchten en 1995 como parte de RUP (*Rational Unified Process*). Organiza la arquitectura en 4 vistas técnicas + 1 vista de casos de uso.

Componentes:

- Vista lógica
- Vista de procesos
- Vista de desarrollo
- Vista de despliegue
- +1: Escenarios (casos de uso que validan las anteriores)



Trazado multivista

Cada vista **responde a una pregunta distinta** y ayuda a **tomar decisiones distintas**.

Lógica: Pedido, Carrito, Usuario, PagoService

Desarrollo: módulo checkout/, paquete orders/, separación frontend/backend

Confirmar pedido

Proceso: validación del cupón (asíncrona), confirmación de pago (event-driven)

Despliegue: balanceador → API Gateway → microservicio checkout + base de datos

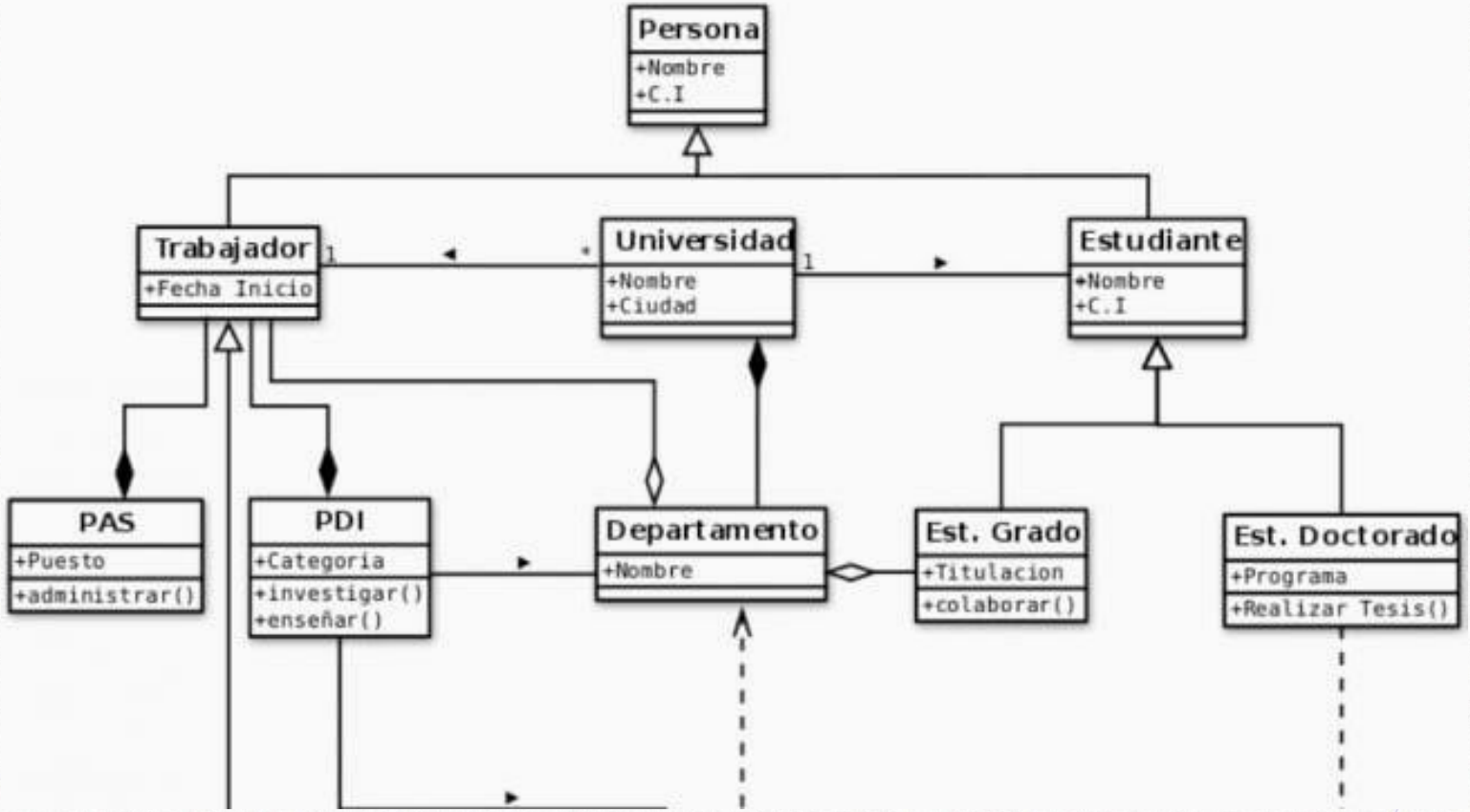
Vista LÓGICA (Logical View)

¿Qué muestra?:

La estructura del sistema desde la perspectiva del dominio funcional.

¿Para quién?: Desarrolladores.

- **Diagramas típicos**:
- UML **Diagramas de clases**
- UML **Diagramas de objetos**
- En DDD: *Modelos de agregados o entidades*



Vista de PROCESO (Process View)

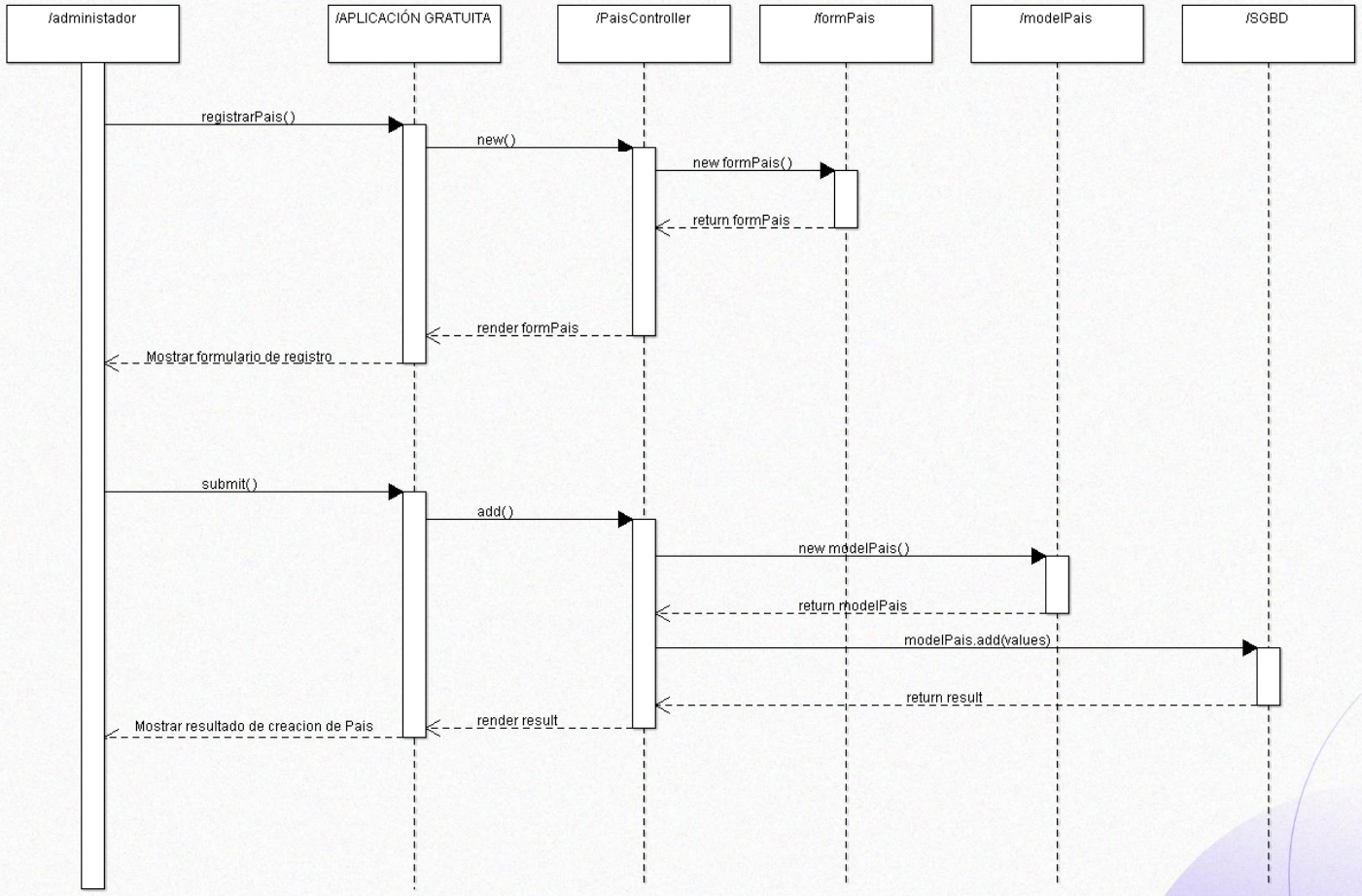
¿Qué muestra?:

Comportamiento dinámico, concurrencia, comunicación entre procesos.

¿Para quién?: Arquitectos técnicos, ingenieros de rendimiento.

Diagramas típicos:

- UML **Diagramas de secuencia**
- UML **Diagramas de actividad**
- **Diagramas de flujo de datos (DFD)**
- Diagramas con threads o colas (event loops)



Vista de DESARROLLO (Development/Implementation View)

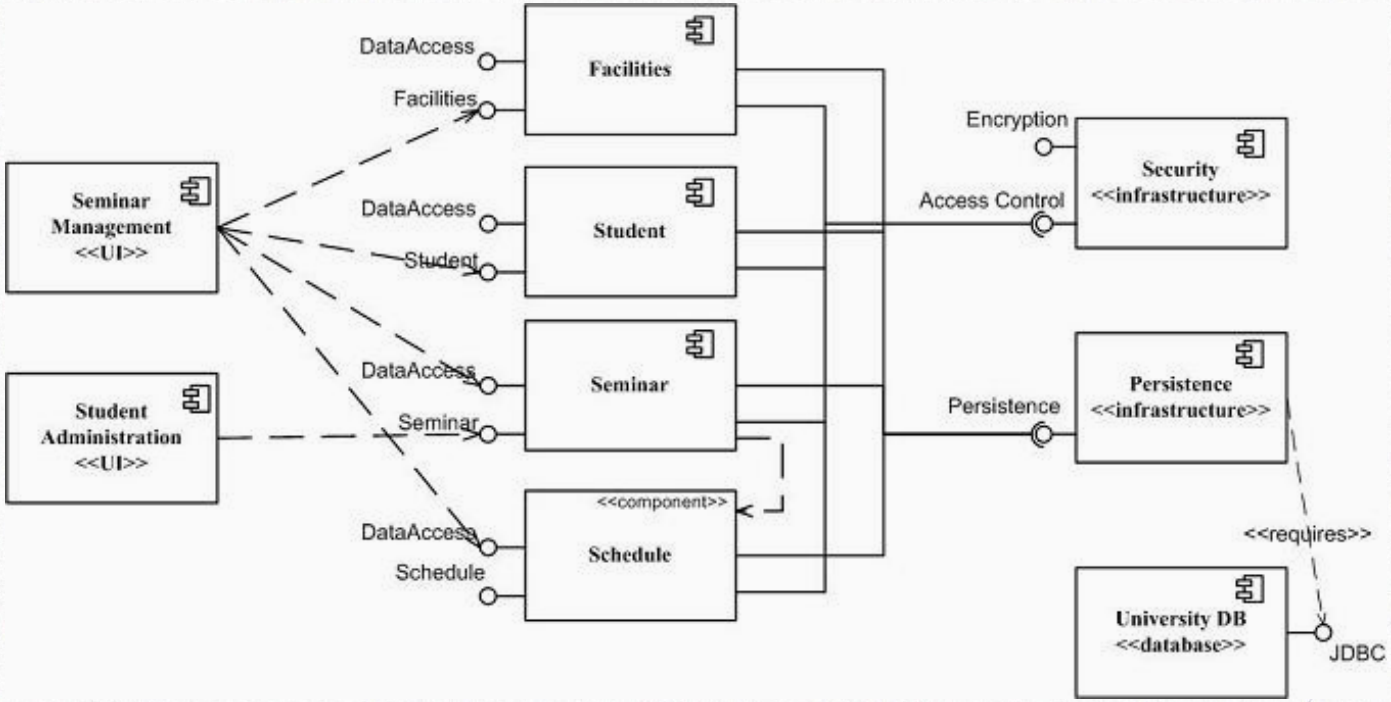
¿Qué muestra?:

La organización del código fuente, estructura de paquetes o módulos.

¿Para quién?: Líder técnico, equipo de desarrollo.

Diagramas típicos:

- UML **Diagramas de componentes**
- **Mapas de carpetas/repositorios**
- Representaciones como *arquitectura hexagonal*, *clean architecture*, etc.



Vista de DESPLIEGUE (Deployment View)

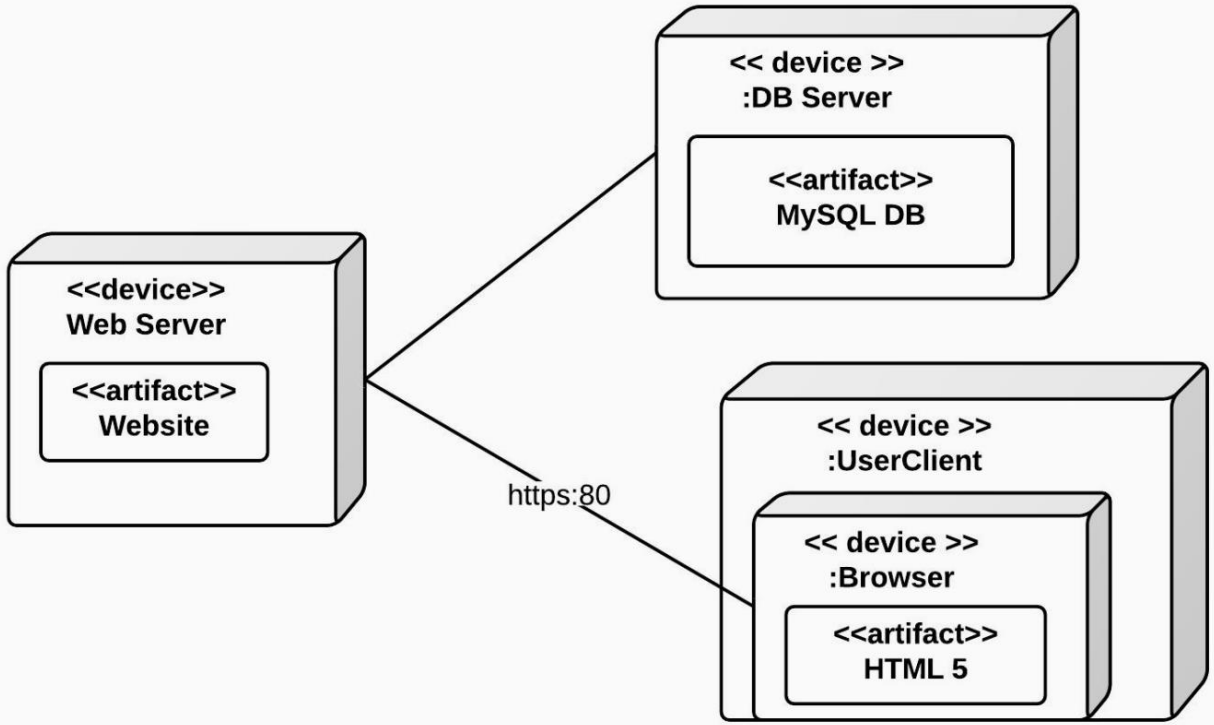
¿Qué muestra?:

Infraestructura física/virtual, nodos, redes, contenedores, cloud.

¿Para quién?: DevOps, ingenieros de plataforma.

Diagramas típicos:

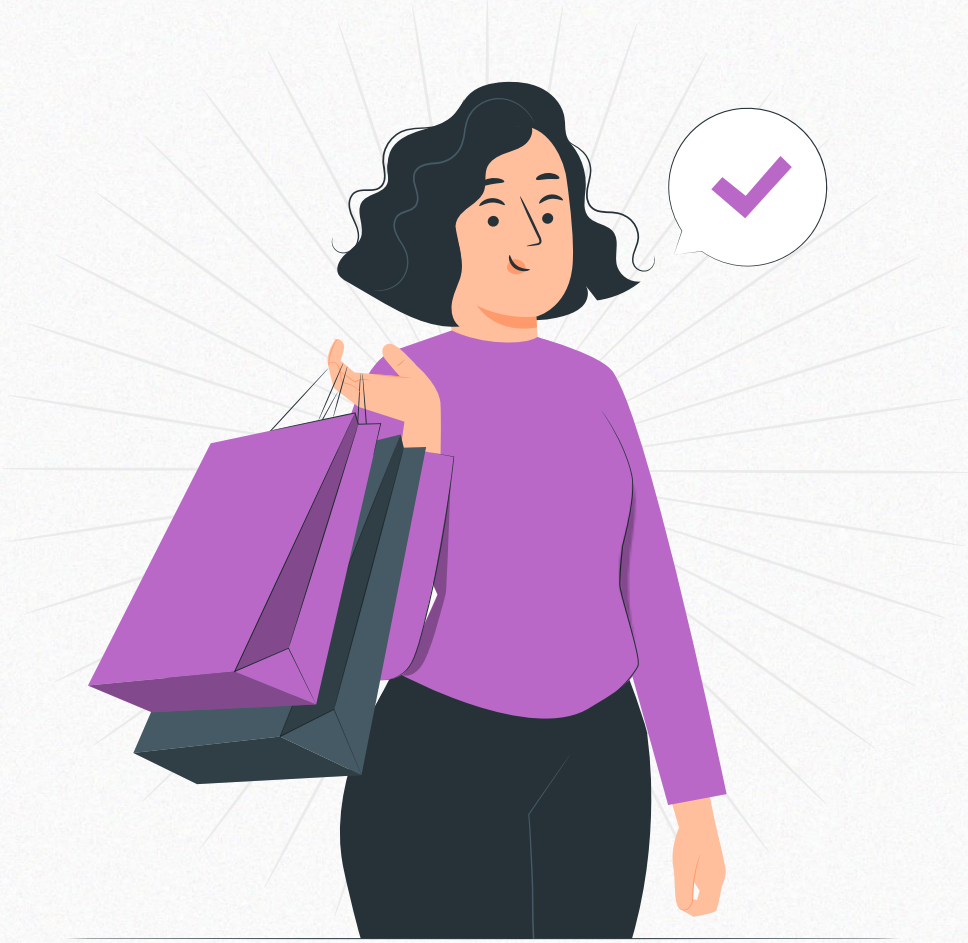
- UML **Diagramas de despliegue**
- **Arquitectura de microservicios (topología)**
- **Infraestructura como código (IaC): Terraform, CloudFormation**



Caso Practico

Caso de uso: *Confirmar pedido*

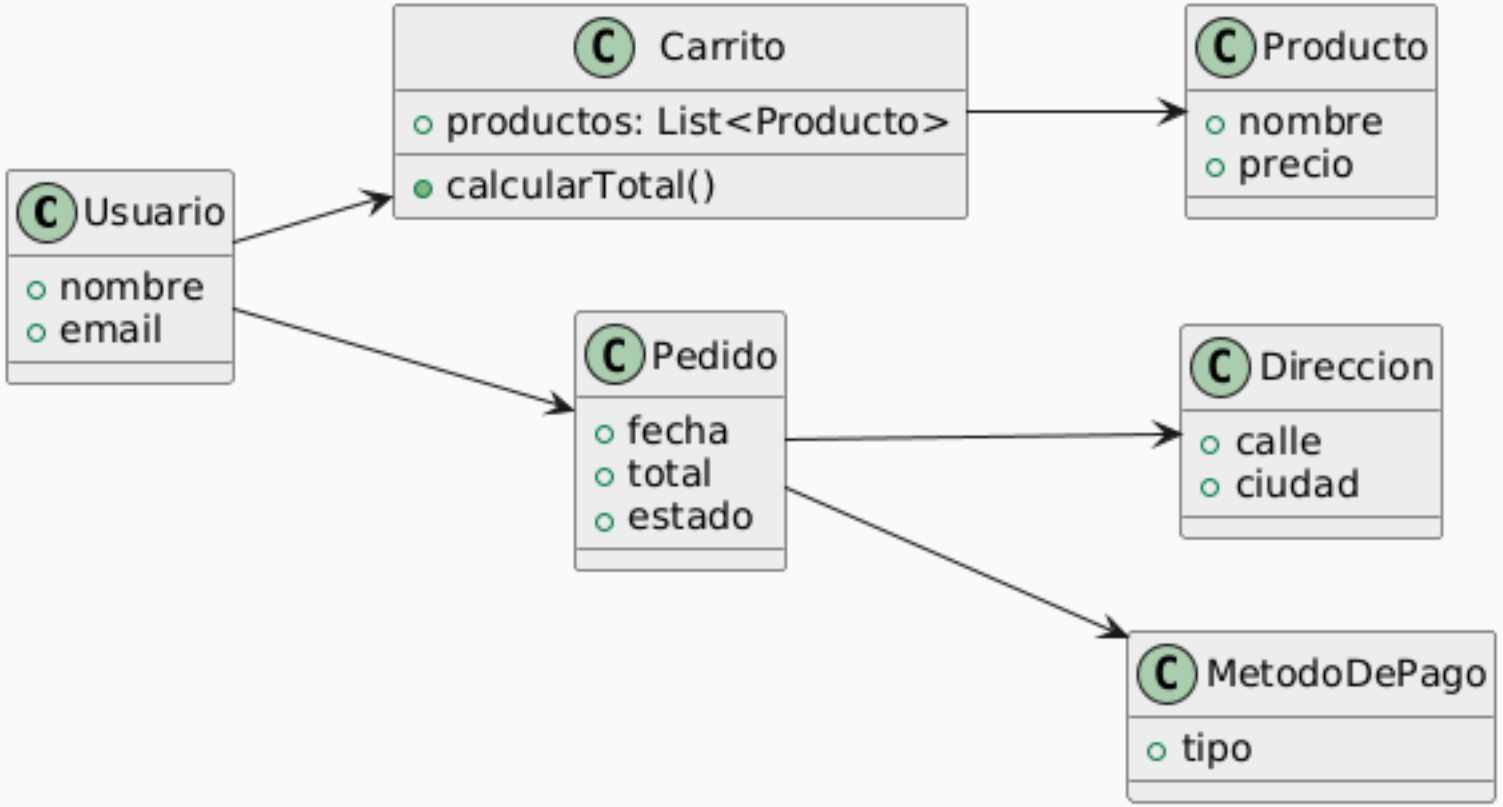
El usuario revisa su carrito, elige un método de pago, ingresa su dirección de envío y confirma el pedido. El sistema procesa la solicitud, registra la orden y muestra una pantalla de confirmación.



Vista Logica

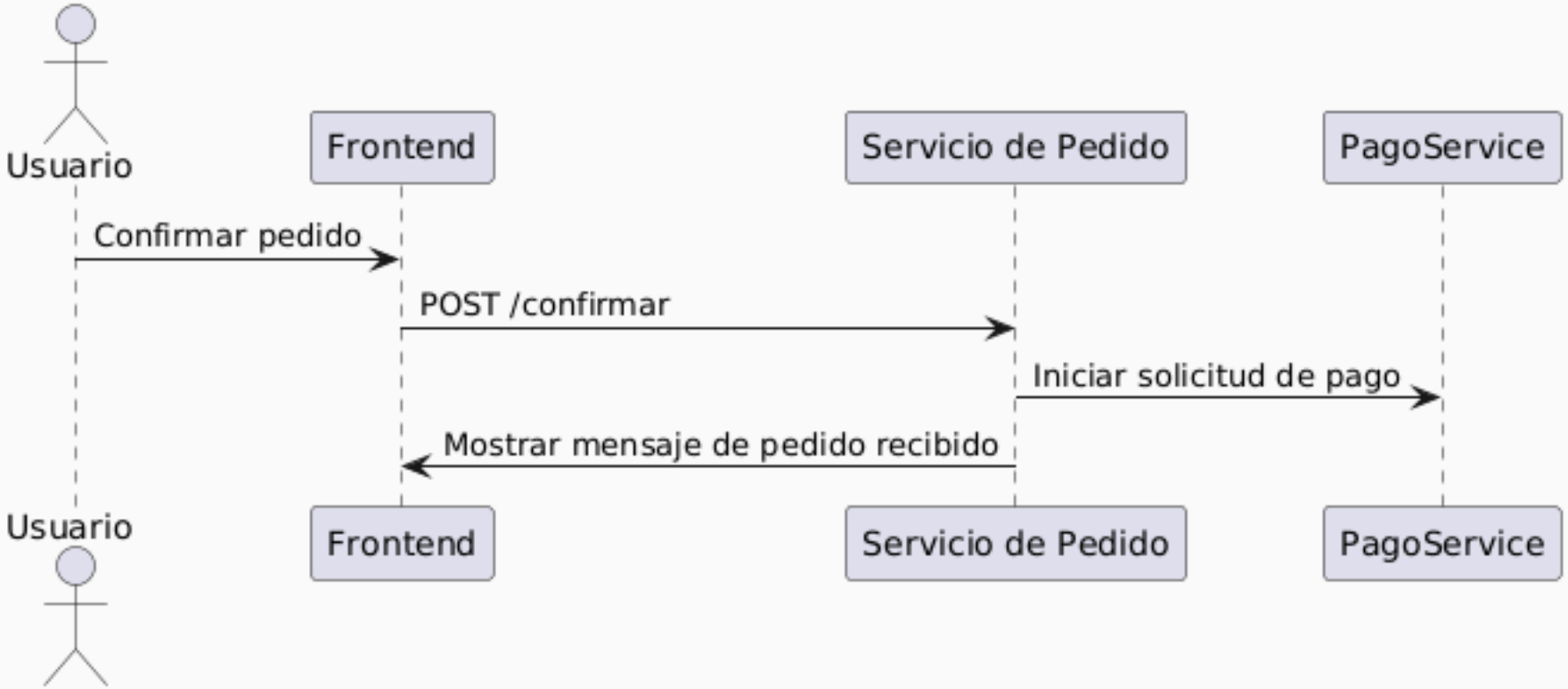
Representa las entidades y relaciones del dominio funcional. Es clave para el modelado orientado a objetos.

- Usuario interactúa con su Carrito.
- Carrito genera un Pedido, que finalmente se procesa mediante PagoService.
- Esta vista ayuda a estructurar el modelo de dominio y delimita responsabilidades.

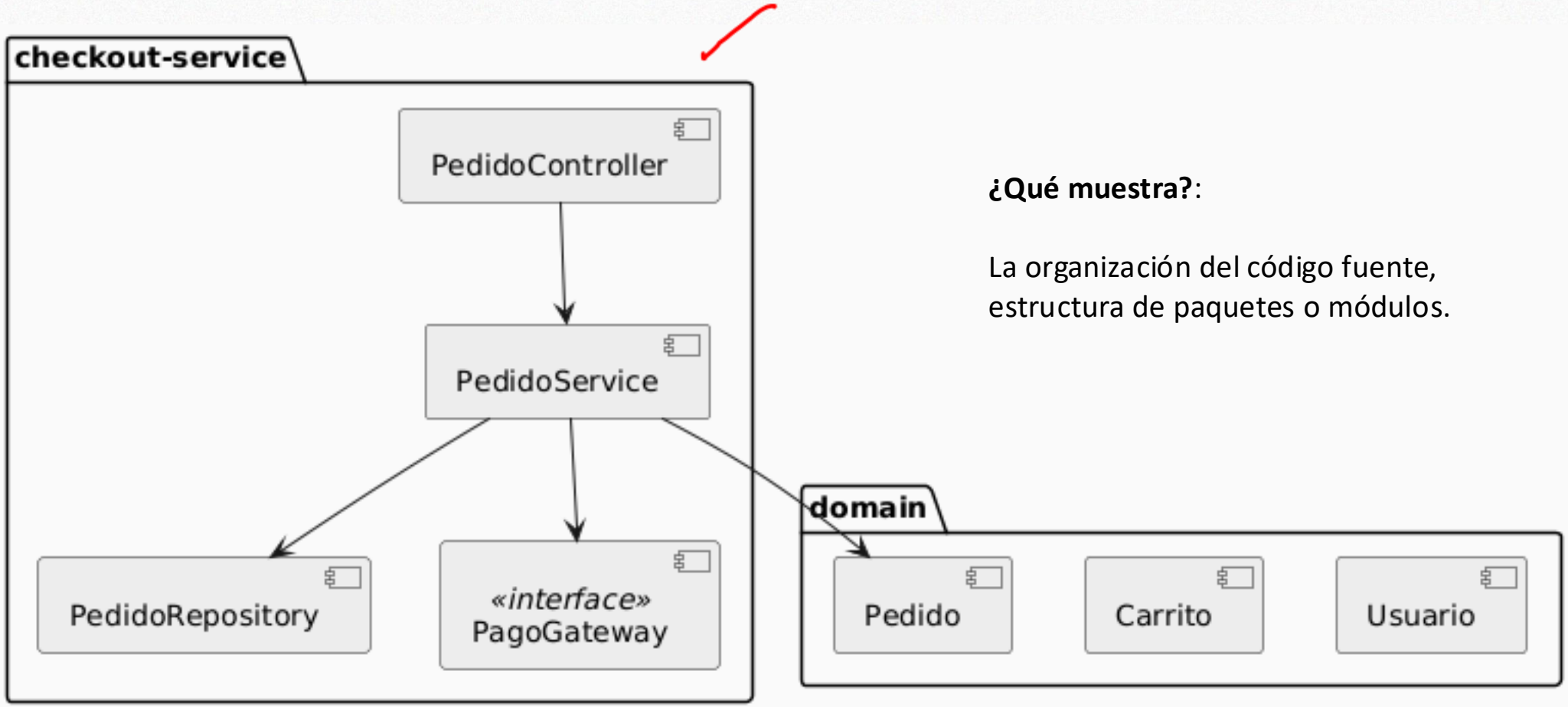


Vista de Proceso

Muestra comportamiento dinámico, concurrencia y comunicación. Útil para decisiones de rendimiento y tolerancia a fallos.



Vista de DESARROLLO
(Development/Implementation View)



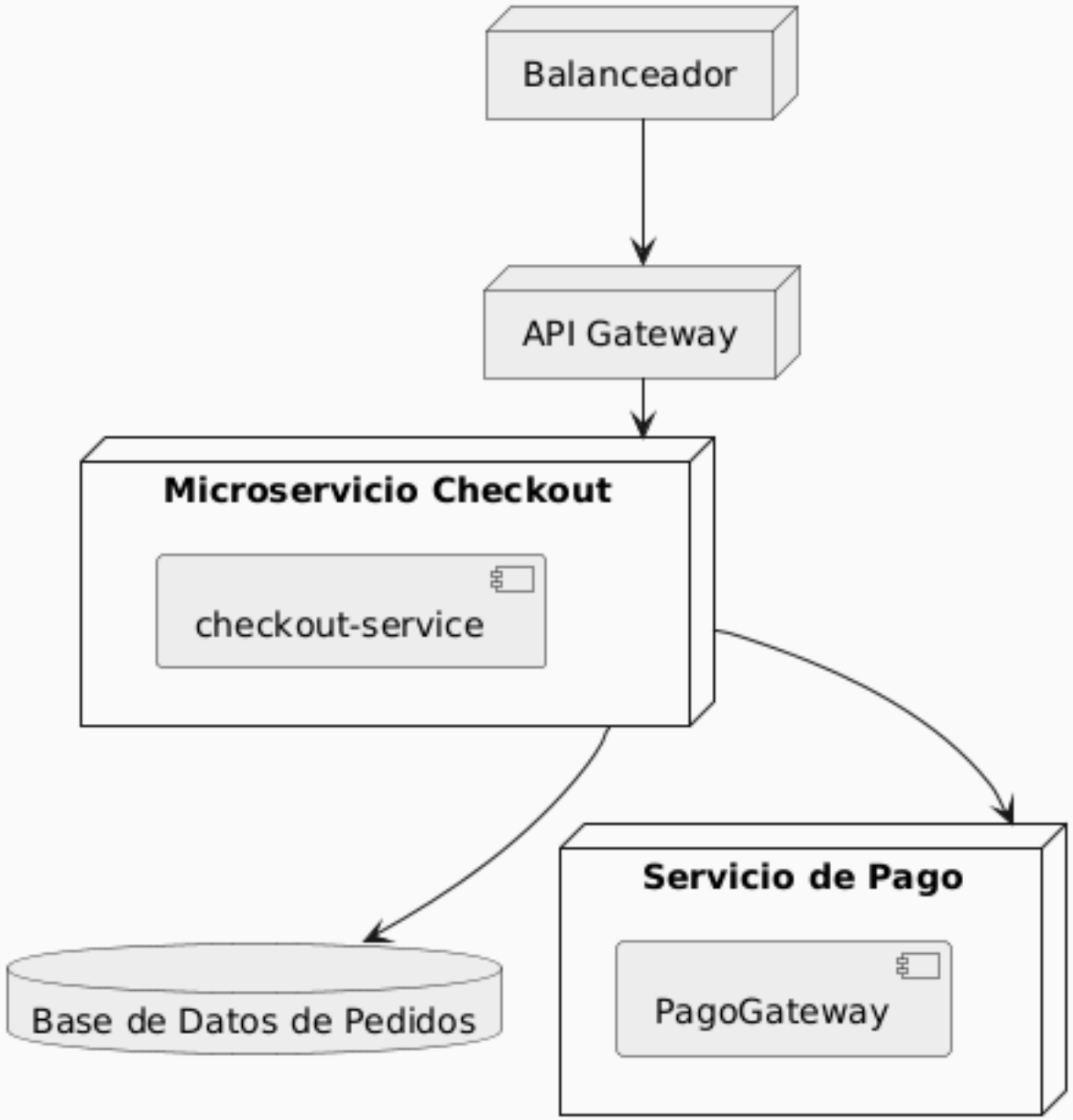
¿Qué muestra?:

La organización del código fuente, estructura de paquetes o módulos.

Vista de DESPLIEGUE (Deployment View)

¿Qué muestra?:

Infraestructura física/virtual, nodos, redes, contenedores, cloud.



Estilos de arquitectura clasicos

Una buena arquitectura no puede ser descrita por un solo modelo. Necesitamos diferentes perspectivas para responder a diferentes preocupaciones.

— Philippe Kruchten, The Rational Unified Process, 1995

¿Podrías describir la arquitectura de un sistema sin hablar de ninguna tecnología?

Un estilo arquitectónico define una familia de sistemas en términos de un vocabulario de componentes y conectores, y un conjunto de restricciones sobre cómo pueden ser combinados.

— Shaw & Garlan, 1996

Un estilo arquitectonico **NO ES:**

- Un framework (como Spring)
- No es un patrón (como Factory)
- No es una herramienta (como Kubernetes)

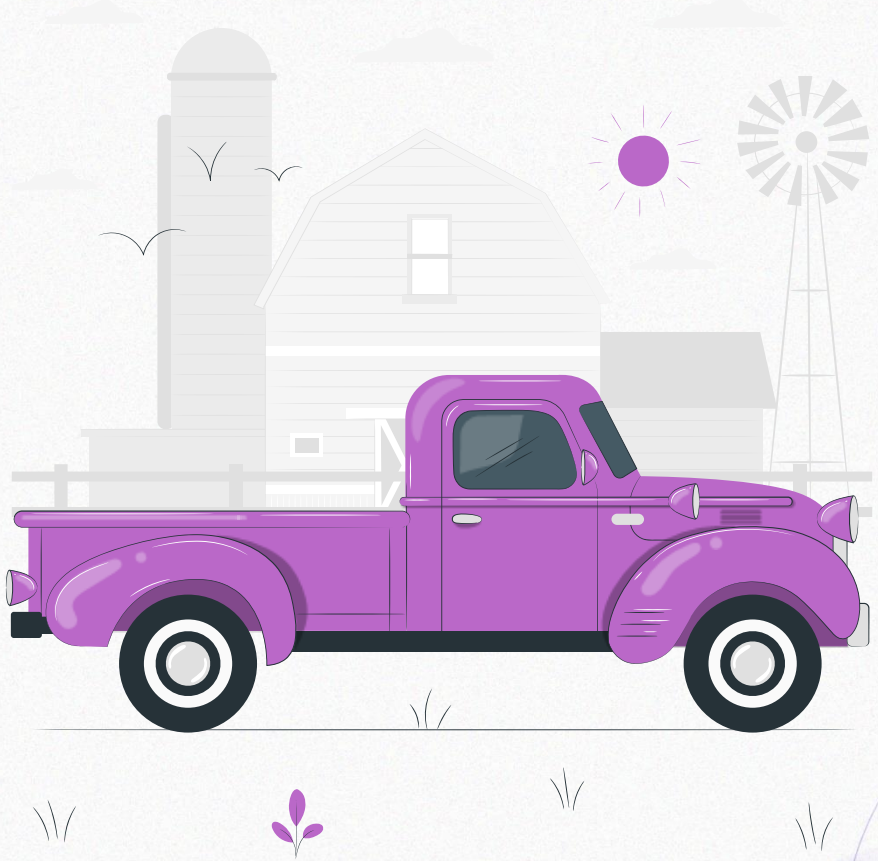
Recuerda:

- Un **estilo arquitectónico** = modelo general y estructural.
- Un **patrón** = solución recurrente a un problema específico (ej. MVC, Singleton).
- Una **tecnología** = herramienta concreta que puede implementar un estilo (ej. Django, Kubernetes).

Estilos Arquitectonicos Clasicos

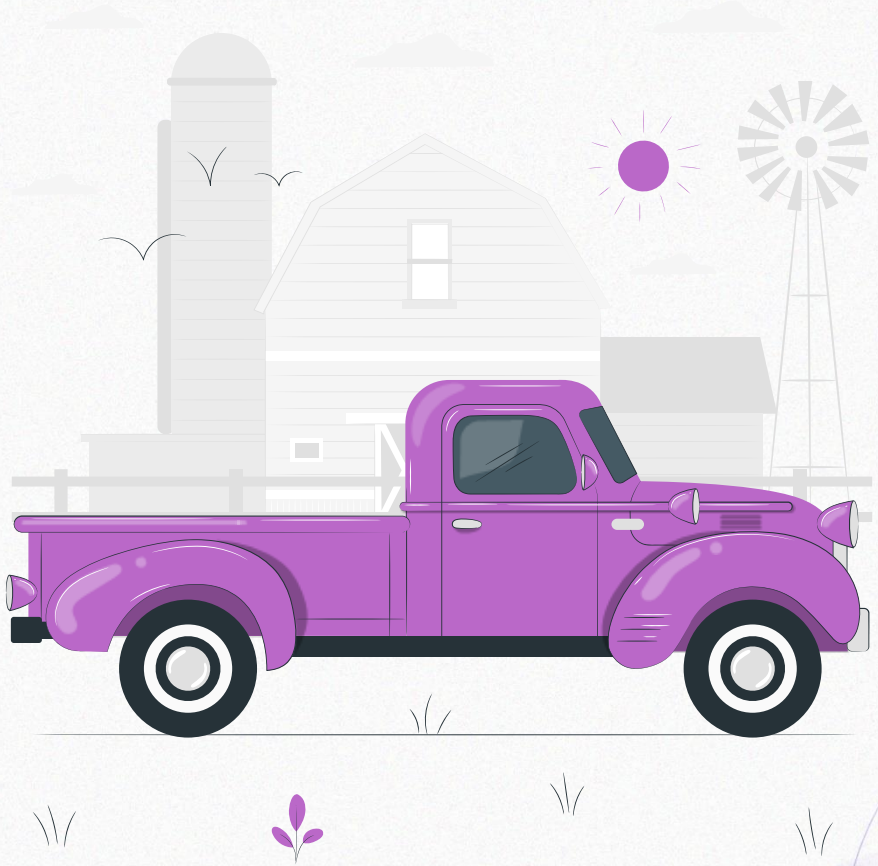
¿Por qué hablar de “clásicos”?

- Fueron los **primeros en sistematizarse académicamente**.
- Sirven como **fundamento conceptual** para estilos modernos (como microservicios o arquitecturas hexagonales).
- Permiten entender el **origen de muchas decisiones arquitectónicas actuales**, especialmente en migraciones o rediseños.



Estilos Arquitectonicos Clasicos

- Monolítico
- Cliente-servidor
- En capas (layered)
- Pipe and Filter
- Event-driven
- Microservicios



Monolítico

En una arquitectura monolítica, todos los componentes de un sistema de software se construyen y despliegan como una única unidad, que se ejecuta dentro de un mismo proceso o contenedor de ejecución.

— Adaptado de Bass, Clements & Kazman, *Software Architecture in Practice* (2021)

Monolítico

Un monolito no implica necesariamente “mala práctica” o “código espagueti”. Se refiere a **una unidad indivisible de despliegue**, no a la calidad del diseño interno.

Características estructurales

- Toda la lógica de negocio, la capa de presentación y el acceso a datos viven en **un solo ejecutable o aplicación empaquetada**.
- Generalmente se usa **una sola base de código**, aunque puede estar organizada en módulos internos.
- El despliegue y el ciclo de vida es **unitario**: se compila, prueba y despliega como un todo.
- La comunicación entre componentes internos es **directa (llamadas locales o en memoria)**.

Monolítico

Un monolito no implica necesariamente “mala práctica” o “código espagueti”. Se refiere a **una unidad indivisible de despliegue**, no a la calidad del diseño interno.

Ejemplos de la industria

- **Aplicaciones Java EE tradicionales** con JSF + EJBs + JPA.
- **Ruby on Rails** o **Laravel** cuando todo vive en un solo proyecto.
- **Sistemas heredados (legacy)** en bancos, ERP, sistemas clínicos

Monolítico

Un monolito no implica necesariamente “mala práctica” o “código espagueti”. Se refiere a **una unidad indivisible de despliegue**, no a la calidad del diseño interno.

Ventajas

- Simplicidad inicial
- Fácil de desarrollar
- Despliegue único
- Ideal para MVPs
- Transacciones más simples

Desventajas

- Difícil de escalar selectivamente
- Baja resiliencia
- Despliegue riesgoso
- Evolución lenta del equipo
- Tecnologías acopladas

Monolítico

¿Cuándo es razonable usar un monolito?

- Startups que lanzan un MVP y buscan **time-to-market** rápido.
- Equipos pequeños que **no justifican complejidad operativa**.
- Sistemas bien definidos y estables, con cambios poco frecuentes.
- Aplicaciones de línea de negocio interna **con bajo volumen de usuarios**.

Monolítico

Buenas practicas

- Diseñar el código **modularmente**, aunque se despliegue como un todo.
- Usar capas bien separadas (UI, servicios, dominio, persistencia).
- Implementar pruebas automatizadas (unitarias e integradas).
- Documentar explícitamente las **dependencias internas**.
- Dejar puntos de extensión (facades, adaptadores) que faciliten futuras migraciones.

Una startup lanza una app de delivery que funciona bien como monolito. A los 6 meses, quieren expandirse a 3 países con pagos, logísticas y monedas distintas.

Un banco tiene un core bancario monolítico con más de 20 años, que aún procesa millones de transacciones diarias sin errores.

¿Puedes diseñar un monolito bien estructurado que luego pueda evolucionar hacia microservicios?

¿Qué consecuencias puede tener forzar microservicios en un sistema que no necesita esa granularidad aún?

¿Cómo construir un buen monolito?

“Monolítico no significa caótico.”
Un buen monolito es modular,
testeable y evolutivo.

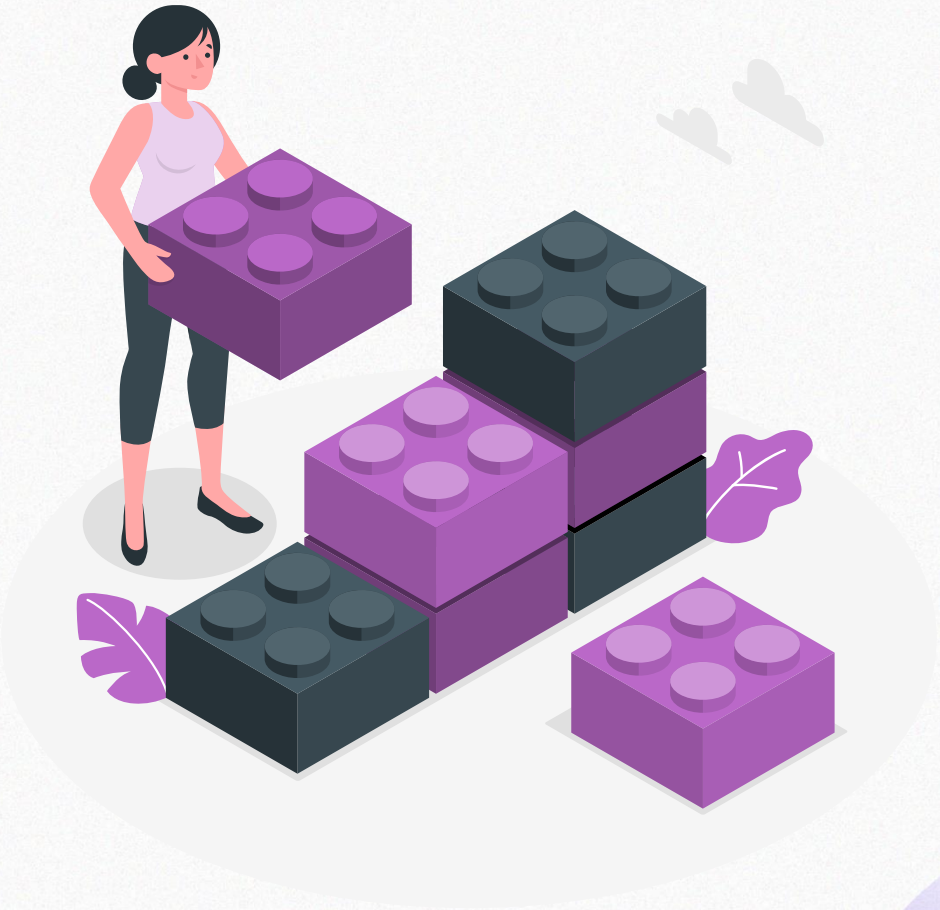
Principios clave para un monolito saludable

Modularidad interna: Separar claramente responsabilidades dentro del código.

Modularidad: División por **función o dominio**

Capas: División por **responsabilidad técnica y** Depende de **flujo vertical**

Richards & Ford (2020) lo llaman "estructura vertical vs. horizontal".



¿Cómo modularizar un monolito?

Separación por dominio funcional: Cada módulo representa una unidad del negocio (no técnica).

Ej: pedidos, productos, clientes, notificaciones.

Límites explícitoS: No se accede directamente a funciones de otros módulos. (Solo a través de contratos APIs internas).

Dirección de dependencias: Si A usa B, B no debe conocer a A. Usar inversión de dependencias si es necesario.

Evitar el “shared” tóxico No tener un módulo utils/ al que todos dependen.

¿Cómo modularizar un monolito?

```
# controller.py
def confirmar_pedido(usuario_id, producto_id):
    user = buscar_usuario(usuario_id)
    producto = buscar_producto(producto_id)
    if producto.stock > 0:
        registrar_pedido(usuario_id, producto_id)
```

controlador llama lógica de usuarios y productos directamente

¿Cómo modularizar un monolito?

```
módulo: pedidos/
   - servicio_confirmacion.py
módulo: usuarios/
   - api.py
módulo: productos/
   - api.py
```

```
# pedidos/servicio_confirmacion.py
from usuarios.api import obtener_usuario
from productos.api import obtener_producto


def confirmar(usuario_id, producto_id):
    user = obtener_usuario(usuario_id)
    producto = obtener_producto(producto_id)
    ...
```

Dilema

Tienes un equipo de 3 programadores y necesitas lanzar un producto funcional en 4 meses. El inversionista espera ver métricas reales y los primeros clientes en la plataforma.

- ¿Qué tipo de arquitectura propondrías?
- ¿Cómo argumentarías tu decisión ante un CTO que insiste en usar Kubernetes, eventos y microservicios desde el día uno?

Cliente - Servidor

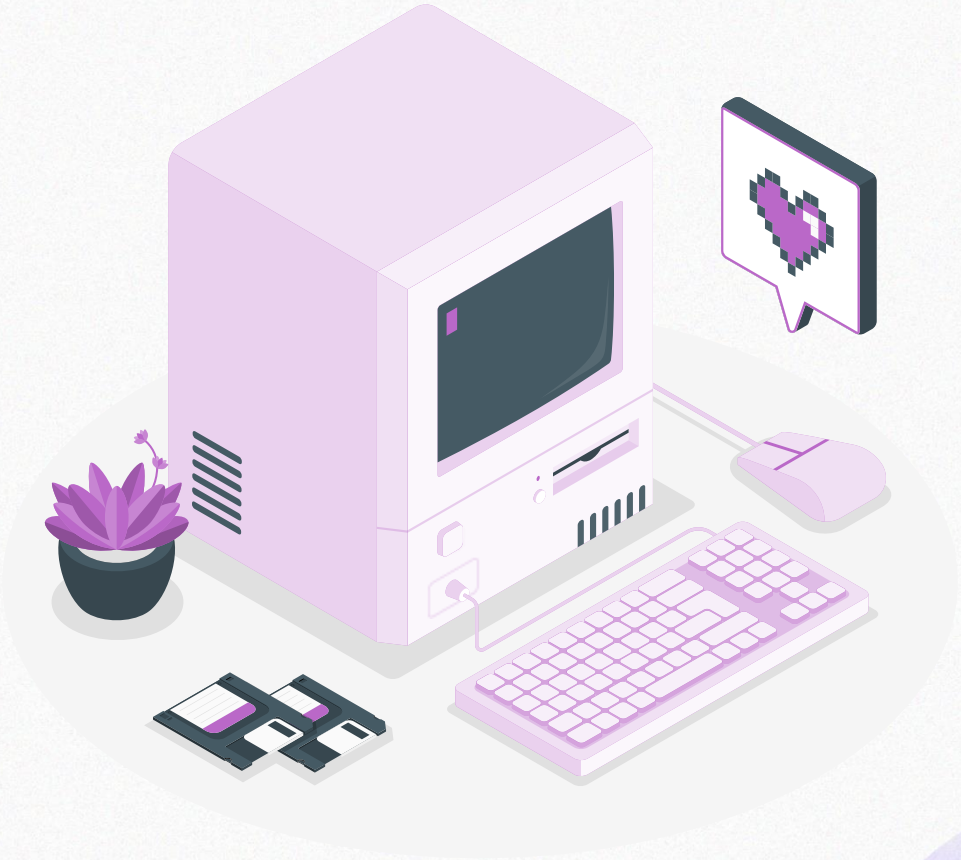
Cliente-servidor es un estilo arquitectónico en el que un conjunto de clientes interactúan con uno o más servidores mediante peticiones explícitas, accediendo a recursos o servicios controlados por estos últimos.

— Basado en Shaw & Garlan, *Software Architecture: Perspectives on an Emerging Discipline* (1996)

Cliente - Servidor

¿Cómo creen que se distribuía el software antes de que existiera internet?

¿Qué problema aparecía cuando una empresa tenía muchas sedes?



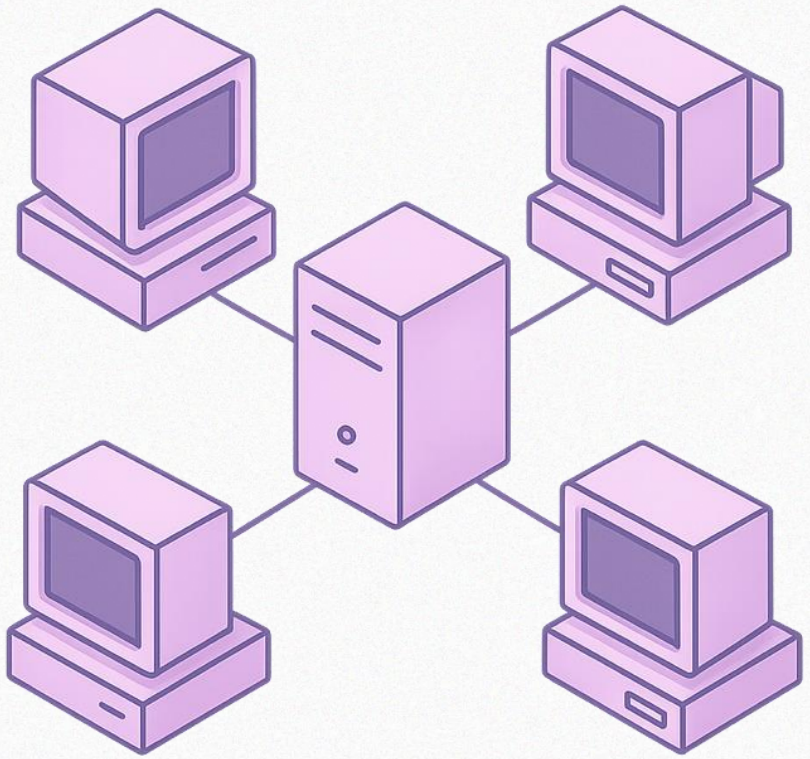
Cliente - Servidor

Antes de la masificación de la red, el software era ejecutado **localmente**, en la máquina del usuario.

Cada instalación era independiente. Si se dañaba, había que reinstalar. Si salía una nueva versión, había que ir PC por PC y actualizar.

Con la **difusión de las redes locales (LAN)** y luego **internet**, nace una pregunta clave:

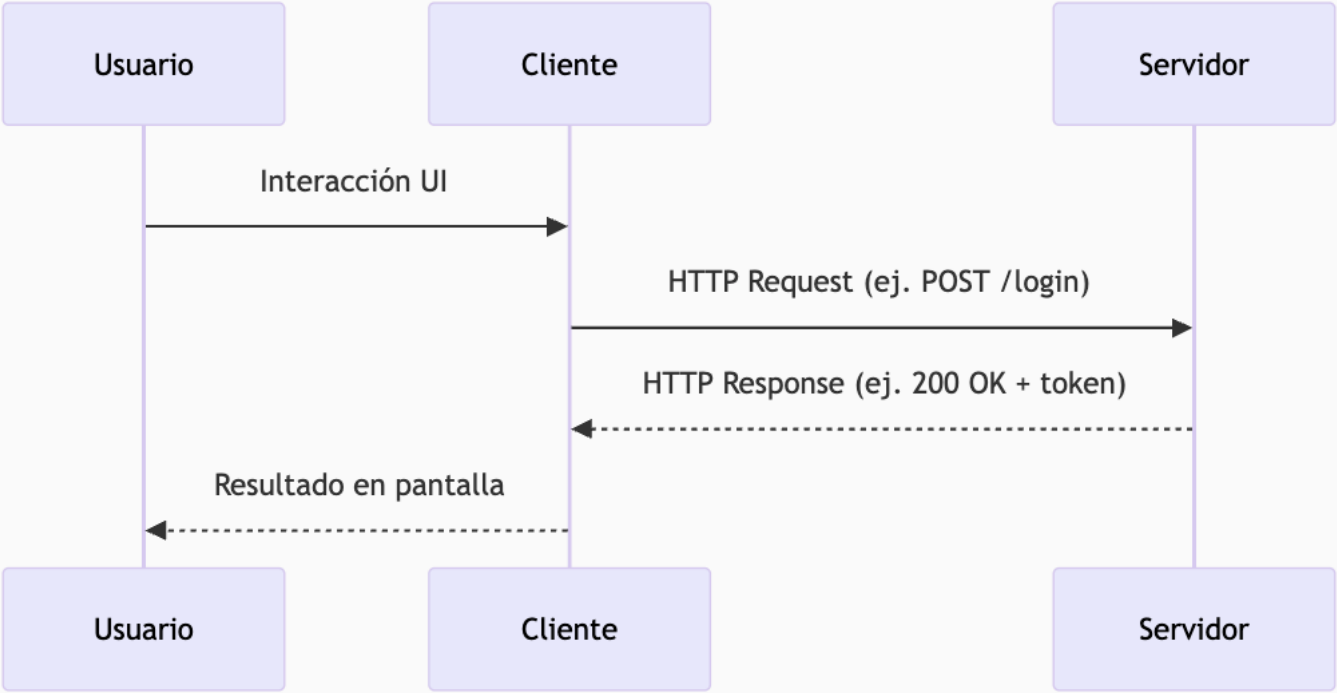
¿Y si centralizamos la lógica del sistema en un lugar, y dejamos que los usuarios accedan remotamente?



Cliente - Servidor

Cliente
- Inicia la interacción.
- Solicita recursos o acciones.

Servidor
- Espera peticiones.
- Procesa solicitudes y devuelve respuestas.



Ejemplo de cliente servidor

```python
# servidor.py
import socket

servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
servidor.bind(('0.0.0.0', 12345))
servidor.listen(1)
print("Servidor esperando conexión...")

conexion, direccion = servidor.accept()
print(f"Conectado desde: {direccion}")

mensaje = conexion.recv(1024).decode()
print(f"Cliente dice: {mensaje}")

conexion.send("Hola cliente, te recibo claro y fuerte.".encode())
conexion.close()
servidor.close()
```

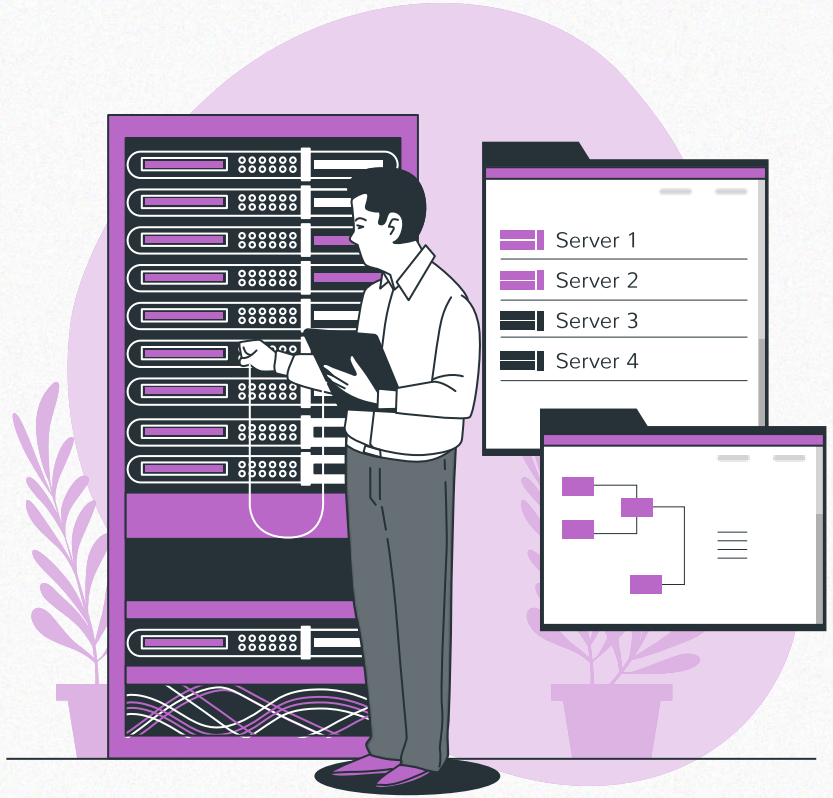
Ejemplo de cliente servidor

```python
# cliente.py
import socket

cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cliente.connect(('127.0.0.1', 12345))

cliente.send("Hola servidor, soy el cliente.".encode())
respuesta = cliente.recv(1024).decode()
print(f"Servidor dice: {respuesta}")

cliente.close()
```

Protocolos

Cuando los primeros servidores empezaron a comunicarse por red, ¿cómo sabían si los bits que recibían representaban una palabra, un número, una orden… o simplemente ruido?

O … ¿Cómo sabían los programadores como enviar una instrucción o mensaje en especifico?



Protocolos

ANTES:
[01001000 01101111 01101100 01100001]

¿"Hola"? ¿4 bytes de una estructura binaria?

PROBLEMA:
- ¿Dónde empieza y termina el mensaje?
- ¿Qué representa cada parte?
- ¿Y si alguien manda basura?

SOLUCIÓN:
Crear un PROTOCOLO: estructura fija, reglas claras, orden.

Protocolos

ANTES:
[01001000 01101111 01101100 01100001]

¿"Hola"? ¿4 bytes de una estructura binaria?

PROBLEMA:
- ¿Dónde empieza y termina el mensaje?
- ¿Qué representa cada parte?
- ¿Y si alguien manda basura?
- No existían formatos universales como JSON o XML.
- Cada sistema inventaba su propio protocolo propietario.

SOLUCIÓN:
Crear un PROTOCOLO: estructura fija, reglas claras, orden.

Protocolos

Un **protocolo** es un conjunto de **reglas formales y estructuras** que definen cómo dos sistemas se comunican correctamente entre sí.

¿Qué pasa exactamente cuando escribes una URL como **http://www.example.com** y presionas Enter?

Protocolos

HTTP (HyperText Transfer Protocol) es el protocolo que permite que un cliente (como un navegador) **pida recursos** (como páginas, imágenes, o datos) a un servidor.

Cliente

GET /index.html HTTP/1.1
Host: www.example.com

- **GET** es el método (acción).
- **/index.html** es el recurso.
- **HTTP/1.1** es la versión del protocolo.
- **Host** indica a qué servidor estamos hablando.

Servidor

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

```
<html>
  <body>Hola mundo</body>
</html>
```

Protocolos

HTTP (HyperText Transfer Protocol) es el protocolo que permite que un cliente (como un navegador) **pida recursos** (como páginas, imágenes, o datos) a un servidor.

Cliente

GET /index.html HTTP/1.1
Host: www.example.com

- **GET** es el método (acción).
- **/index.html** es el recurso.
- **HTTP/1.1** es la versión del protocolo.
- **Host** indica a qué servidor estamos hablando.

Servidor

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

```
<html>
  <body>Hola mundo</body>
</html>
```

Cliente - Servidor

Variantes Arquitectonicas

Thin client

- Toda la lógica en el servidor.
- Cliente solo presenta datos (ej. navegador simple).

Fat client

- Parte de la lógica vive en el cliente.
- Requiere mayor sincronización con el servidor.

Multi-servidor

- Diversos servicios especializados: autenticación, recursos, datos.

Cliente - Servidor

¿Sigue siendo relevante?

Sí, pero ha **evolucionado** en múltiples direcciones:

- Backend RESTful que sirve múltiples frontends (SPA, apps móviles).
- Arquitecturas de 3 capas o n-capas (web → API Gateway → backend).
- Combinado con eventos o colas de mensajes para desacoplar componentes.

Componentes del protocolo HTTP

Metodo: Indica la acción: GET, POST, PUT, DELETE, etc.

Ruta/URI: Especifica el recurso solicitado.

Cabeceras: Añaden metadatos (tipo de contenido, codificación, autenticación…).

Cuerpo: Contiene datos que se envían (opcional).

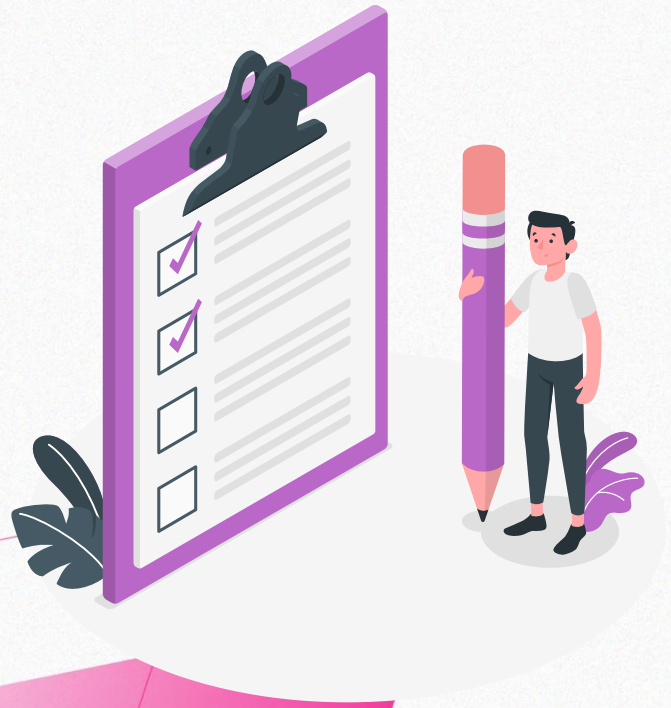
Código de estado: El servidor responde con un código que indica el resultado (200 OK, 404 Not Found, 500 Internal Server Error, etc).

Ejemplo Servidor HTTP

```
const express = require('express');
const app = express();

// Middleware para interpretar JSON en el cuerpo
app.use(express.json());

// Ruta tipo GET
app.get('/', (req, res) => {
  res.send('Hola desde el servidor Express');
});

// Ruta tipo POST
app.post('/saludo', (req, res) => {
  const nombre = req.body.nombre;
  res.send(`Hola, ${nombre}`);
});

// Iniciar servidor
app.listen(3000, () => {
  console.log('Servidor escuchando en http://localhost:3000');
});
```

Rest / Restful

REST no es un protocolo, ni una tecnología, sino un **conjunto de principios** propuestos por Roy Fielding en su tesis doctoral (2000).



Principios de REST

1. **Cliente-servidor:** Separación de responsabilidades: el cliente consume, el servidor provee.
2. **Sin estado (Stateless):** Cada solicitud HTTP contiene toda la información necesaria. El servidor **no guarda contexto** entre llamadas.
3. **Caché:** Las respuestas deben indicar si pueden ser almacenadas o no.
4. **Interfaz uniforme** *(el más importante):* Se accede a los recursos a través de URLs y **métodos estándar de HTTP**.
5. **Sistema en capas:** Puede haber proxies, balanceadores, etc. ocultos al cliente.
6. **Código bajo demanda (opcional):** Posibilidad de enviar scripts ejecutables (rara vez se usa).

REST trabaja con recursos y colecciones

¿Qué es un recurso?

Un **recurso** es cualquier entidad **identificable** dentro del sistema.

GET /usuarios/42 Esto representa **el usuario con ID 42**.

¿Qué es una colección?

Una **colección** es un conjunto de recursos del mismo tipo.

GET /usuarios Todos los usuarios
POST /usuarios Crear un nuevo usuario

REST trabaja con recursos y colecciones

¿Qué es un recurso?

Un **recurso** es cualquier entidad **identificable** dentro del sistema.

¿Qué es una colección?

Una **colección** es un conjunto de recursos del mismo tipo.

Acción	Verbo HTTP	URL	Resultado
Listar usuarios	GET	/usuarios	Devuelve la colección
Ver un usuario	GET	/usuarios/42	Devuelve un recurso
Crear usuario	POST	/usuarios	Añade un nuevo recurso
Actualizar completo	PUT	/usuarios/42	Reemplaza todo
Actualizar parcial	PATCH	/usuarios/42	Modifica campos
Eliminar usuario	DELETE	/usuarios/42	Elimina recurso

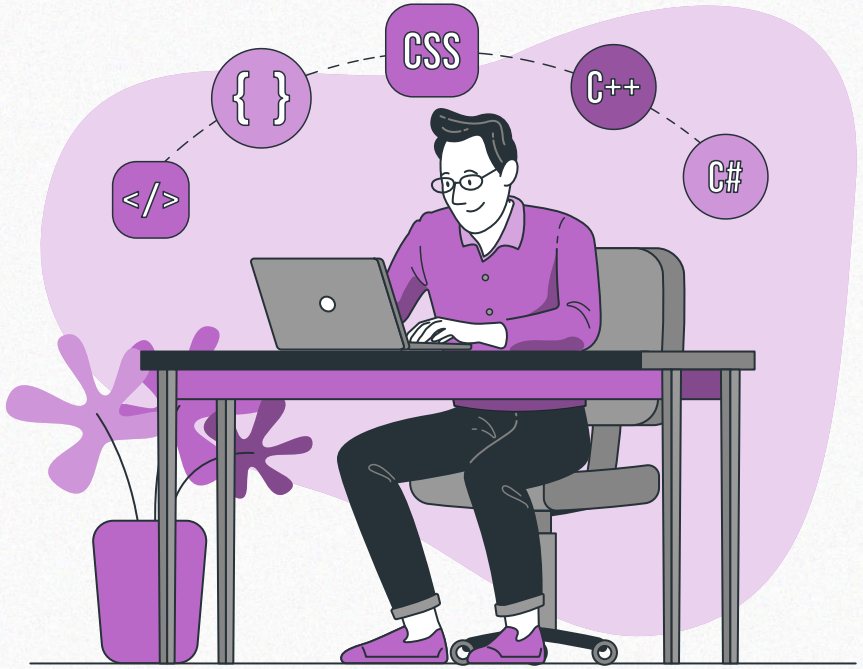
¿Por qué recursos y no funciones?

En REST **no se invocan funciones** tipo getUser().

En su lugar, **se accede a representaciones de recursos** usando las rutas.

Esto hace que las APIs sean más:

- **Predecibles**
- **Cacheables**
- **Compatibles con la Web**



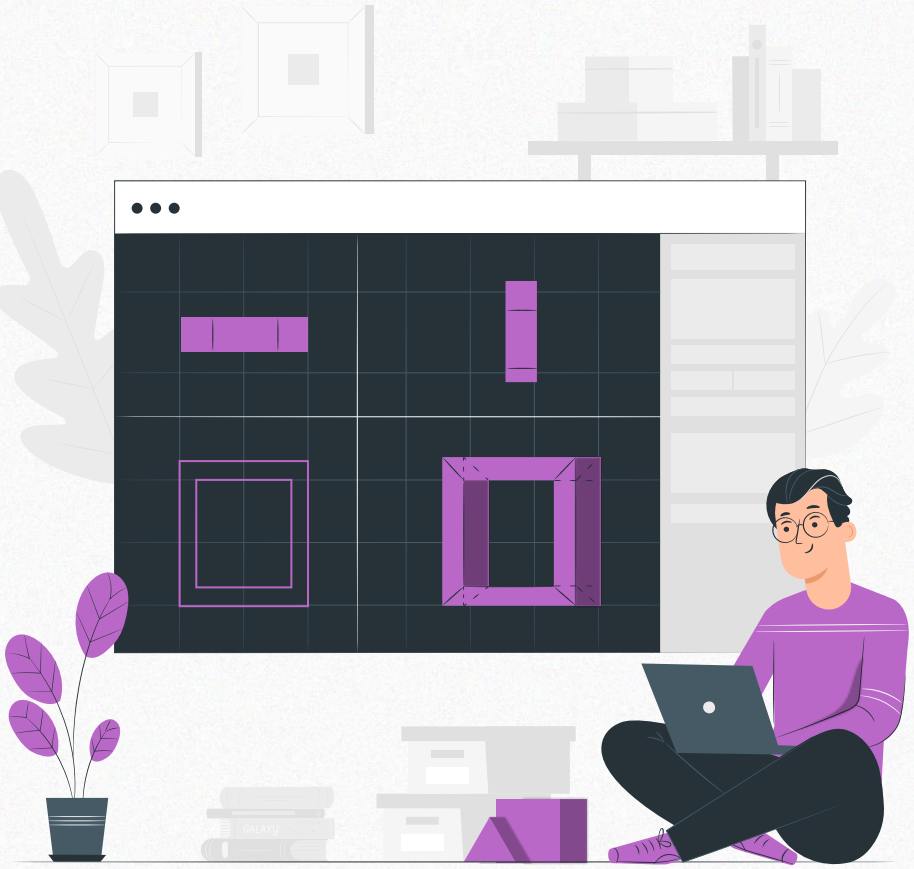
Representaciones

REST se llama así porque devuelve una **“Representación”** del recurso.

Puede ser en formato:

- application/json
- text/html
- application/xml

```
{
  "id": 42,
  "nombre": "Joel",
  "rol": "Administrador"
}
```



Restful

REST es como una dieta saludable; RESTful es seguirla bien: con disciplina y sin trampas.

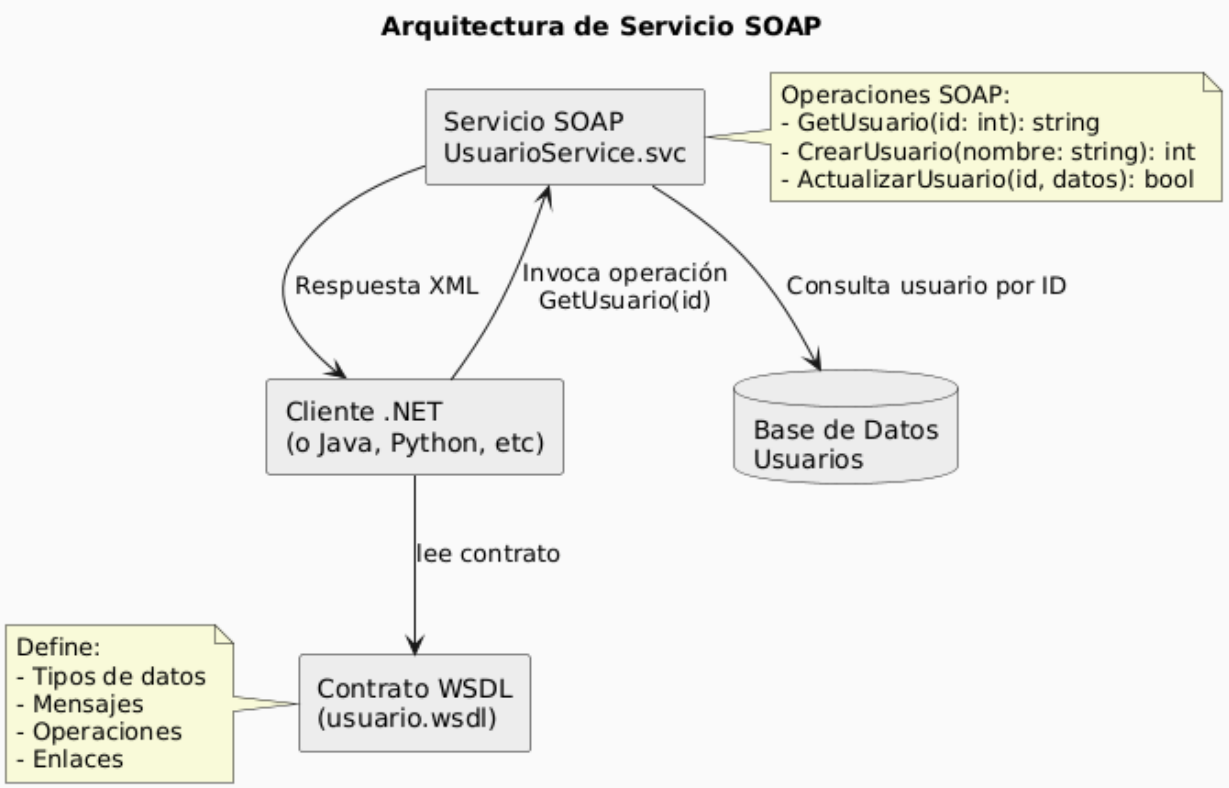
¿REST será *la única* forma correcta de permitir la comunicación cliente-servidor en el mundo moderno?

¿Hay casos donde REST no alcanza… y necesitamos algo más o diferente?

SOAP

Simple Object Access Protocol es un **protocolo de mensajería** basado en **XML**, diseñado para permitir la comunicación entre sistemas distribuidos a través de redes, especialmente en contextos **empresariales y heterogéneos**.

- Formato definido (XML)
- Estructura fija (envelope, header, body)
- Contratos (WSDL)
- Reglas de validación



WSDL

WSDL (Web Services Description Language) es un **documento XML** que describe **de manera formal y estandarizada** cómo interactuar con un servicio web SOAP.

- Establece un **acuerdo estricto** entre el **cliente y el servidor**.
- Si ambos respetan ese contrato, pueden comunicarse sin ambigüedad, incluso si están escritos en distintos lenguajes (Java, .NET, Python).
- Es como un “manual técnico” automático, generado desde el servicio y **consumido por herramientas** que crean clientes automáticamente.

```xml
<!-- Tipos de datos -->
<types>
  <xsd:schema targetNamespace="http://ejemplo.com/usuario">
    <xsd:element name="GetUsuarioRequest">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="id" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="GetUsuarioResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="nombre" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</types>

<!-- Mensajes -->
<message name="GetUsuarioInput">
  <part name="parameters" element="tns:GetUsuarioRequest"/>
</message>
<message name="GetUsuarioOutput">
  <part name="parameters" element="tns:GetUsuarioResponse"/>
</message>

<!-- PortType: define las operaciones -->
<portType name="UsuarioPortType">
  <operation name="GetUsuario">
    <input message="tns:GetUsuarioInput"/>
    <output message="tns:GetUsuarioOutput"/>
  </operation>
</portType>

<!-- Binding: cómo se hace la llamada -->
<binding name="UsuarioBinding" type="tns:UsuarioPortType">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetUsuario">
    <soap:operation soapAction="http://ejemplo.com/usuario/GetUsuario"/>
    <input>
```

¿Dónde se usa SOAP?

Servicios de **banca, seguros, gobierno**, donde se necesita:

- Precisión formal
- Firmas digitales
- Esquemas bien definidos
- Compatibilidad entre lenguajes (Java ↔ .NET)

¿Cómo se define un servicio SOAP?

Mediante un archivo **WSDL** que describe:

- Operaciones disponibles
- Tipos de datos
- Estructura de entrada/salida
- Puertos y ubicaciones

```
<definitions name="UsuarioService" ...>
  <message name="getUsuarioRequest">
    <part name="id" type="xsd:int"/>
  </message>
  <message name="getUsuarioResponse">
    <part name="nombre" type="xsd:string"/>
  </message>
  ...
</definitions>
```

REST:

Dame el recurso *usuario 42*.
GET /usuarios/42

SOAP:

Ejecuta la operación *getUsuario(id=42)*

(dentro del SOAP Body, no en la URL)

```
var cliente = new UsuarioServiceRef.UsuarioServiceClient();
var resultado = cliente.GetUsuario(42);
Console.WriteLine("Respuesta del servidor: " + resultado);
```

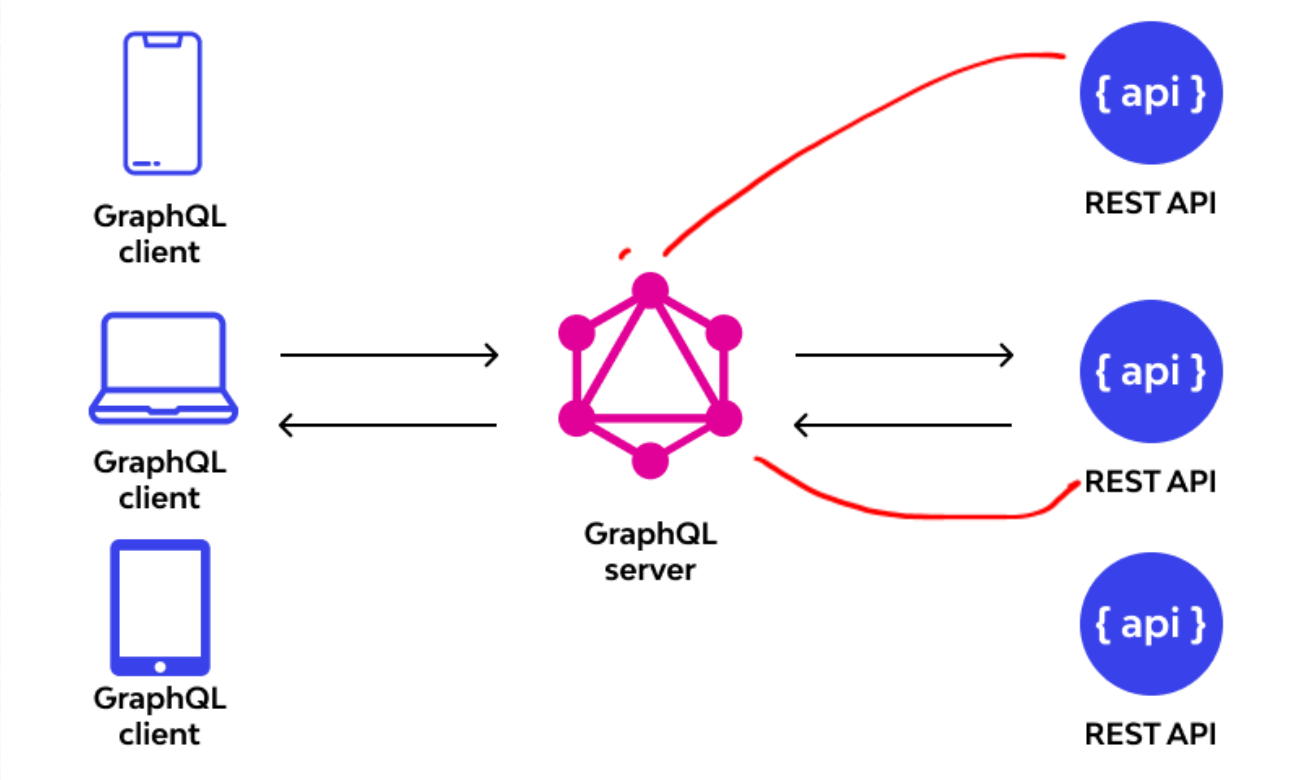
BFF

¿Qué pasa si una app móvil solo necesita el nombre del usuario… pero nuestra API REST devuelve 30 campos?

GraphQL

Es un lenguaje de consulta para APIs y un entorno de ejecución del lado del servidor, que permite a los clientes **definir exactamente qué datos necesitan**.

```
query {
  usuario(id: 42) {
    nombre
    email
    direccion {
      ciudad
    }
  }
}
```



GraphQL

Es un lenguaje de consulta para APIs y un entorno de ejecución del lado del servidor, que permite a los clientes **definir exactamente qué datos necesitan**.

Componente	Función
Schema	Define los tipos, relaciones y operaciones disponibles
Query	Consultas de lectura
Mutation	Operaciones que modifican datos
Resolver	Función que ejecuta la lógica detrás de cada campo

```
type Usuario {
  id: ID
  nombre: String
  email: String
  direccion: Direccion
}

type Direccion {
  ciudad: String
  pais: String
}

type Query {
  usuario(id: ID!): Usuario
}

type Mutation {
  crearUsuario(nombre: String, email: String): Usuario
}
```

GraphQL sobre APIs

El servidor GraphQL consulta una API REST existente

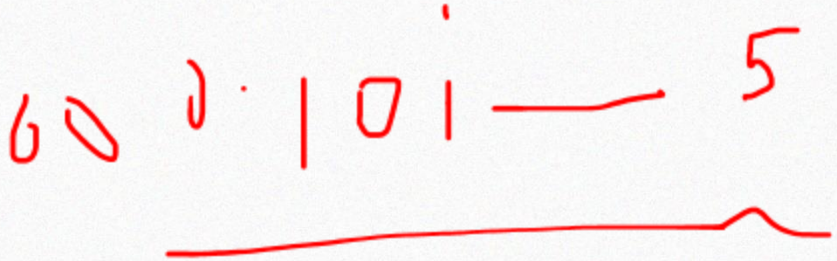
```
// servidor.js
const { ApolloServer, gql } = require('apollo-server');
const fetch = require('node-fetch');

const typeDefs = gql`
  type Usuario {
    id: ID
    nombre: String
  }

  type Query {
    usuario(id: ID!): Usuario
  }
`;

const resolvers = {
  Query: {
    usuario: async (_, { id }) => {
      const res = await fetch(`https://api.ejemplo.com/usuarios/${id}`);
      const data = await res.json();
      return {
        id: data.id,
        nombre: data.nombre
      };
    }
  }
};

const server = new ApolloServer({ typeDefs, resolvers });
server.listen().then(({ url }) => {
  console.log(`Servidor GraphQL en ${url}`);
});
```

GraphQL sobre Modelos o DB

El servidor GraphQL accede directamente a los modelos o DB

```javascript
// usuarioModel.js
const usuariosDB = [
  { id: '42', nombre: 'Joel' },
  { id: '1', nombre: 'Andrea' }
];

function buscarUsuario(id) {
  return usuariosDB.find((u) => u.id === id);
}

module.exports = { buscarUsuario };
```

```javascript
// servidor.js
const { ApolloServer, gql } = require('apollo-server');
const { buscarUsuario } = require('./usuarioModel');

const typeDefs = gql`
  type Usuario {
    id: ID
    nombre: String
  }

  type Query {
    usuario(id: ID!): Usuario
  }
`;

const resolvers = {
  Query: {
    usuario: (_, { id }) => buscarUsuario(id)
  }
};
```

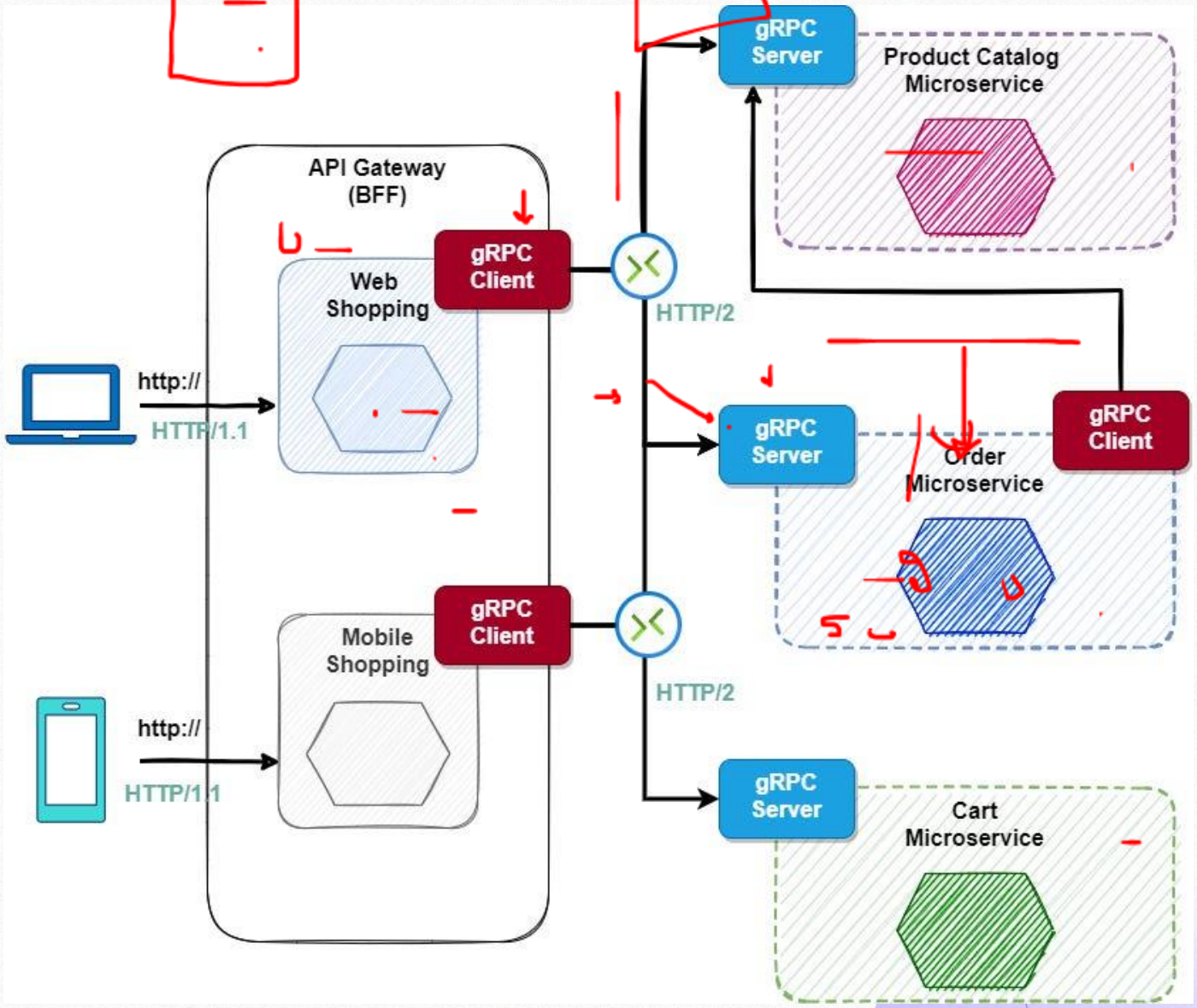


¿Qué pasaría si una empresa con cientos de microservicios necesitara que todos se comuniquen millones de veces al día… y rápido?

¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

- HTTP/2 (soporte nativo para streams, multiplexing)
- **Protocol Buffers (Protobuf)** como formato de serialización
- Código fuertemente tipado generado automáticamente



¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder

```
// usuario.proto
syntax = "proto3";

service UsuarioService {
  rpc GetUsuario (UsuarioRequest) returns (UsuarioResponse);
}

message UsuarioRequest {
  int32 id = 1;
}

message UsuarioResponse {
  string nombre = 1;
}
```

¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder

```js
// servidor.js
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

const paqueteDef = protoLoader.loadSync('usuario.proto');
const grpcObj = grpc.loadPackageDefinition(paqueteDef);
const servidor = new grpc.Server();

servidor.addService(grpcObj.UsuarioService.service, {
  GetUsuario: (call, callback) => {
    const id = call.request.id;
    const nombre = id === 42 ? "Joel" : "Desconocido";
    callback(null, { nombre });
  }
});

servidor.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createIns
  servidor.start();
});
```

¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

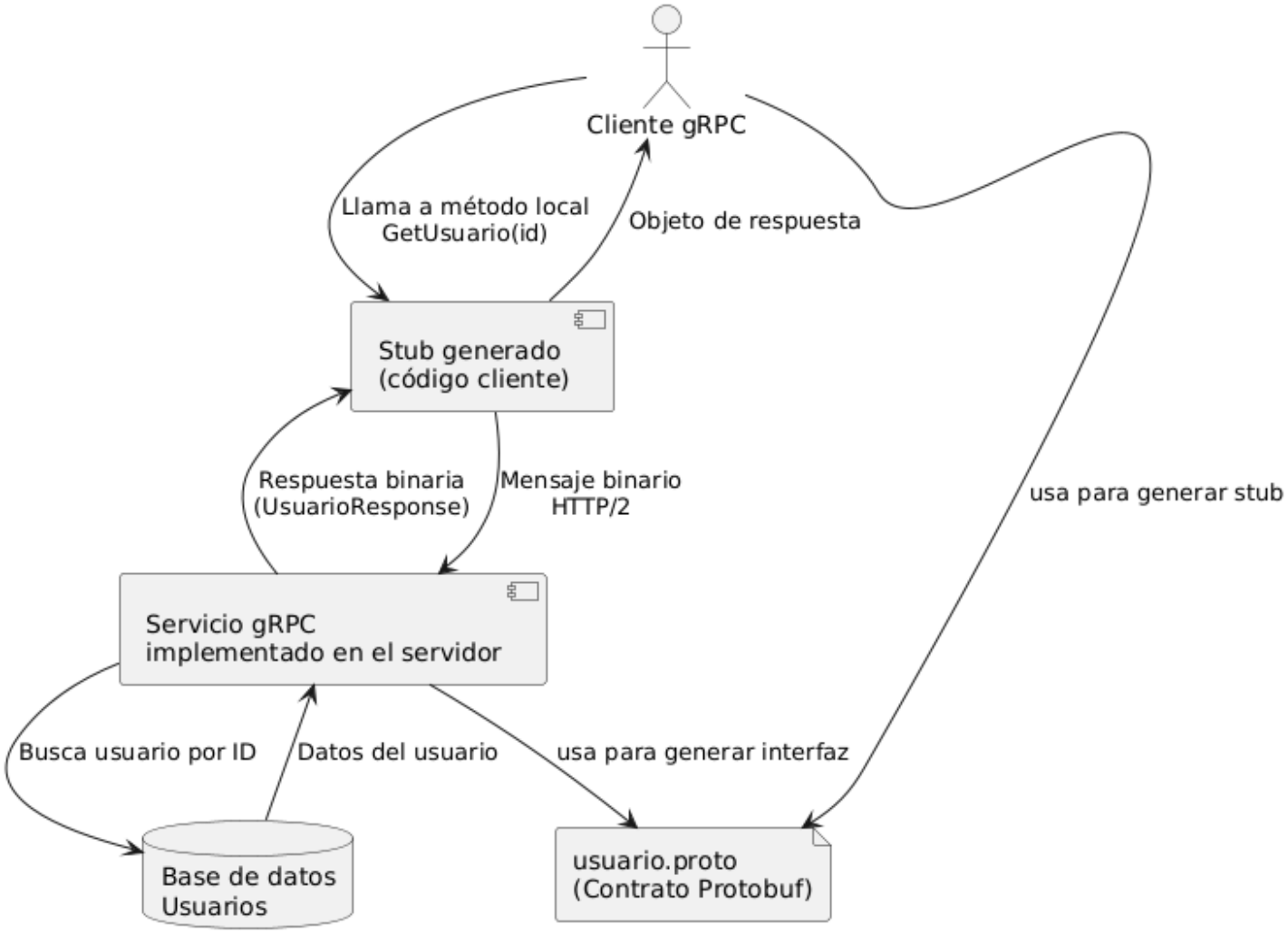
Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder

```
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

const paqueteDef = protoLoader.loadSync('usuario.proto');
const grpcObj = grpc.loadPackageDefinition(paqueteDef);

const cliente = new grpcObj.UsuarioService('localhost:50051',

cliente.GetUsuario({ id: 42 }, (err, respuesta) => {
  console.log("Nombre:", respuesta.nombre);
});
```

¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder



Por qué seguimos hablando de arquitectura en capas?

¿Qué imaginan cuando piensan en “arquitectura en capas”?

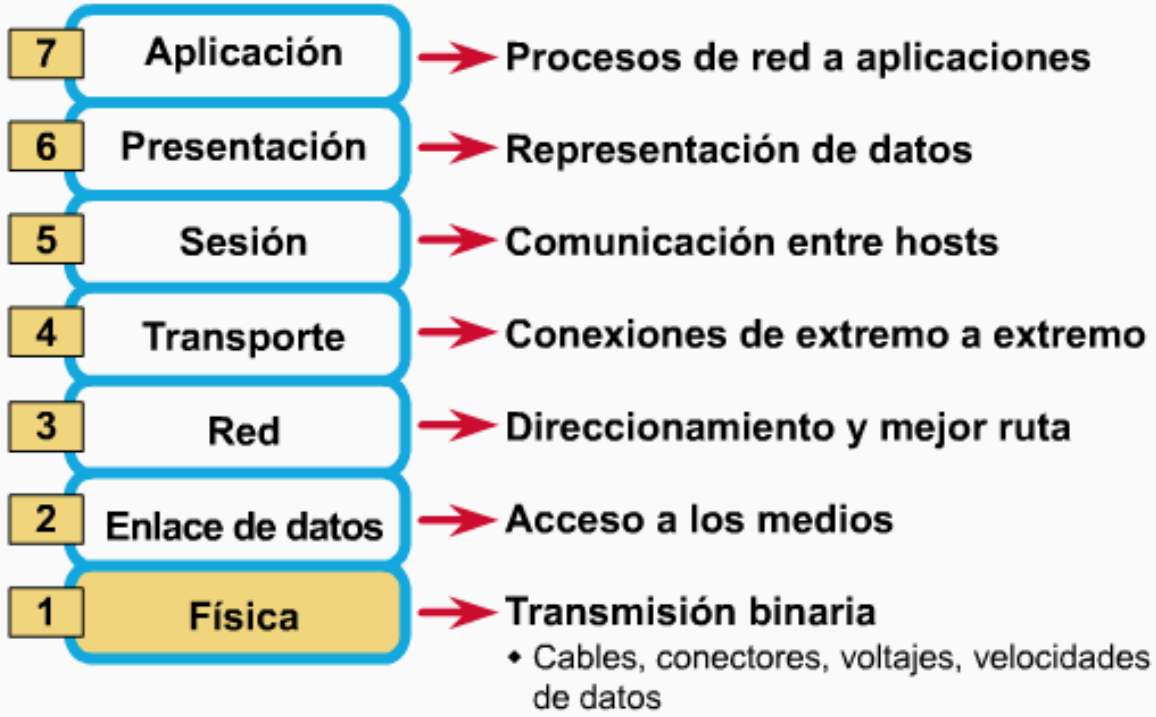
¿Qué capa creen que es la más importante?

¿Qué problemas creen que trae escalar una aplicación basada en capas?

La separación en capas permite aislar responsabilidades, pero también puede encerrar la lógica de negocio entre paredes difíciles de escalar.

Bass, Clements & Kazman
(Software Architecture in Practice)

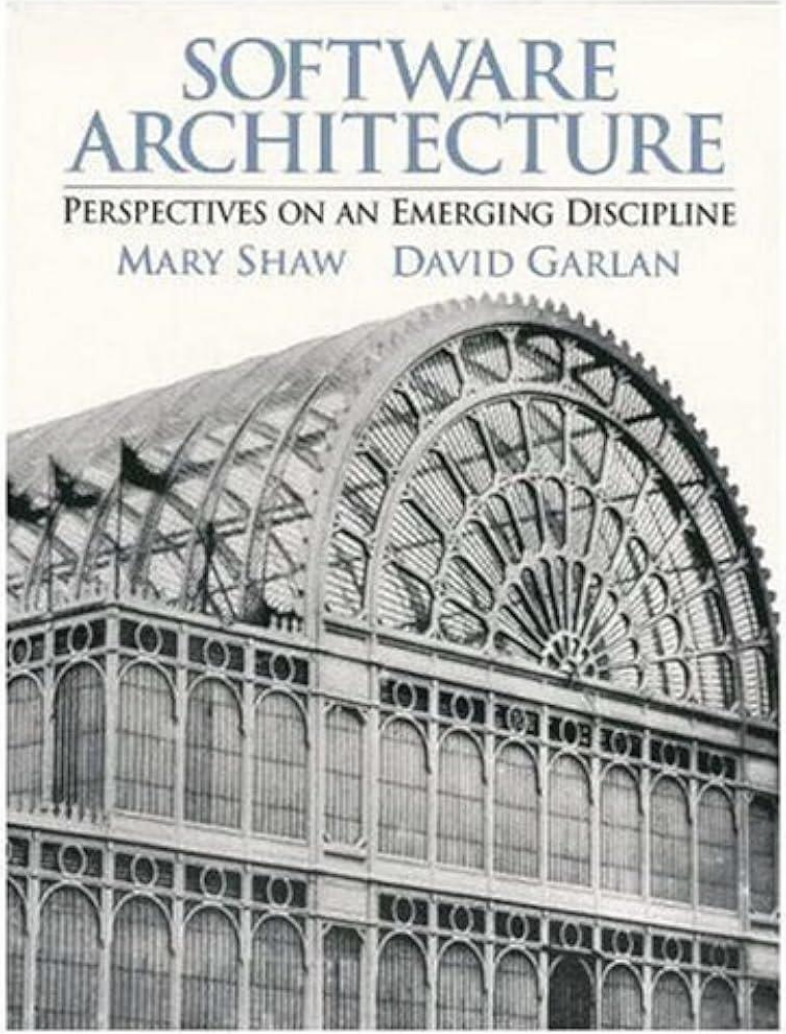
¿Cuántos de ustedes tienen carpetas llamadas controllers, services y repositories… ?



Modelo OSI

Si funciona para las redes, ¿por qué no para el software?

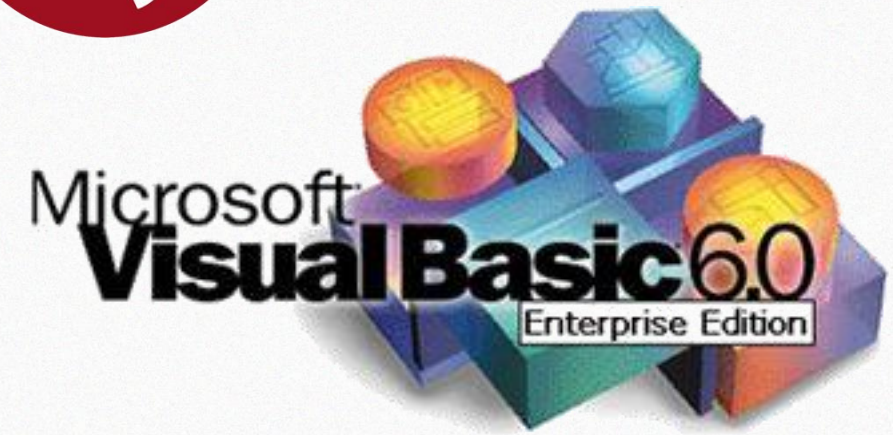
Aunque la separación por capas ya era practicada implícitamente desde los 70s, fue **Mary Shaw y David Garlan en 1995** quienes la identificaron formalmente como un *estilo arquitectónico* en la teoría de arquitectura de software. A partir de entonces, textos como *Software Architecture in Practice* la consolidaron como una práctica habitual.



Presentation (UI)

Procesamiento de lógica (Aplicación)

Acceso a datos (Infraestructura)



¿Qué tenían en común esas aplicaciones que hacía que las capas fueran útiles?

“Layered architecture emerged as a response to increasing complexity and team size. It allowed teams to specialize, and systems to become maintainable.”

— Bass, Clements & Kazman –
Software Architecture in Practice

Lógica en la base de datos

El reinado de los Stored Procedures (80s-90s)

- La lógica de negocio no estaba en el código. Estaba en el SQL.
- Roles especializados y diferenciados (DBA, Ingeniero de Aplicaciones)

The business logic in stored procedures became a bottleneck for change and reuse.

— Martin Fowler, *Patterns of Enterprise Application Architecture*

Presentation (Visual Programming)

Business Logic (PL/SQL)

Arquitectura de 3 capas (90s-2000s)

El modelo que todos copiamos

¿Por qué se separó la capa de negocio de la capa de datos?

Layered architecture helped teams to focus on a single concern at a time. But it also led to excessive boilerplate code

— Mark Richards, *Fundamentals of Software Architecture*

Presentation (HTML, Swing, JSP)

Application / Business Logic

Datos (DAO, JDBC, Hibernate)

Arquitectura de 3 capas (90s-2000s)

El modelo que todos copiamos

¿Por qué se separó la capa de negocio de la capa de datos?

Layered architecture helped teams to focus on a single concern at a time. But it also led to excessive boilerplate code

— Mark Richards, *Fundamentals of Software Architecture*

Presentation	**UserController**
Business Logic	**User Service**
Data	**User Repository**

La inflación de capas (2000s–2010s)

Arquitectura “n capas”

- Equipos grandes, necesidad de delegar tareas y responsabilidades específicas.
- Aparecen roles como: desarrollador de entidades, de servicios, de mapeadores, etc.
- Herramientas como Spring, Hibernate, WCF reforzaban esta estructura.

Layered systems often reflect the organizational structure rather than the problem domain

UserController

UserService

UserUseCase

UserApplicationValidator

UserMapper

UserDTO

UserVO

UserEntity

Enfoques Modernos

No todo lo que tiene capas es arquitectura. A veces, es solo burocracia en forma de código.

Después de años construyendo arquitecturas rígidas, con capas que no hacían nada, la industria empezó a preguntarse: ¿realmente esto agrega valor?

- Énfasis en entregas frecuentes
- Feedback temprano
- Menos tolerancia a estructuras pesadas
- Una única capa de presentación ya no era suficiente

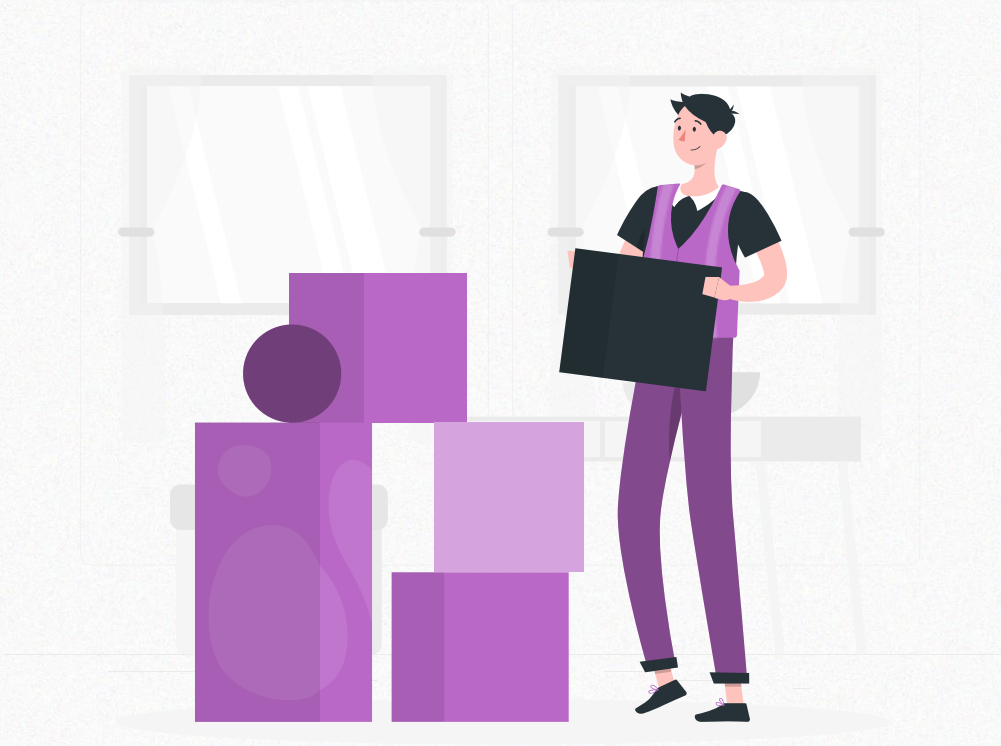
Enfoques Modernos

Menos capas, más responsabilidades claras

- No separar por separar.
- Solo extraer una capa si encapsula una lógica que **cambia por separado** o **vale la pena abstraer**.

Clean Architecture / DDD

- El dominio no depende de frameworks, bases de datos ni controladores.
- El código “core” es **independiente** y **testeable sin infraestructura**.

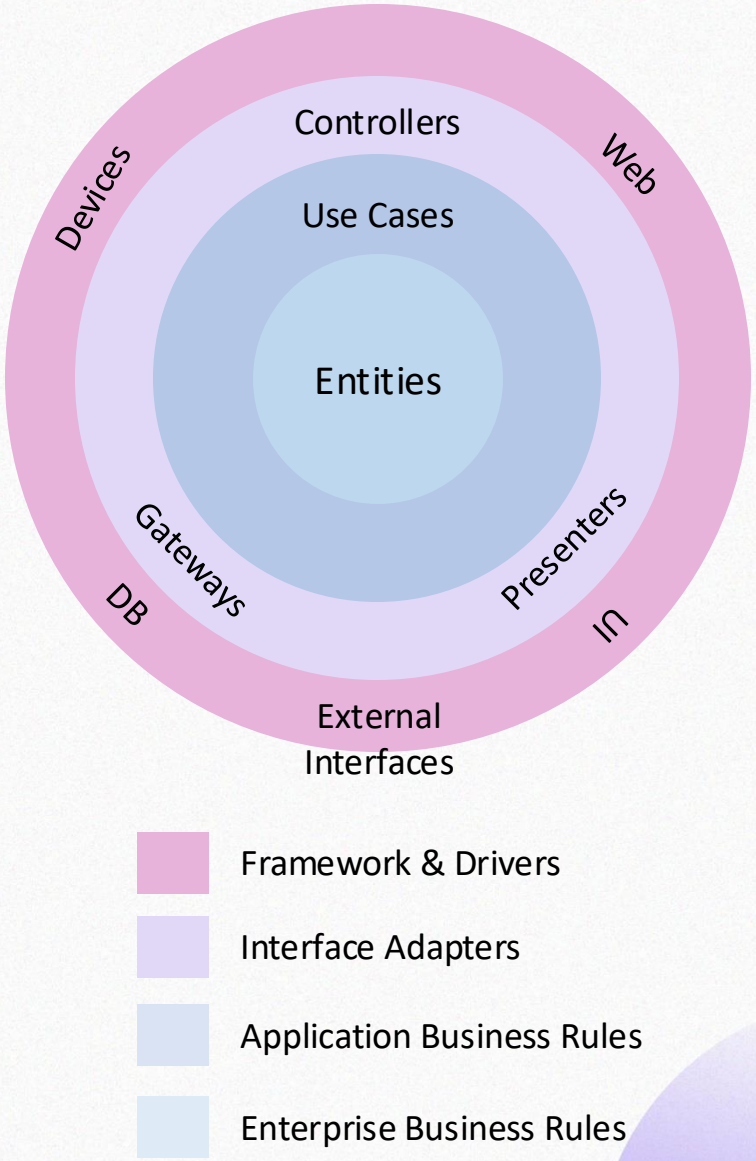


Clean Architecture

Describe a una arquitectura centrada en el **dominio de negocio**, donde **las dependencias siempre apuntan hacia el núcleo del sistema**, sin importar qué tecnología se use alrededor.

¿Por qué cuando cambias de base de datos, tu lógica de negocio se rompe?

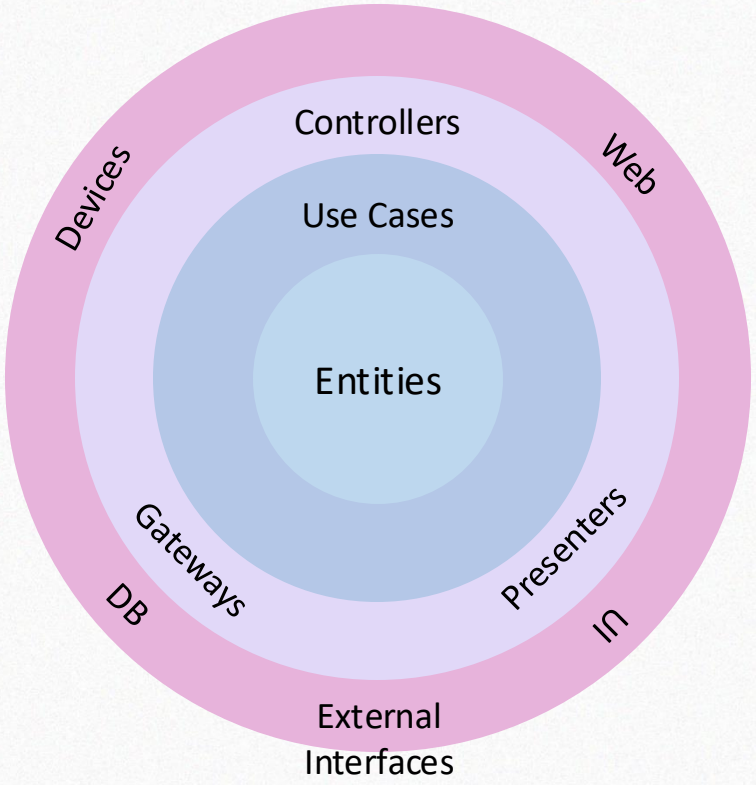
Porque la lógica está **al servicio del framework**, en vez de que el framework esté al servicio del negocio.



Clean Architecture

"Un estudiante se inscribe en un curso si hay cupo disponible."

Entity (Dominio)	Lógica interna de Curso
Use Case (Aplicación)	InscribirEstudianteUseCase
Input Adapter	InscripcionController
Output Port	CursoRepository (interfaz)
Output Adapter	MongoCursoRepository



Framework & Drivers

Interface Adapters

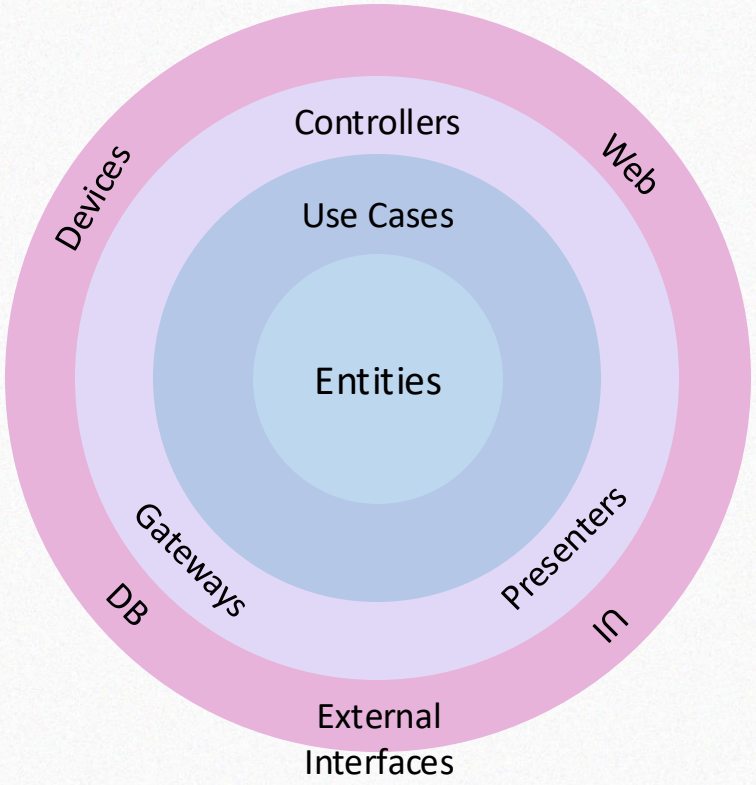
Application Business Rules

Enterprise Business Rules

Clean Architecture

Enterprise Business Rule – Dominio

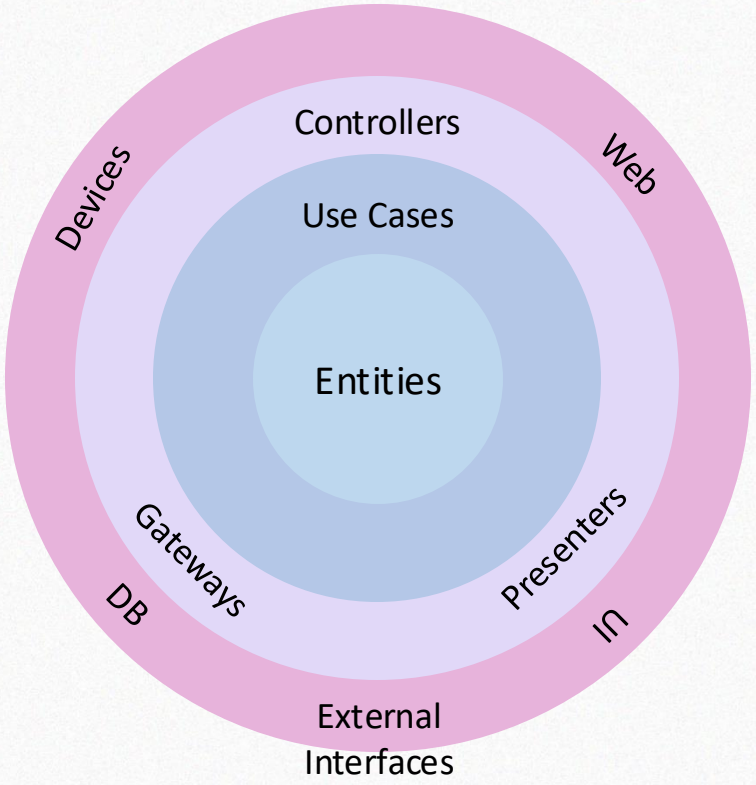
```java
public class Curso {
    private final String id;
    private final int cupoMaximo;
    private final List<String> estudiantes;

    public Curso(String id, int cupoMaximo) {
        this.id = id;
        this.cupoMaximo = cupoMaximo;
        this.estudiantes = new ArrayList<>();
    }

    public boolean puedeInscribir() {
        return estudiantes.size() < cupoMaximo;
    }

    public void inscribir(String estudianteId) {
        if (!puedeInscribir()) {
            throw new RuntimeException("Cupo lleno");
        }
        estudiantes.add(estudianteId);
    }

    public List<String> getEstudiantes() {
        return estudiantes;
    }
}
```



Clean Architecture

Application Business Rule – Caso de Uso

```java
public class InscribirEstudianteUseCase {
    private final CursoRepository cursoRepository;

    public InscribirEstudianteUseCase(CursoRepository cursoRepository) {
        this.cursoRepository = cursoRepository;
    }

    public void ejecutar(String cursoId, String estudianteId) {
        Curso curso = cursoRepository.buscarPorId(cursoId);
        curso.inscribir(estudianteId);
        cursoRepository.guardar(curso);
    }
}
```



Framework & Drivers

Interface Adapters

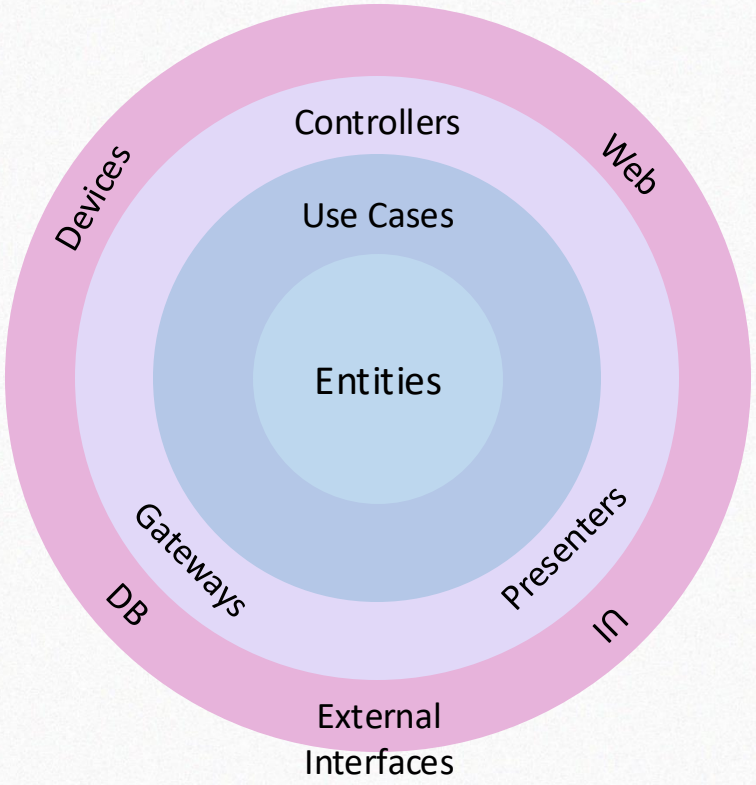
Application Business Rules

Enterprise Business Rules

Clean Architecture

Interface Output Port – Repositorio

```
public interface CursoRepository {
    Curso buscarPorId(String id);
    void guardar(Curso curso);
}
```



Framework & Drivers

Interface Adapters

Application Business Rules

Enterprise Business Rules

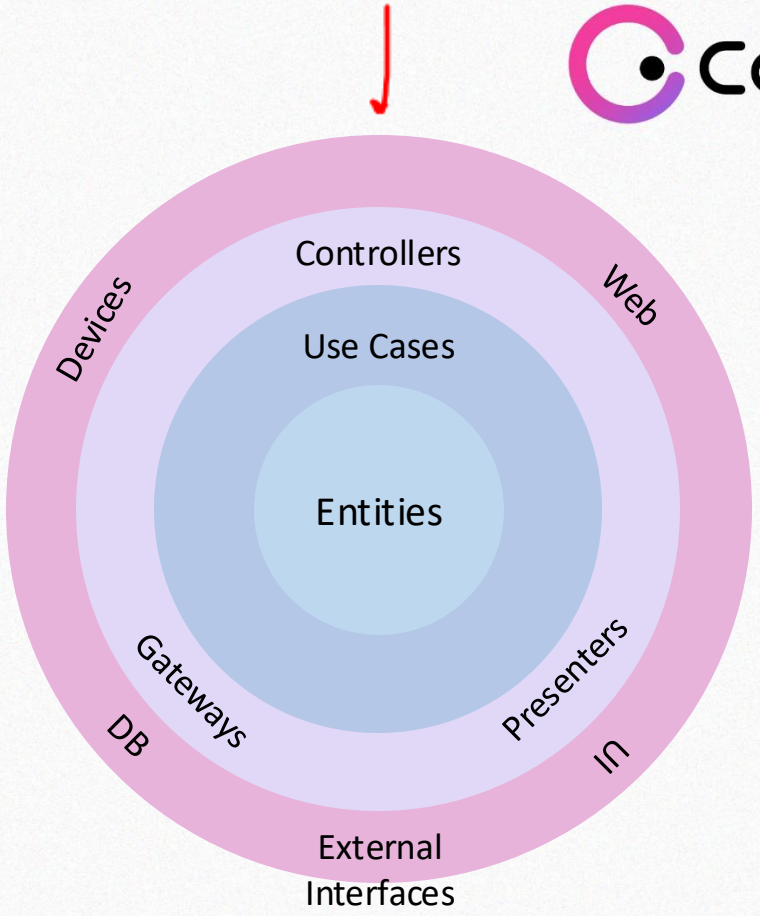
Clean Architecture

Output Adapter – Implementación Mongo

```java
public class MongoCursoRepository implements CursoRepository {
    @Override
    public Curso buscarPorId(String id) {
        // Simulación Mongo
        return new Curso(id, 2); // curso con cupo máximo 2
    }

    @Override
    public void guardar(Curso curso) {
        // Simular persistencia
        System.out.println("Guardando curso con " + curso.getEstu
    }
}
```

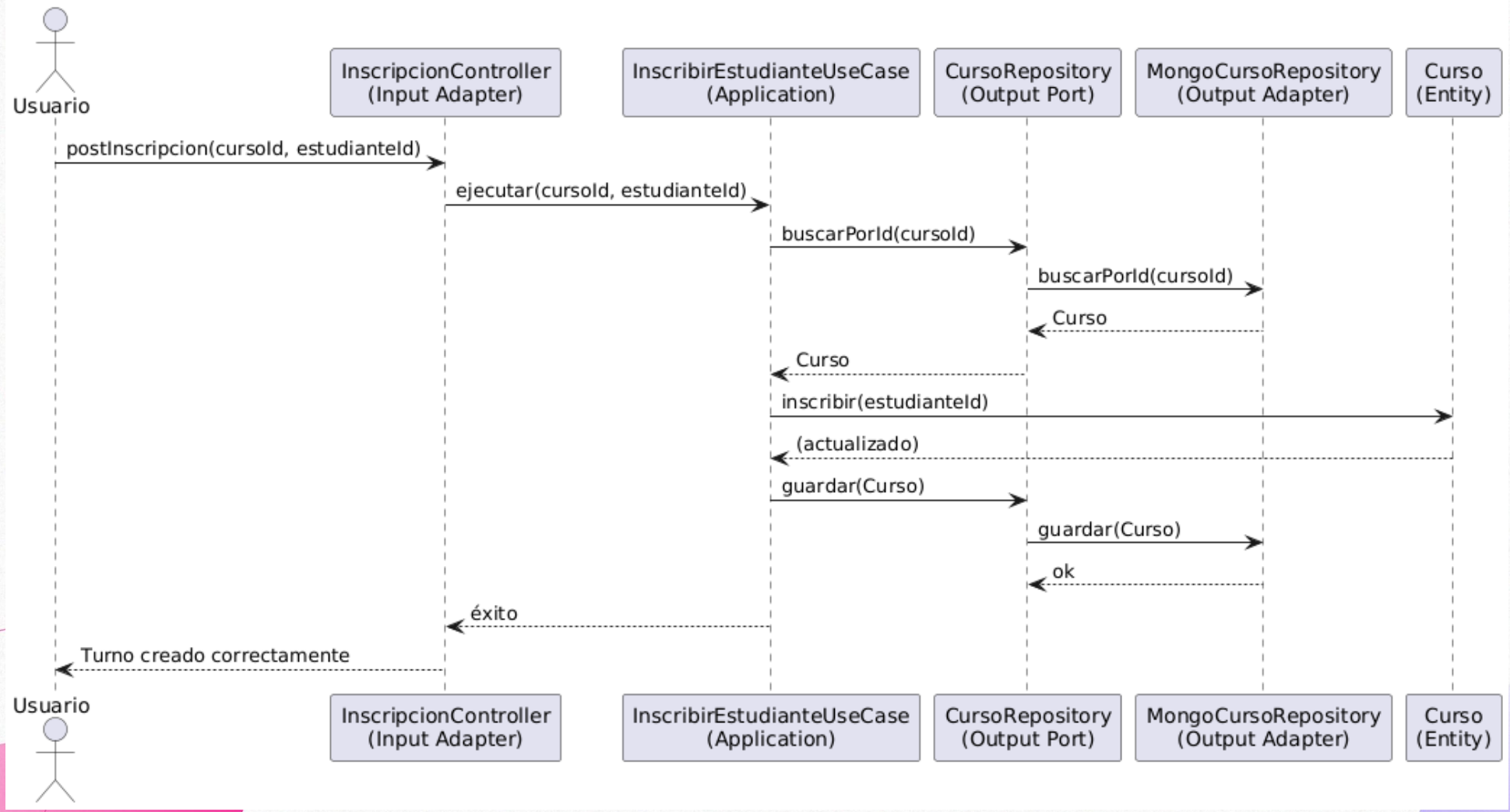
Input Adapter – Controlador REST (simplificado)

```java
public class InscripcionController {
    private final InscribirEstudianteUseCase useCase;

    public InscripcionController(InscribirEstudianteUseCase useCase) {
        this.useCase = useCase;
    }

    public void postInscripcion(String cursoId, String estudianteId) {
        useCase.ejecutar(cursoId, estudianteId);
    }
}
```



- Framework & Drivers
- Interface Adapters
- Application Business Rules
- Enterprise Business Rules

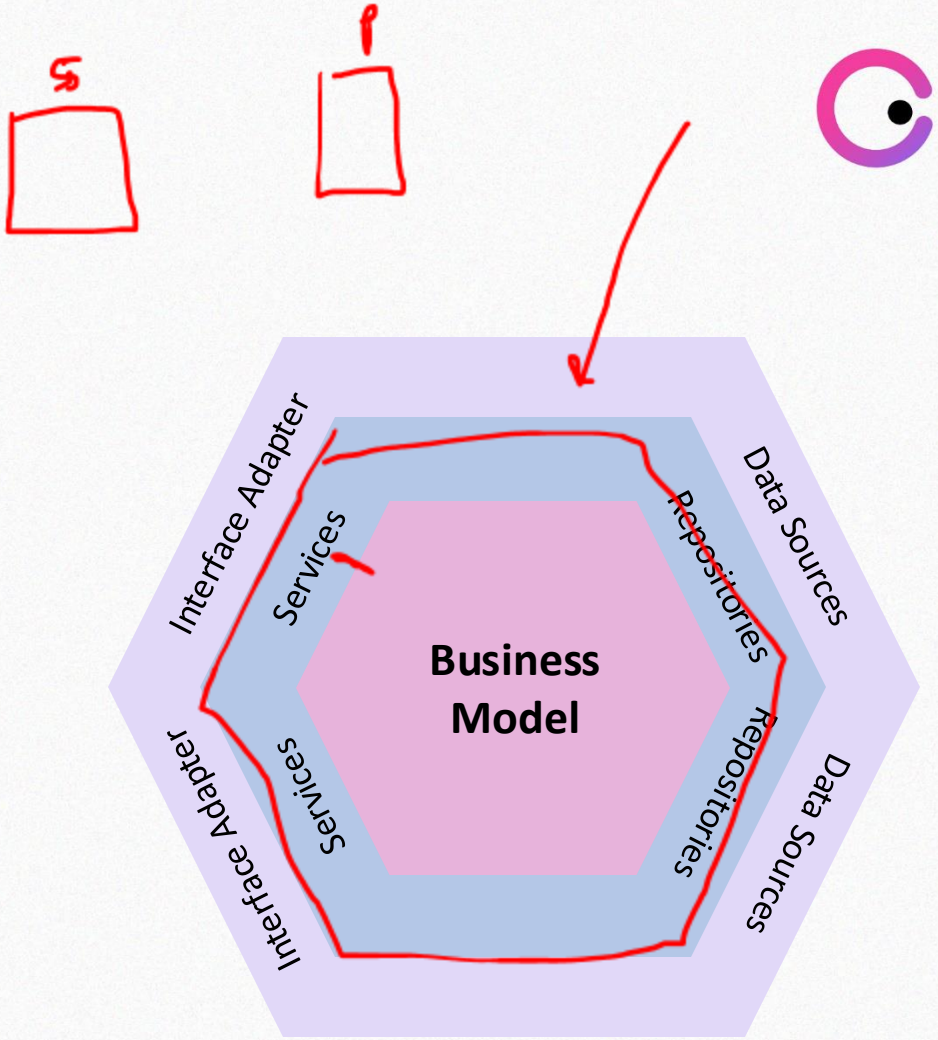
Clean Architecture



Arquitectura Hexagonal

Propuesto por **Alistair Cockburn** separa el **núcleo de la aplicación** (la lógica de negocio) de sus **interfaces externas** (como bases de datos, APIs REST, colas de mensajes, interfaces gráficas), mediante **puertos** (interfaces) y **adaptadores** (implementaciones).

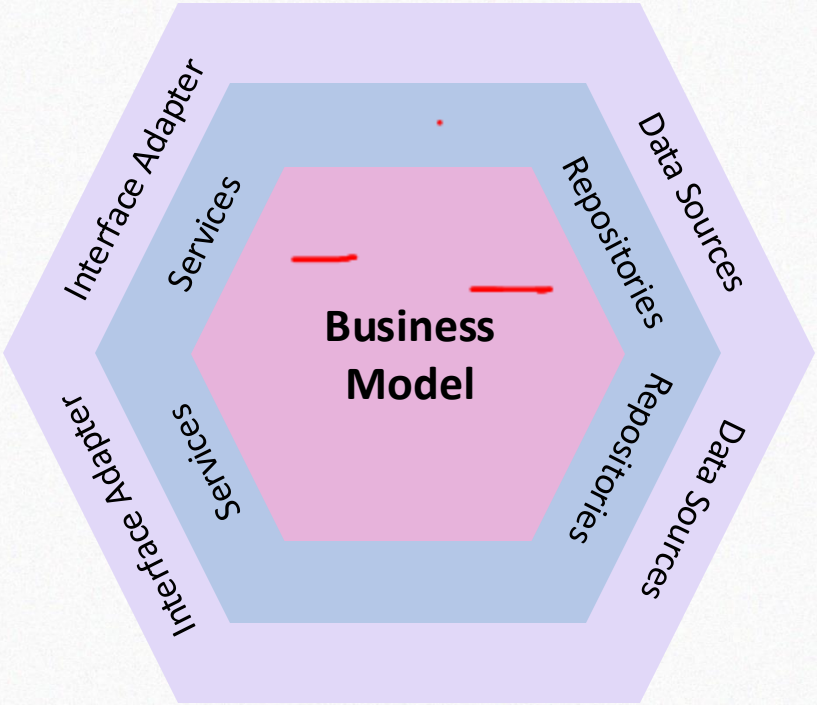
El core no debería saber nada del mundo exterior. Pero el mundo exterior sí debe conocer cómo comunicarse con él



Arquitectura Hexagonal

"Un usuario consulta el saldo de su cuenta desde una app móvil."

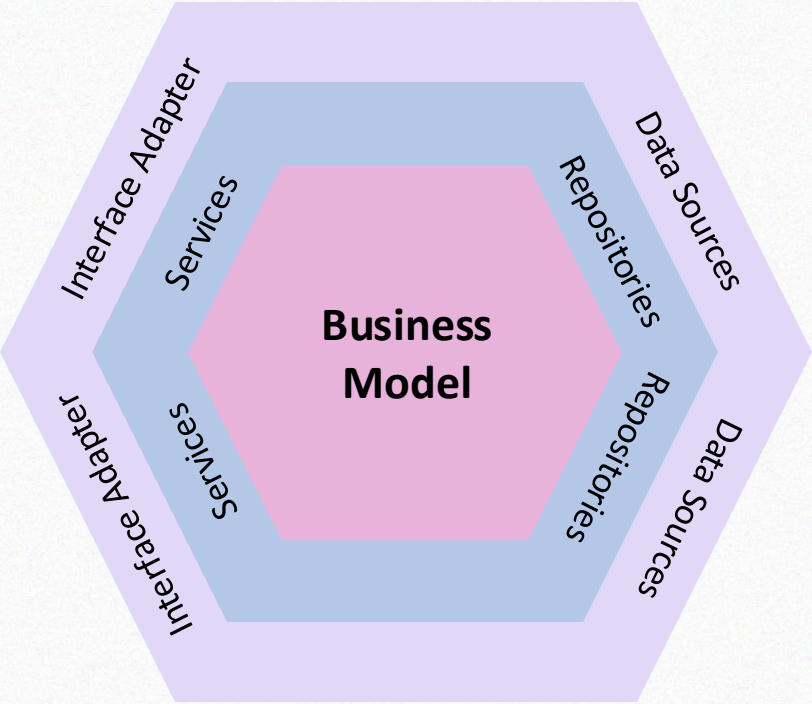
Elemento	Significado en Hexagonal
Core / Dominio	Contiene la lógica de negocio independiente
Ports (interfaces)	Definen cómo interactuar con el core (entrada y salida)
Adapters	Implementan esos puertos para interactuar con sistemas reales
Driving adapter	Llama al core (ej. controlador REST)
Driven adapter	Es llamado desde el core (ej. repositorio, API externa)



Arquitectura Hexagonal

Dominio: Entidad y lógica

```java
public class Cuenta {
    private final String id;
    private final double saldo;

    public Cuenta(String id, double saldo) {
        this.id = id;
        this.saldo = saldo;
    }

    public double consultarSaldo() {
        return saldo;
    }
}
```



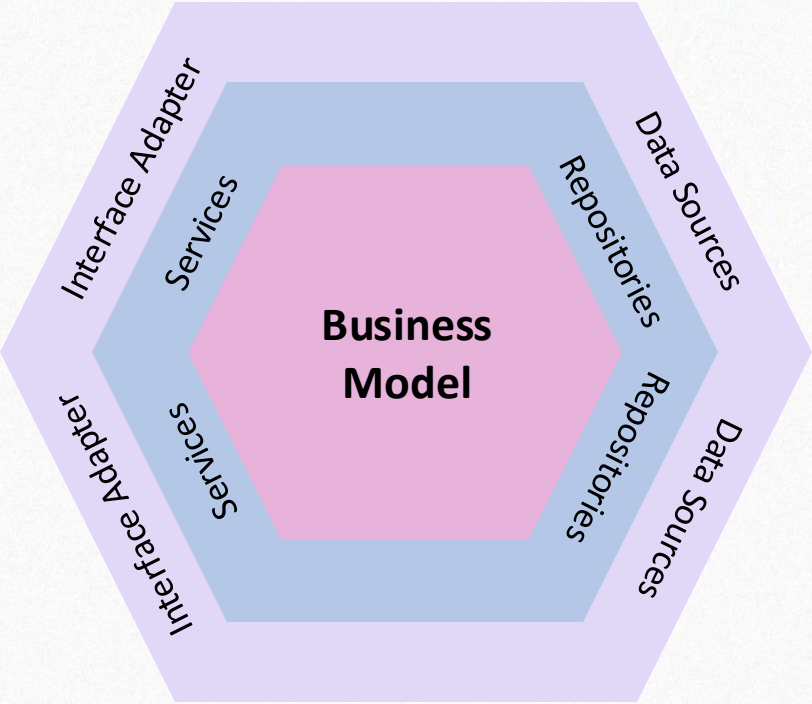
Arquitectura Hexagonal

Puerto de entrada (Driving Port)

```
public interface ConsultarSaldoUseCase {
    double consultar(String cuentaId);
}
```

Puerto de entrada (Driving Port)

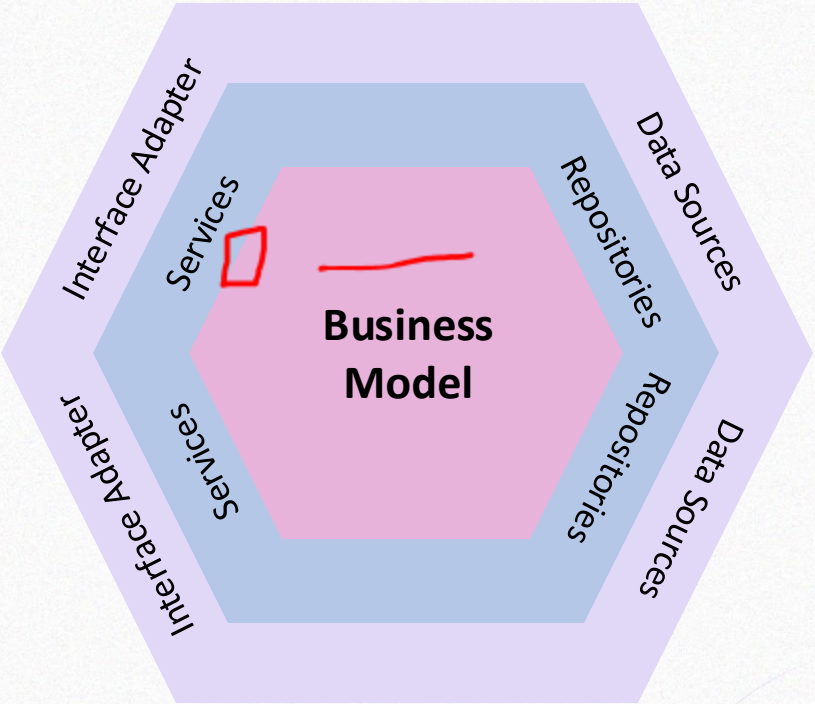
```
public interface CuentaRepository {
    Cuenta buscarPorId(String id);
}
```



Arquitectura Hexagonal

Caso de uso (implementación del puerto de entrada)

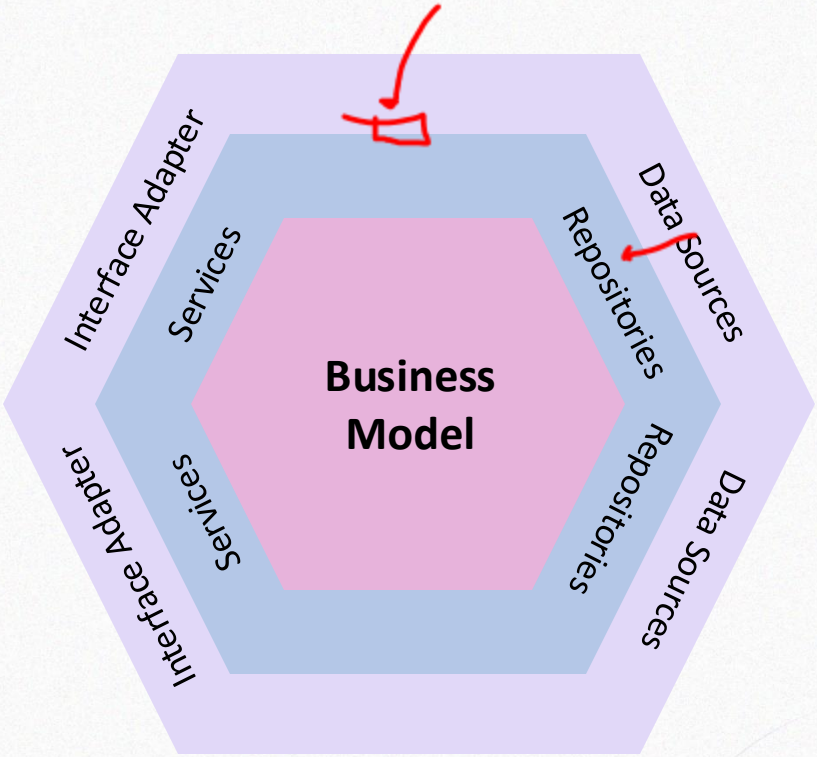
```java
public class ConsultarSaldoService implements ConsultarSaldoUseCase {
    private final CuentaRepository cuentaRepository;

    public ConsultarSaldoService(CuentaRepository cuentaRepository) {
        this.cuentaRepository = cuentaRepository;
    }

    @Override
    public double consultar(String cuentaId) {
        Cuenta cuenta = cuentaRepository.buscarPorId(cuentaId);
        return cuenta.consultarSaldo();
    }
}
```



Arquitectura Hexagonal

Adaptador de entrada (REST controller)

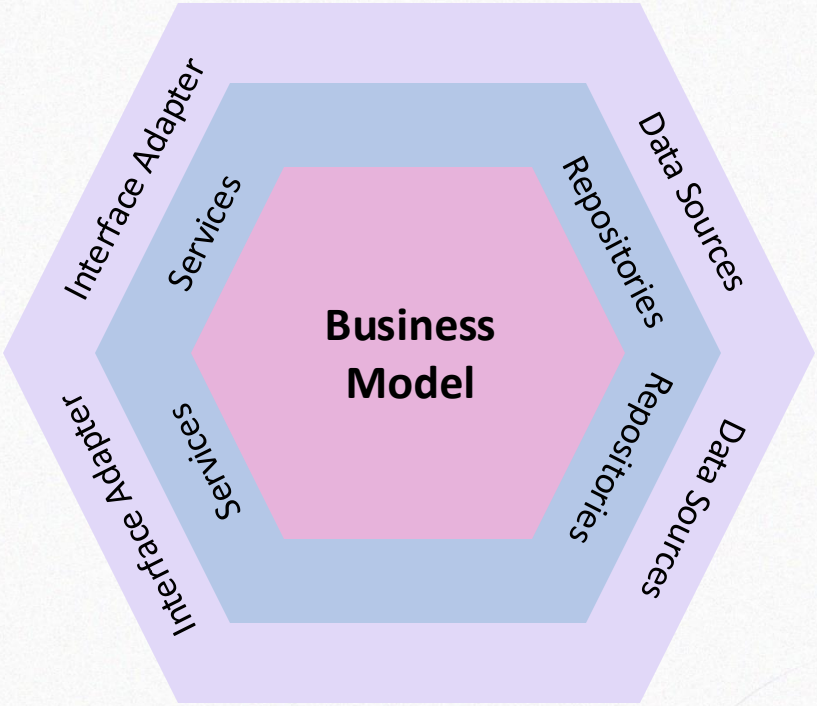
```java
@RestController
public class CuentaController {
    private final ConsultarSaldoUseCase consultarSaldoUseCase;

    public CuentaController(ConsultarSaldoUseCase useCase) {
        this.consultarSaldoUseCase = useCase;
    }

    @GetMapping("/cuentas/{id}/saldo")
    public ResponseEntity<Double> obtenerSaldo(@PathVariable String id) {
        double saldo = consultarSaldoUseCase.consultar(id);
        return ResponseEntity.ok(saldo);
    }
}
```

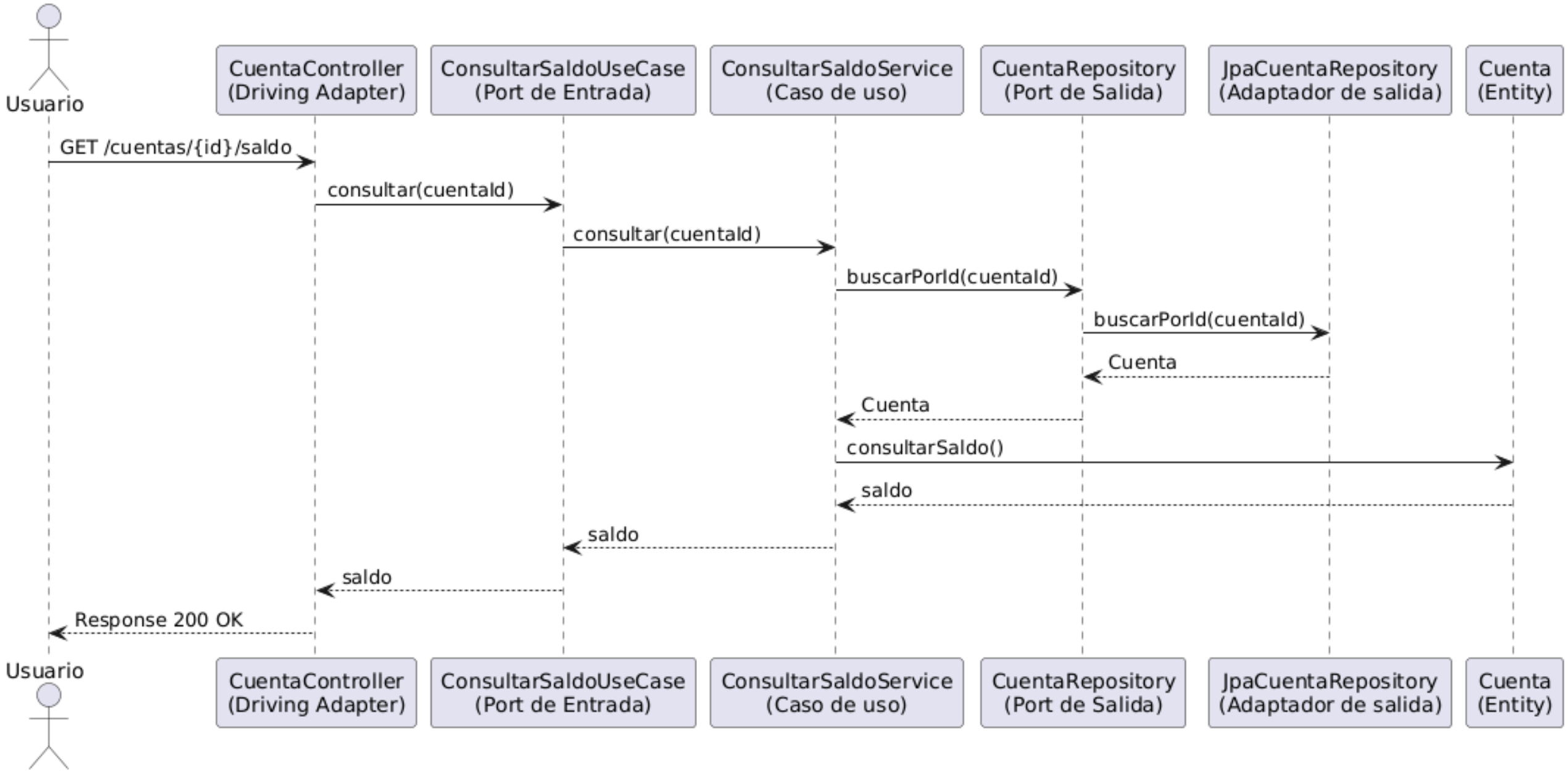


Arquitectura Hexagonal

Adaptador de salida (infraestructura)

```java
@Repository
public class JpaCuentaRepository implements CuentaRepository {
    @Override
    public Cuenta buscarPorId(String id) {
        // Simulación. Aquí usarías JPA o JDBC.
        return new Cuenta(id, 1500.0);
    }
}
```




Usuario
CuentaController (Driving Adapter)
ConsultarSaldoUseCase (Port de Entrada)
ConsultarSaldoService (Caso de uso)
CuentaRepository (Port de Salida)
JpaCuentaRepository (Adaptador de salida)
Cuenta (Entity)
GET /cuentas/{id}/saldo
consultar(cuentaId)
consultar(cuentaId)
buscarPorId(cuentaId)
buscarPorId(cuentaId)
Cuenta
Cuenta
consultarSaldo()
saldo
saldo
saldo
Response 200 OK
Usuario
CuentaController (Driving Adapter)
ConsultarSaldoUseCase (Port de Entrada)
ConsultarSaldoService (Caso de uso)
CuentaRepository (Port de Salida)
JpaCuentaRepository (Adaptador de salida)
Cuenta (Entity)

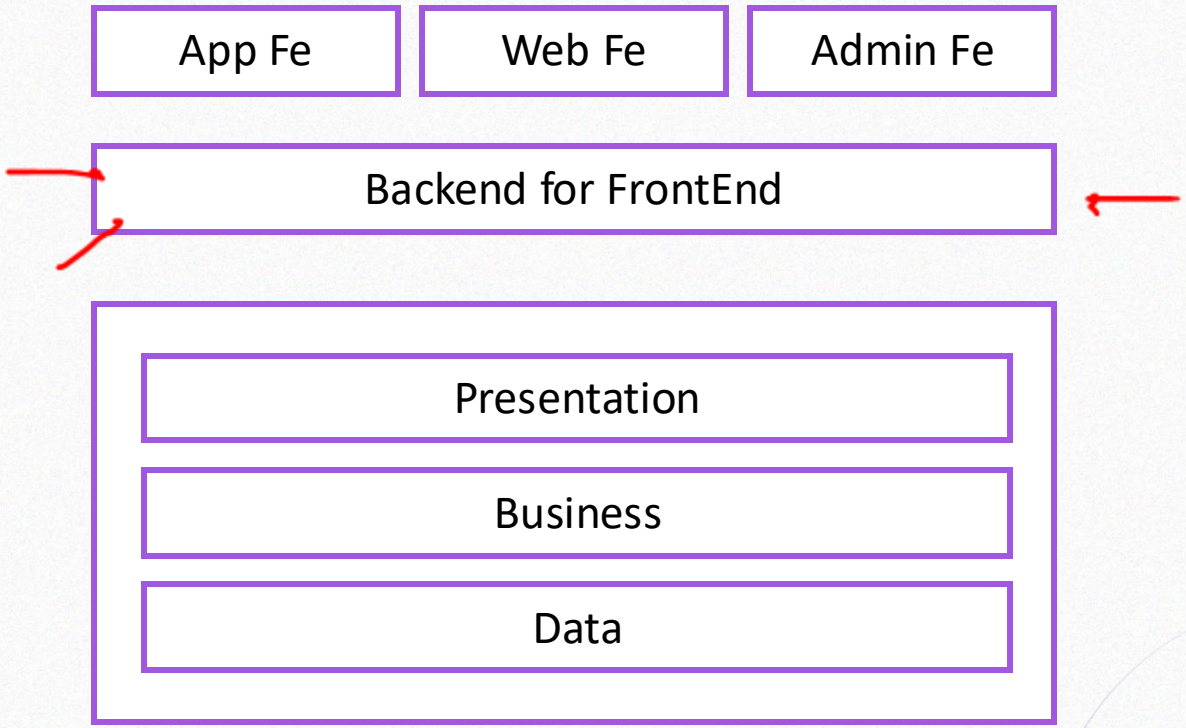
Backend for Frontend y capas en el frontend moderno

"Antes, el frontend pedía datos. Ahora, exige su propio backend."

Es un **patrón arquitectónico** en el que se crea un **backend específico para cada tipo de cliente** (por ejemplo, web, móvil, smartwatch), adaptando los datos y flujos de negocio según lo que ese cliente necesita.

¿Qué problema resuelve?

- **Evita que el frontend haga lógica de negocio** (e.g., calcular totales, aplicar reglas).
- **Evita la sobrecarga de datos:** un GET /user que trae 50 campos, de los cuales la app móvil solo necesita 5.
- **Permite adaptar flujos sin romper la API general.**



Backend for Frontend y capas en el frontend moderno

“Antes, el frontend pedía datos. Ahora, exige su propio backend.”

La arquitectura moderna ya no reniega de las capas. Las usa… pero **cuando y donde tienen sentido.**

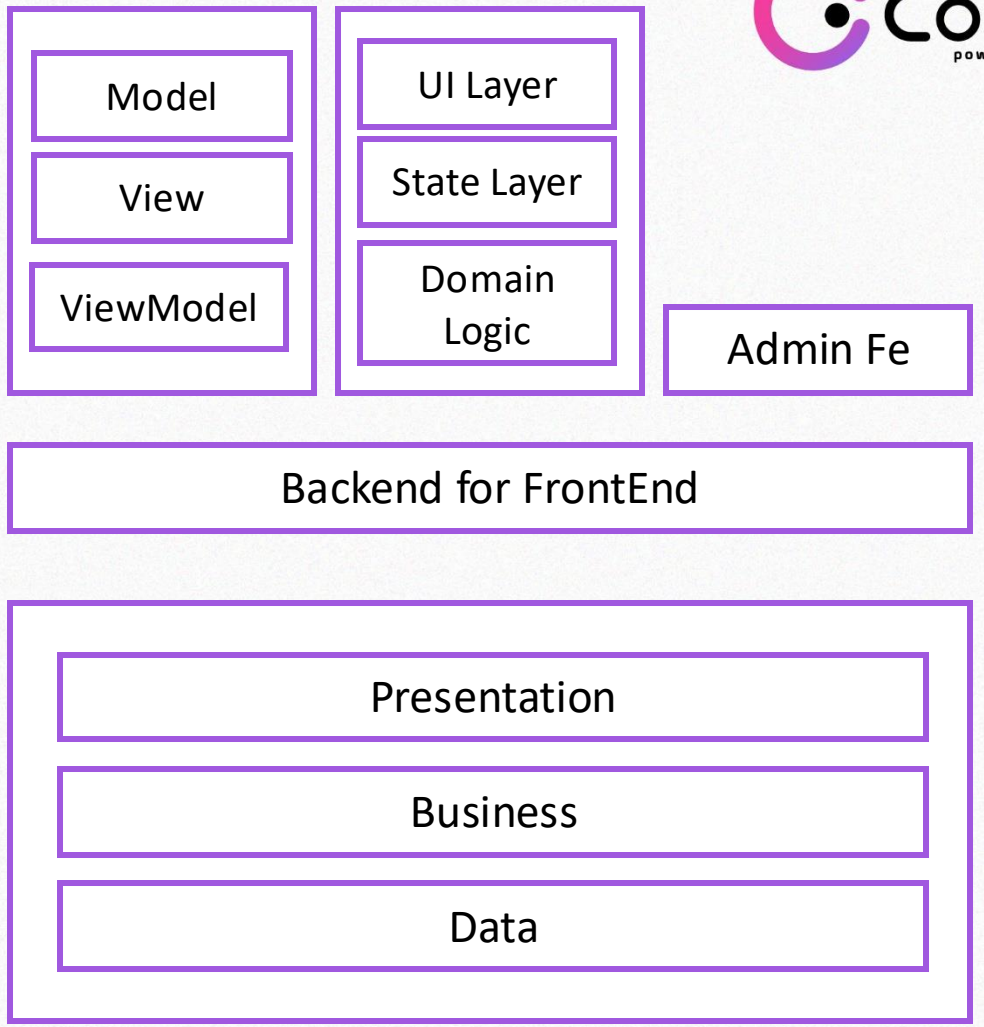
Antes: Capas por doctrina

- Se aplicaban capas “porque así se hace”.
- Lógica fragmentada en múltiples archivos sin lógica real.
- DTO → Mapper → VO → Entity… incluso para flujos triviales.
- Duplicación sin propósito y sobreingeniería.

Hoy: Capas por propósito

Se usa una capa si:

- Encapsula una **responsabilidad clara y cambiante**,
- Mejora la **testabilidad**,
- Separa **preocupaciones que evolucionan distinto**.



Caso Practico

Modelando una arquitectura en capas

Una clínica quiere que los pacientes reserven turnos con sus médicos.

Cada paciente debe poder:

- Ver médicos disponibles por especialidad
- Ver horarios libres
- Reservar un turno
- Cancelar un turno
- Consultar su historial de citas

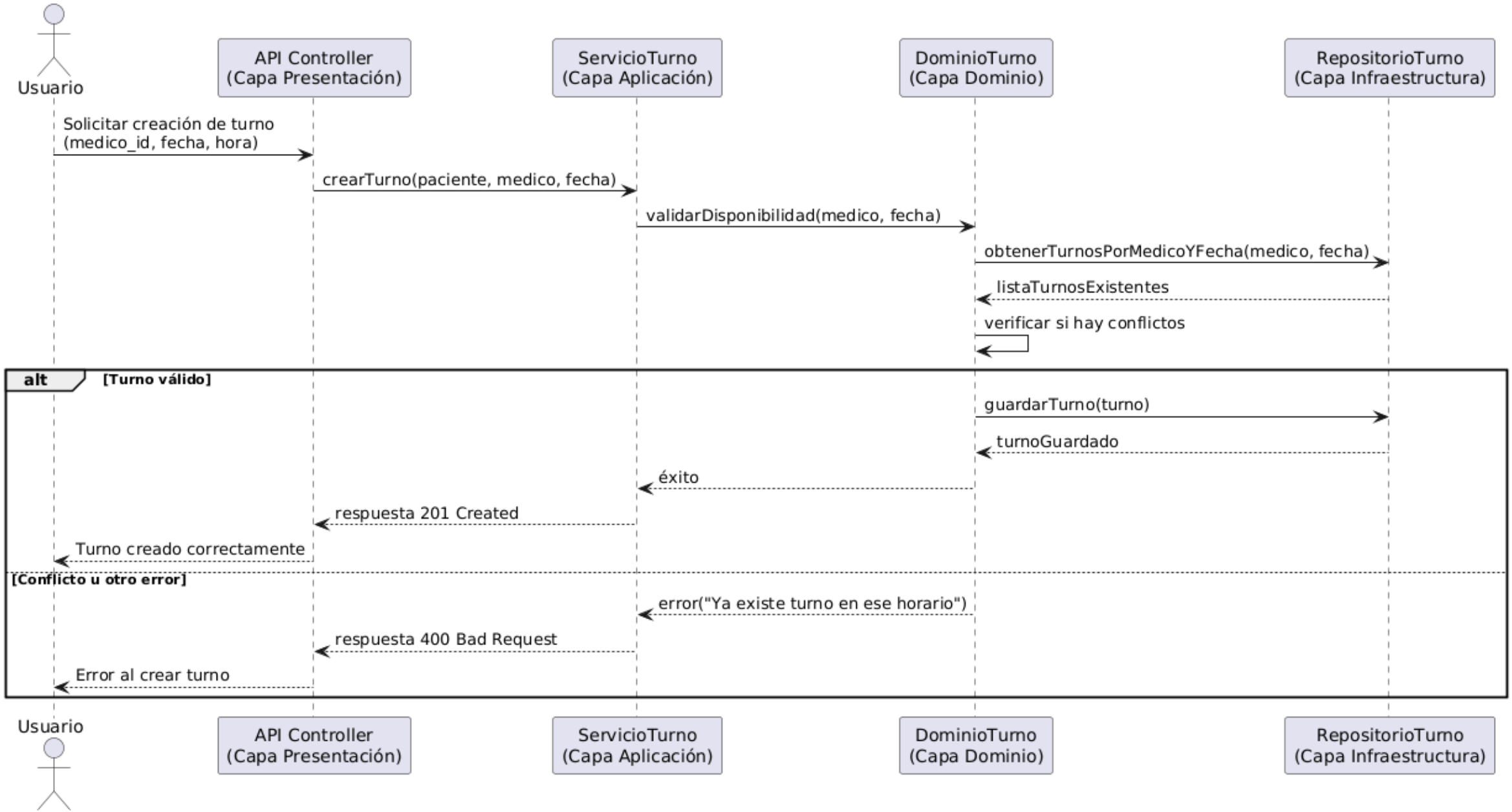
Caso Practico

¿Qué capa creen que debería validar si el médico está disponible?

¿Y cuál debería guardar el turno?

Caso Practico

Capa de Presentación	• Recibe la solicitud con paciente_id, medico_id y horario • Valida formato básico
Capa de Aplicación (Servicio)	• Orquesta la lógica: llama a Dominio y Repositorio • Devuelve un DTO o estado
Capa de Dominio	• Valida que el horario esté disponible • Aplica reglas (solo puede agendarse 1 día antes, etc.)
Capa de Datos/Infraestructura	• Consulta a la base de datos • Persiste el nuevo turno


Usuario
API Controller
(Capa Presentación)
ServicioTurno
(Capa Aplicación)
DominioTurno
(Capa Dominio)
RepositorioTurno
(Capa Infraestructura)
Solicitar creación de turno
(medico_id, fecha, hora)
crearTurno(paciente, medico, fecha)
validarDisponibilidad(medico, fecha)
obtenerTurnosPorMedicoYFecha(medico, fecha)
listaTurnosExistentes
verificar si hay conflictos
alt
[Turno válido]
guardarTurno(turno)
turnoGuardado
éxito
respuesta 201 Created
Turno creado correctamente
[Conflicto u otro error]
error("Ya existe turno en ese horario")
respuesta 400 Bad Request
Error al crear turno
Usuario
API Controller
(Capa Presentación)
ServicioTurno
(Capa Aplicación)
DominioTurno
(Capa Dominio)
RepositorioTurno
(Capa Infraestructura)

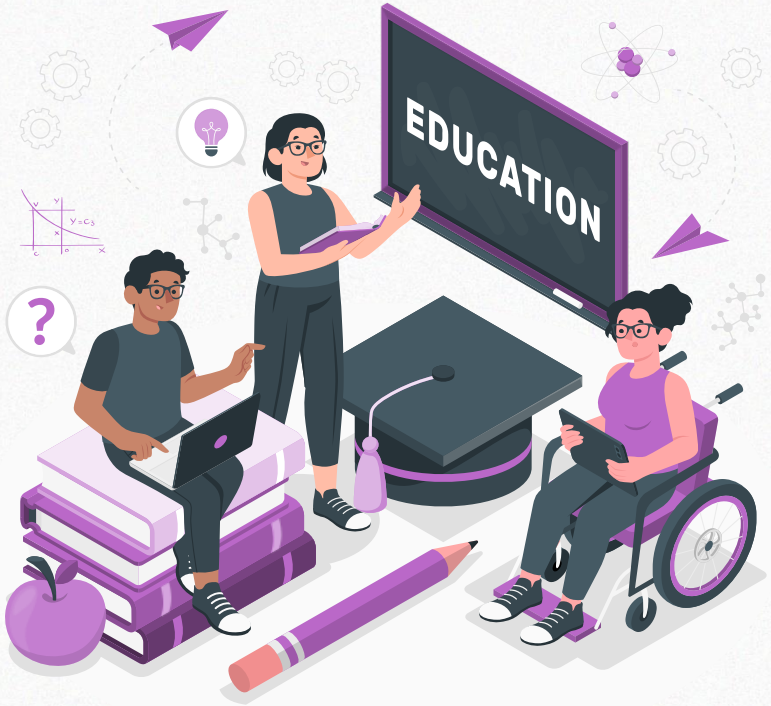
Caso Practico

Aplicando un enfoque moderno

Una startup está construyendo una plataforma tipo Udemy.

Necesitan una arquitectura que permita:

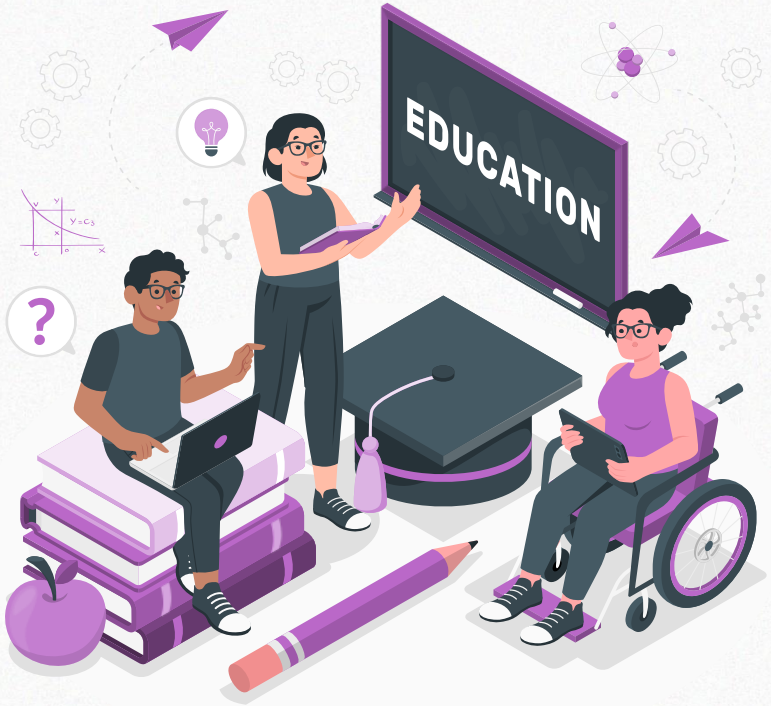
- Registrar cursos
- Inscribir usuarios
- Gestionar el avance de los estudiantes
- Exponer la plataforma por web y móvil
- Agregar nuevas formas de pago sin afectar el core



Caso Practico

¿Qué pasaría si el equipo de pagos quiere cambiar de Stripe a MercadoPago?

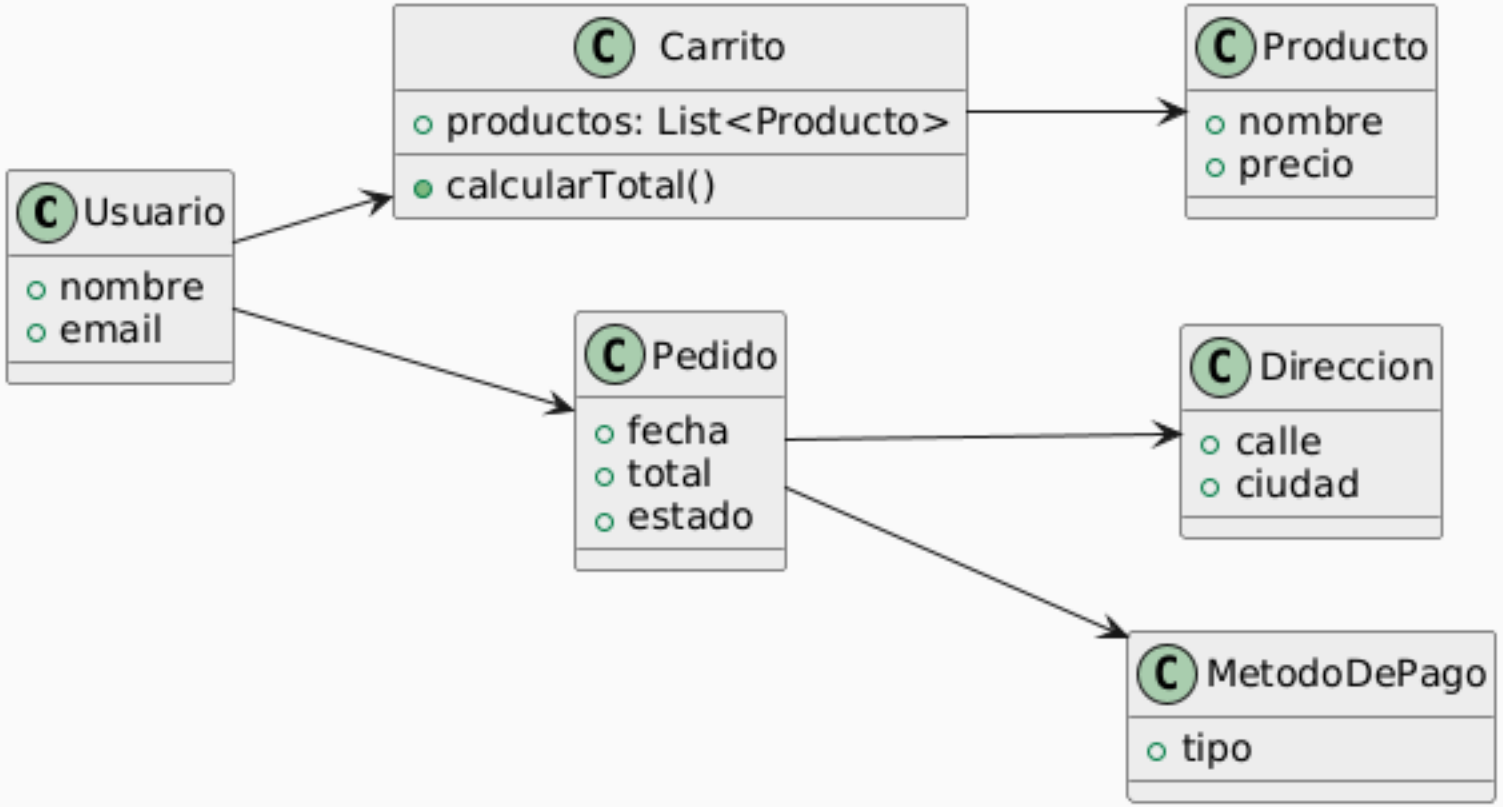
¿O si mañana lanzamos una app nativa y necesitamos mostrar solo los cursos comprados?



Vista Logica

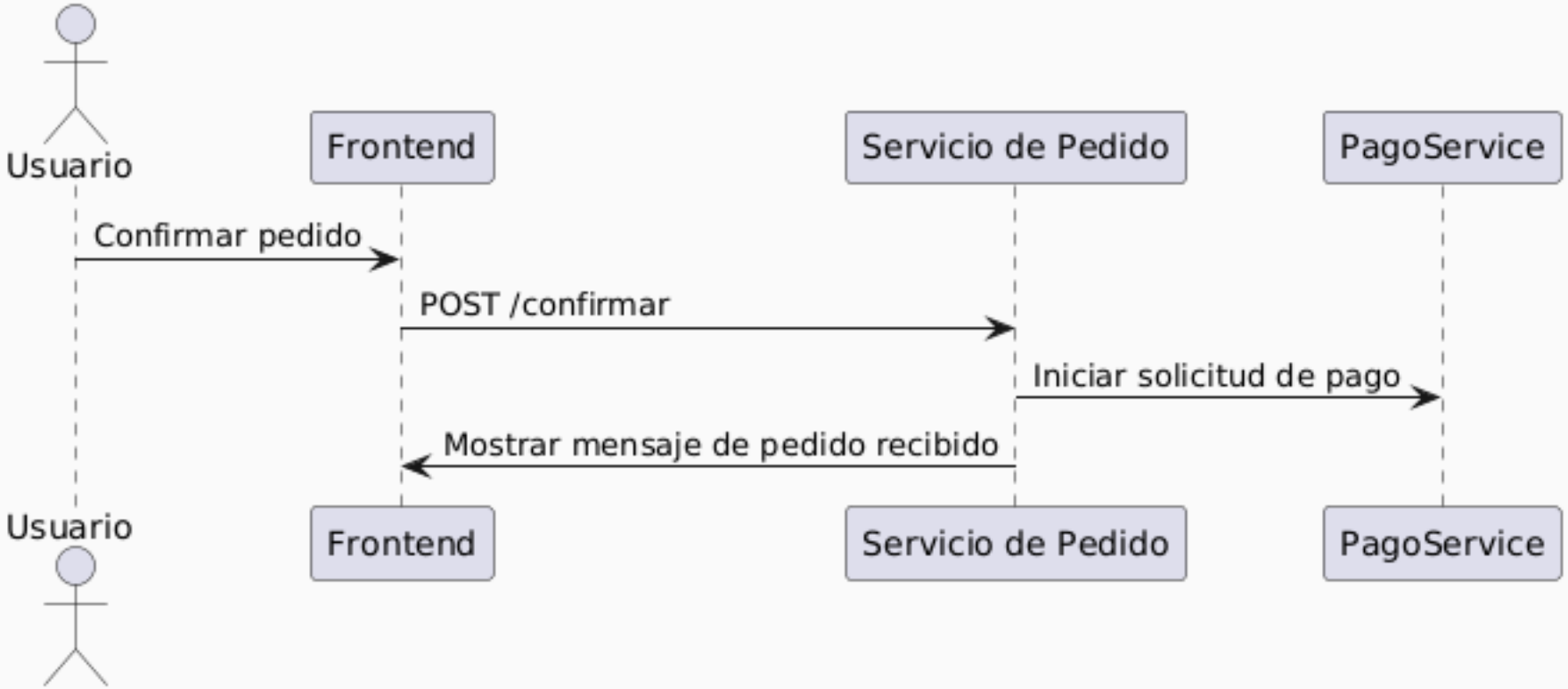
Representa las entidades y relaciones del dominio funcional. Es clave para el modelado orientado a objetos.

- Usuario interactúa con su Carrito.
- Carrito genera un Pedido, que finalmente se procesa mediante PagoService.
- Esta vista ayuda a estructurar el modelo de dominio y delimita responsabilidades.

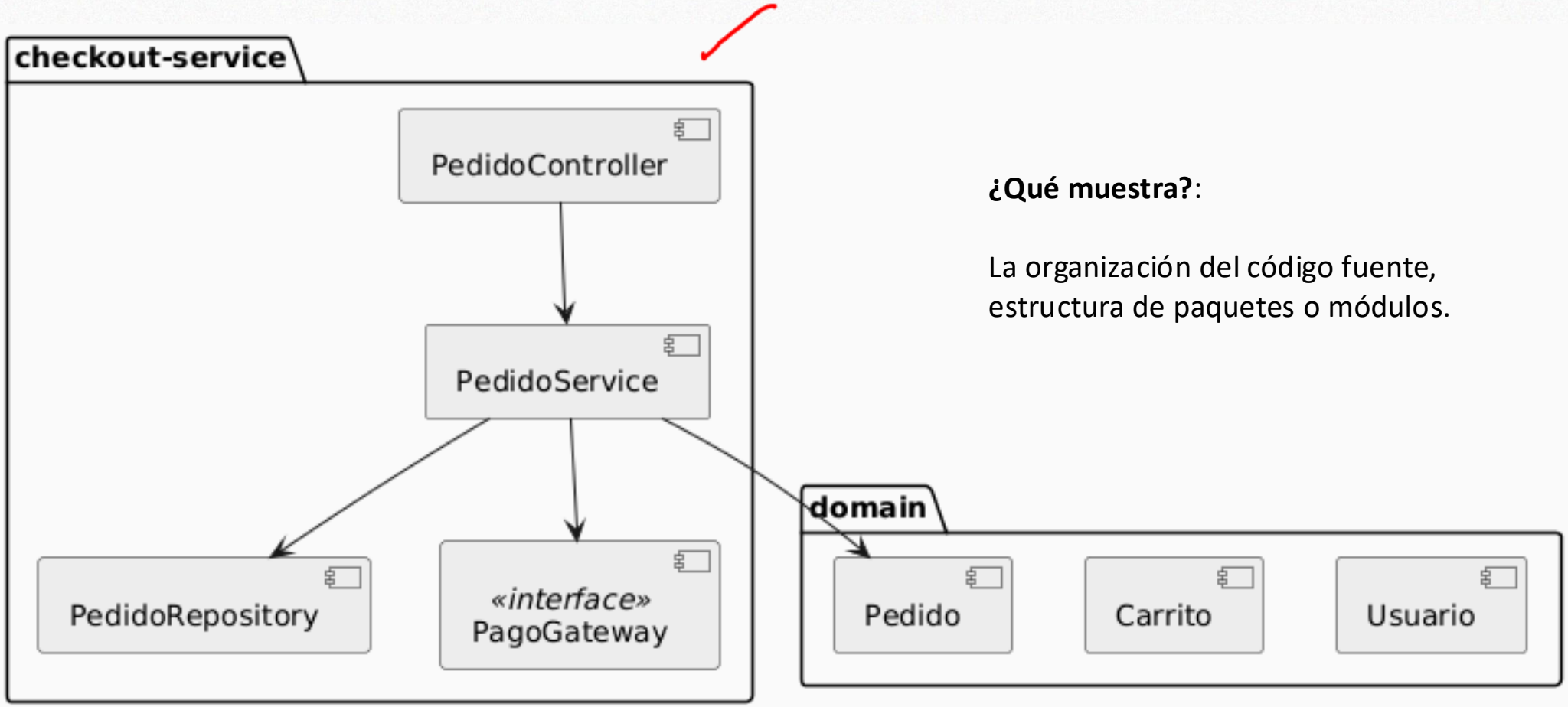


Vista de Proceso

Muestra comportamiento dinámico, concurrencia y comunicación. Útil para decisiones de rendimiento y tolerancia a fallos.



Vista de DESARROLLO
(Development/Implementation View)



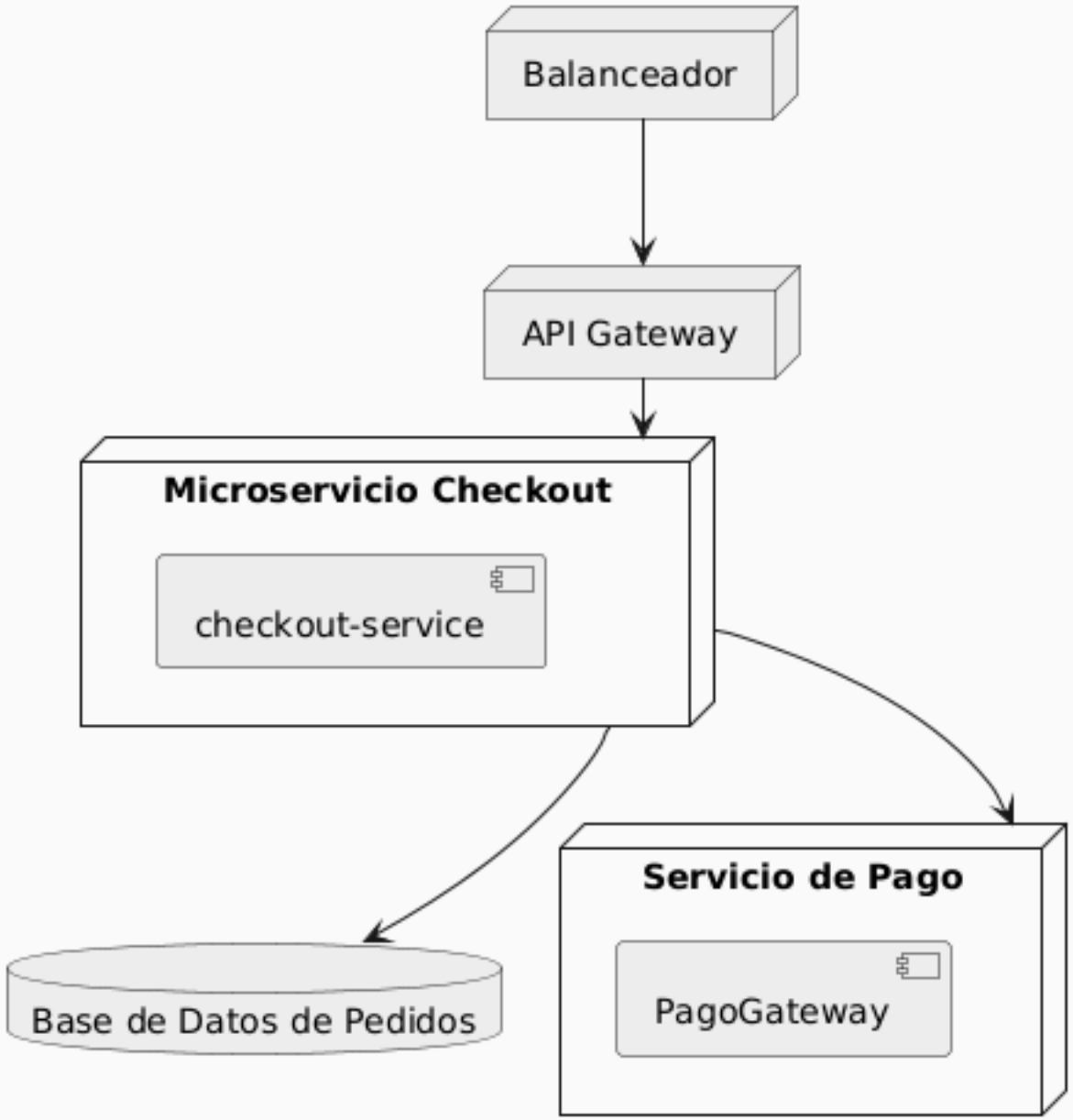
¿Qué muestra?:

La organización del código fuente, estructura de paquetes o módulos.

Vista de DESPLIEGUE (Deployment View)

¿Qué muestra?:

Infraestructura física/virtual, nodos, redes, contenedores, cloud.



Estilos Arquitectonicos

¿Podrías describir la arquitectura de un sistema sin hablar de ninguna tecnología?

Un estilo arquitectónico define una familia de sistemas en términos de un vocabulario de componentes y conectores, y un conjunto de restricciones sobre cómo pueden ser combinados.

— Shaw & Garlan, 1996

Un estilo arquitectonico **NO ES:**

- Un framework (como Spring)
- No es un patrón (como Factory)
- No es una herramienta (como Kubernetes)

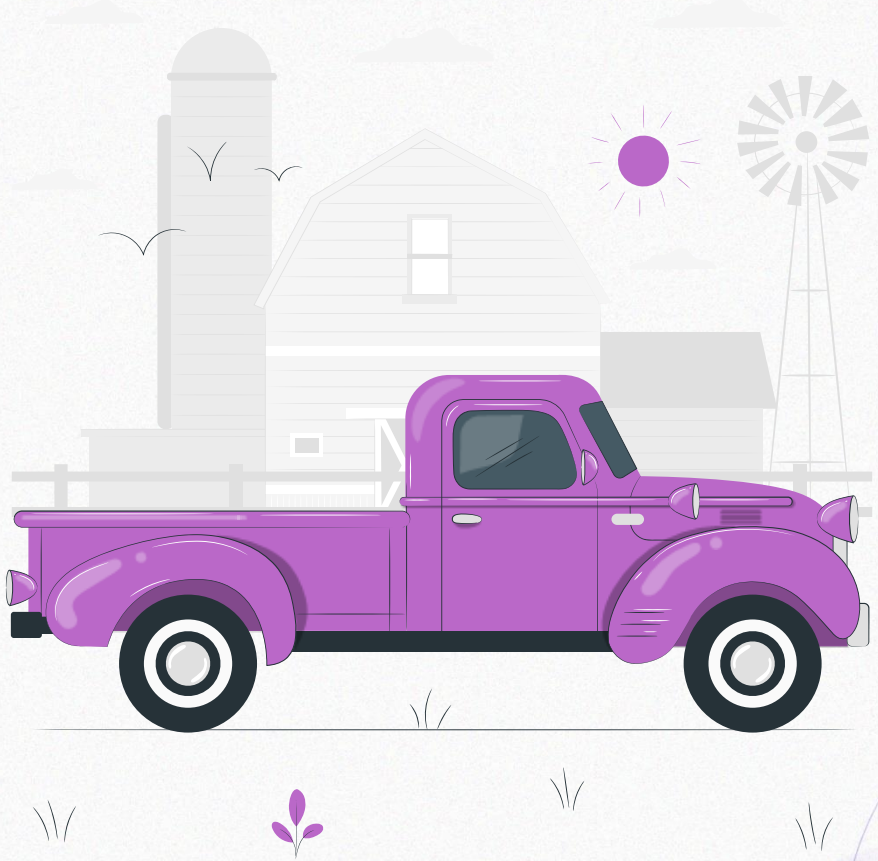
Recuerda:

- Un **estilo arquitectónico** = modelo general y estructural.
- Un **patrón** = solución recurrente a un problema específico (ej. MVC, Singleton).
- Una **tecnología** = herramienta concreta que puede implementar un estilo (ej. Django, Kubernetes).

Estilos Arquitectonicos Clasicos

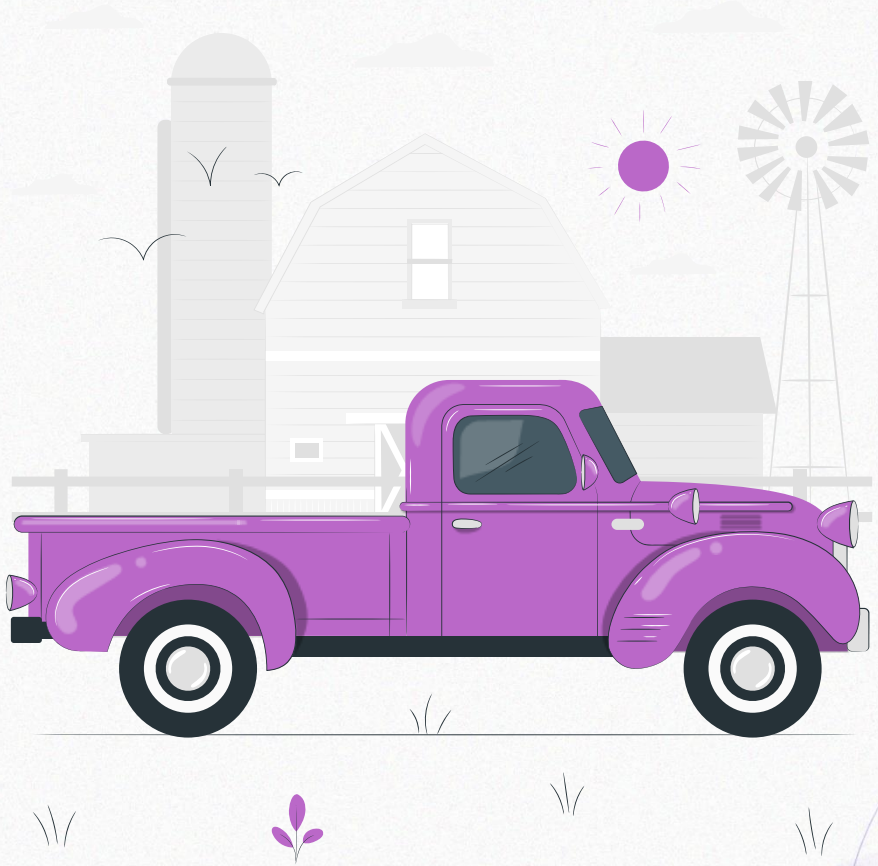
¿Por qué hablar de “clásicos”?

- Fueron los **primeros en sistematizarse académicamente**.
- Sirven como **fundamento conceptual** para estilos modernos (como microservicios o arquitecturas hexagonales).
- Permiten entender el **origen de muchas decisiones arquitectónicas actuales**, especialmente en migraciones o rediseños.



Estilos Arquitectonicos Clasicos

- Monolítico
- Cliente-servidor
- En capas (layered)
- Pipe and Filter
- Event-driven
- Microservicios



Monolítico

En una arquitectura monolítica, todos los componentes de un sistema de software se construyen y despliegan como una única unidad, que se ejecuta dentro de un mismo proceso o contenedor de ejecución.

— Adaptado de Bass, Clements & Kazman, *Software Architecture in Practice* (2021)

Monolítico

Un monolito no implica necesariamente “mala práctica” o “código espagueti”. Se refiere a **una unidad indivisible de despliegue**, no a la calidad del diseño interno.

Características estructurales

- Toda la lógica de negocio, la capa de presentación y el acceso a datos viven en **un solo ejecutable o aplicación empaquetada**.
- Generalmente se usa **una sola base de código**, aunque puede estar organizada en módulos internos.
- El despliegue y el ciclo de vida es **unitario**: se compila, prueba y despliega como un todo.
- La comunicación entre componentes internos es **directa (llamadas locales o en memoria)**.

Monolítico

Un monolito no implica necesariamente “mala práctica” o “código espagueti”. Se refiere a **una unidad indivisible de despliegue**, no a la calidad del diseño interno.

Ejemplos de la industria

- **Aplicaciones Java EE tradicionales** con JSF + EJBs + JPA.
- **Ruby on Rails** o **Laravel** cuando todo vive en un solo proyecto.
- **Sistemas heredados (legacy)** en bancos, ERP, sistemas clínicos

Monolítico

Un monolito no implica necesariamente “mala práctica” o “código espagueti”. Se refiere a **una unidad indivisible de despliegue**, no a la calidad del diseño interno.

Ventajas

- Simplicidad inicial
- Fácil de desarrollar
- Despliegue único
- Ideal para MVPs
- Transacciones más simples

Desventajas

- Difícil de escalar selectivamente
- Baja resiliencia
- Despliegue riesgoso
- Evolución lenta del equipo
- Tecnologías acopladas

Monolítico

¿Cuándo es razonable usar un monolito?

- Startups que lanzan un MVP y buscan **time-to-market** rápido.
- Equipos pequeños que **no justifican complejidad operativa**.
- Sistemas bien definidos y estables, con cambios poco frecuentes.
- Aplicaciones de línea de negocio interna **con bajo volumen de usuarios**.

Monolítico

Buenas practicas

- Diseñar el código **modularmente**, aunque se despliegue como un todo.
- Usar capas bien separadas (UI, servicios, dominio, persistencia).
- Implementar pruebas automatizadas (unitarias e integradas).
- Documentar explícitamente las **dependencias internas**.
- Dejar puntos de extensión (facades, adaptadores) que faciliten futuras migraciones.

Una startup lanza una app de delivery que funciona bien como monolito. A los 6 meses, quieren expandirse a 3 países con pagos, logísticas y monedas distintas.

Un banco tiene un core bancario monolítico con más de 20 años, que aún procesa millones de transacciones diarias sin errores.

¿Puedes diseñar un monolito bien estructurado que luego pueda evolucionar hacia microservicios?

¿Qué consecuencias puede tener forzar microservicios en un sistema que no necesita esa granularidad aún?

¿Cómo construir un buen monolito?

“Monolítico no significa caótico.”
Un buen monolito es modular,
testeable y evolutivo.

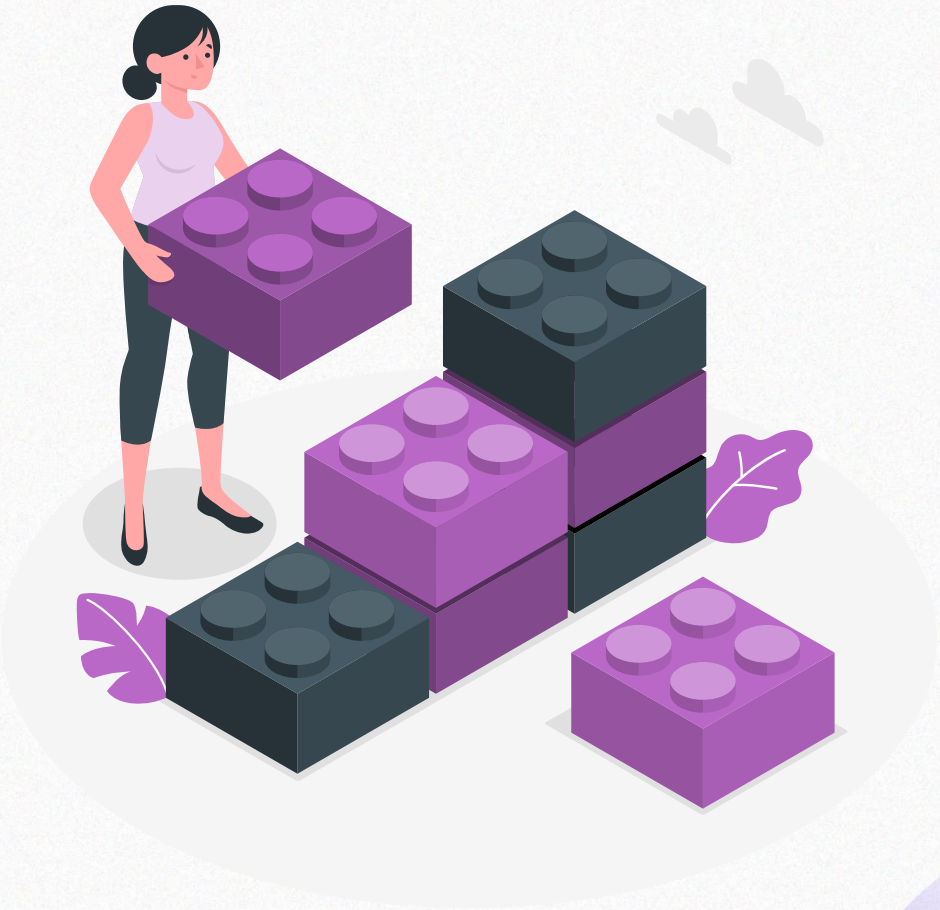
Principios clave para un monolito saludable

Modularidad interna: Separar claramente responsabilidades dentro del código.

Modularidad: División por **función o dominio**

Capas: División por **responsabilidad técnica y** Depende de **flujo vertical**

Richards & Ford (2020) lo llaman "estructura vertical vs. horizontal".



¿Cómo modularizar un monolito?

Separación por dominio funcional: Cada módulo representa una unidad del negocio (no técnica).

Ej: pedidos, productos, clientes, notificaciones.

Límites explícitoS: No se accede directamente a funciones de otros módulos. (Solo a través de contratos APIs internas).

Dirección de dependencias: Si A usa B, B no debe conocer a A. Usar inversión de dependencias si es necesario.

Evitar el “shared” tóxico No tener un módulo utils/ al que todos dependen.

¿Cómo modularizar un monolito?

```
# controller.py
def confirmar_pedido(usuario_id, producto_id):
    user = buscar_usuario(usuario_id)
    producto = buscar_producto(producto_id)
    if producto.stock > 0:
        registrar_pedido(usuario_id, producto_id)
```

controlador llama lógica de usuarios y productos directamente

¿Cómo modularizar un monolito?

```
módulo: pedidos/
   - servicio_confirmacion.py
módulo: usuarios/
   - api.py
módulo: productos/
   - api.py
```

```
# pedidos/servicio_confirmacion.py
from usuarios.api import obtener_usuario
from productos.api import obtener_producto


def confirmar(usuario_id, producto_id):
    user = obtener_usuario(usuario_id)
    producto = obtener_producto(producto_id)
    ...
```

Dilema

Tienes un equipo de 3 programadores y necesitas lanzar un producto funcional en 4 meses. El inversionista espera ver métricas reales y los primeros clientes en la plataforma.

- ¿Qué tipo de arquitectura propondrías?
- ¿Cómo argumentarías tu decisión ante un CTO que insiste en usar Kubernetes, eventos y microservicios desde el día uno?

Cliente - Servidor

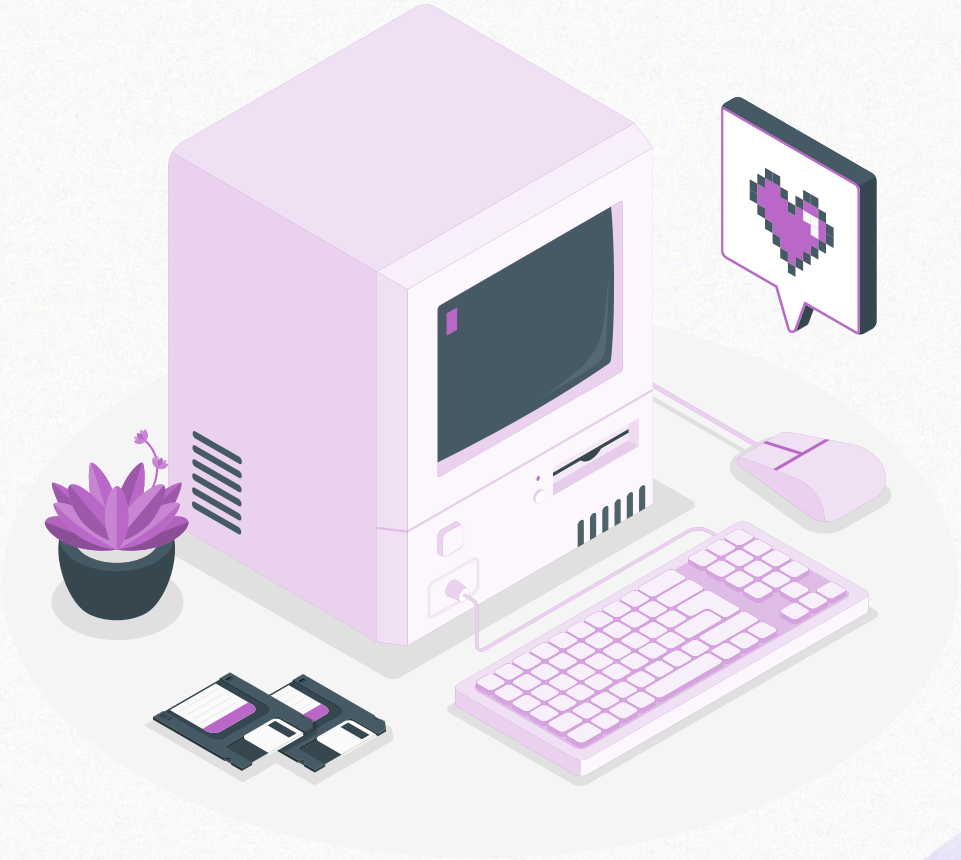
Cliente-servidor es un estilo arquitectónico en el que un conjunto de clientes interactúan con uno o más servidores mediante peticiones explícitas, accediendo a recursos o servicios controlados por estos últimos.

— Basado en Shaw & Garlan, *Software Architecture: Perspectives on an Emerging Discipline* (1996)

Cliente - Servidor

¿Cómo creen que se distribuía el software antes de que existiera internet?

¿Qué problema aparecía cuando una empresa tenía muchas sedes?



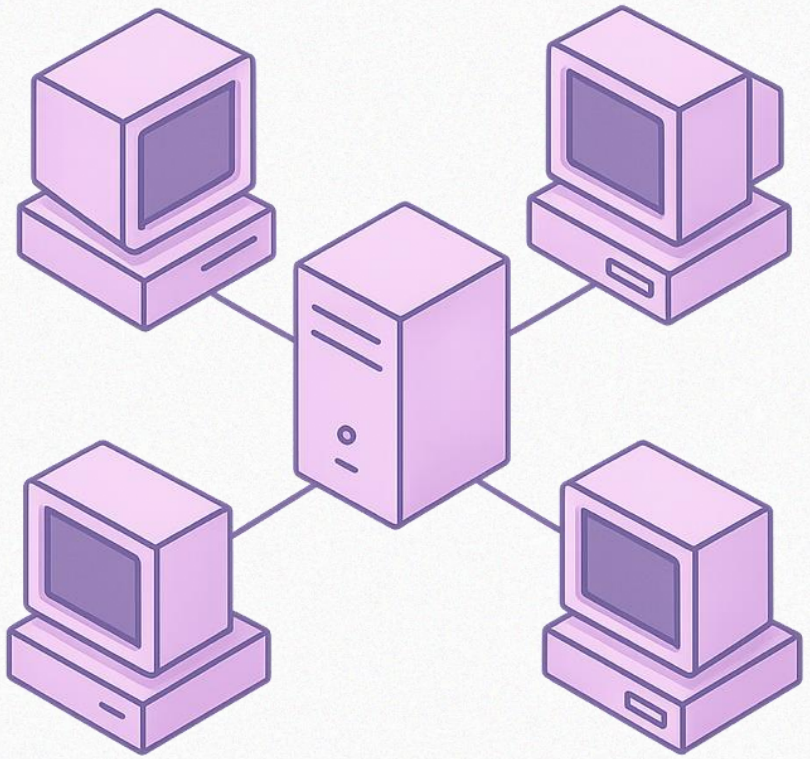
Cliente - Servidor

Antes de la masificación de la red, el software era ejecutado **localmente**, en la máquina del usuario.

Cada instalación era independiente. Si se dañaba, había que reinstalar. Si salía una nueva versión, había que ir PC por PC y actualizar.

Con la **difusión de las redes locales (LAN)** y luego **internet**, nace una pregunta clave:

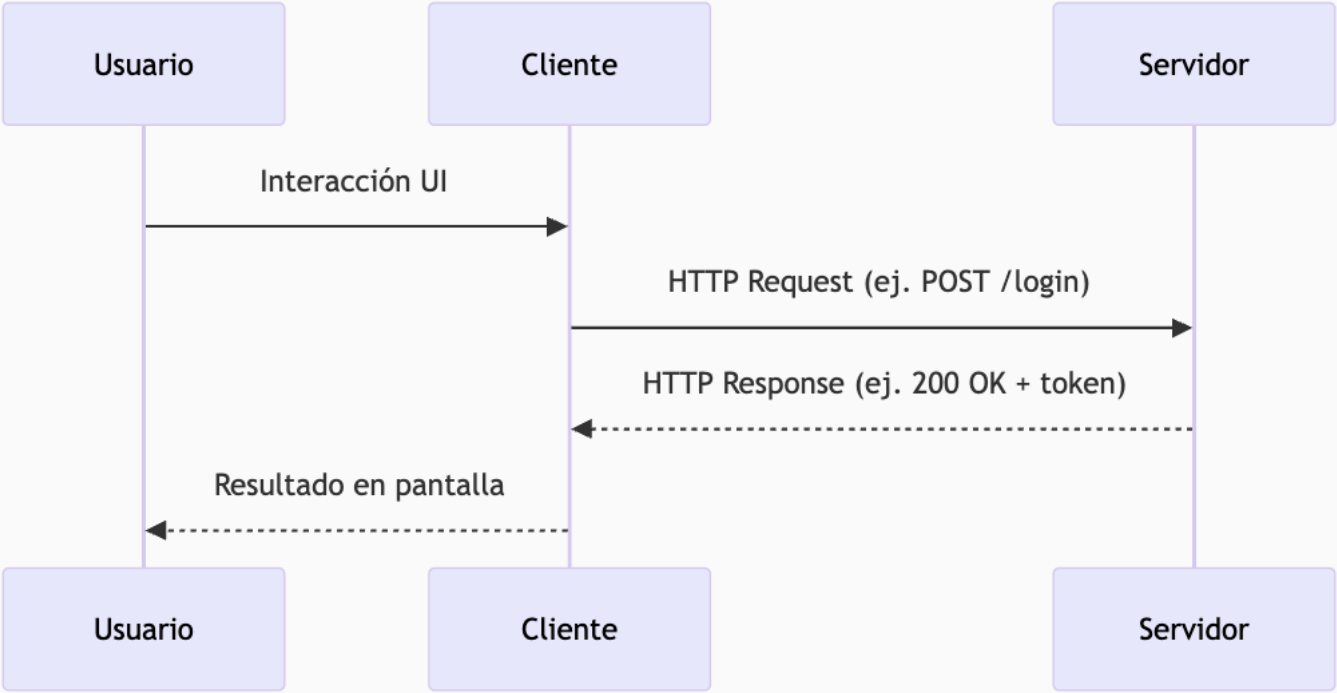
¿Y si centralizamos la lógica del sistema en un lugar, y dejamos que los usuarios accedan remotamente?



Cliente - Servidor

Cliente
- Inicia la interacción.
- Solicita recursos o acciones.

Servidor
- Espera peticiones.
- Procesa solicitudes y devuelve respuestas.



Ejemplo de cliente servidor

```python
# servidor.py
import socket

servidor = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
servidor.bind(('0.0.0.0', 12345))
servidor.listen(1)
print("Servidor esperando conexión...")

conexion, direccion = servidor.accept()
print(f"Conectado desde: {direccion}")

mensaje = conexion.recv(1024).decode()
print(f"Cliente dice: {mensaje}")

conexion.send("Hola cliente, te recibo claro y fuerte.".encode())
conexion.close()
servidor.close()
```

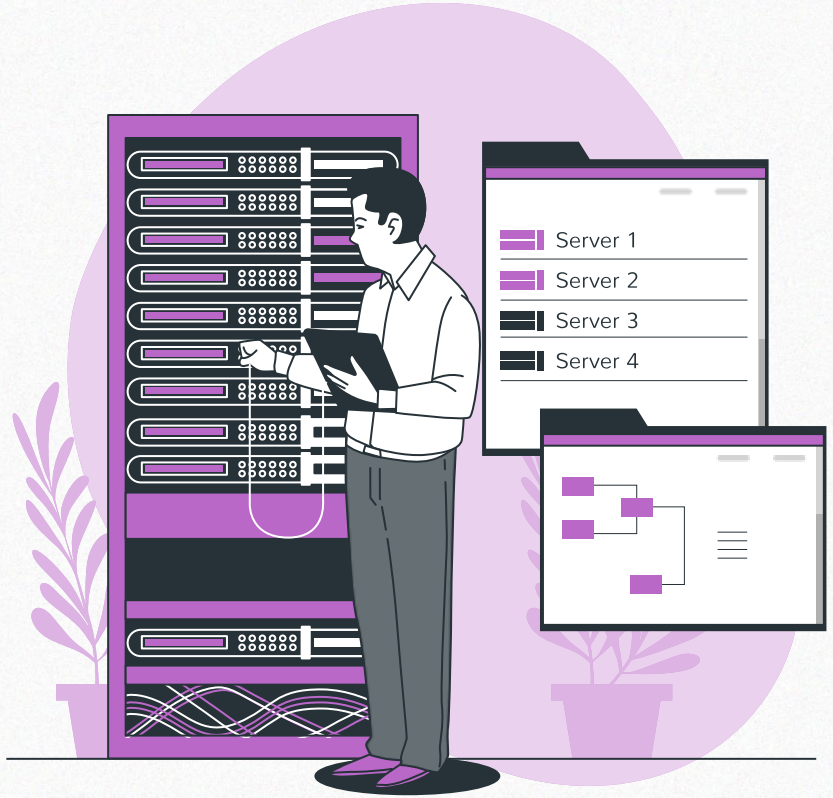
Ejemplo de cliente servidor

```python
# cliente.py
import socket

cliente = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
cliente.connect(('127.0.0.1', 12345))

cliente.send("Hola servidor, soy el cliente.".encode())
respuesta = cliente.recv(1024).decode()
print(f"Servidor dice: {respuesta}")

cliente.close()
```

Protocolos

Cuando los primeros servidores empezaron a comunicarse por red, ¿cómo sabían si los bits que recibían representaban una palabra, un número, una orden… o simplemente ruido?

O … ¿Cómo sabían los programadores como enviar una instrucción o mensaje en especifico?



Protocolos

ANTES:
[01001000 01101111 01101100 01100001]

¿"Hola"? ¿4 bytes de una estructura binaria?

PROBLEMA:
- ¿Dónde empieza y termina el mensaje?
- ¿Qué representa cada parte?
- ¿Y si alguien manda basura?

SOLUCIÓN:
Crear un PROTOCOLO: estructura fija, reglas claras, orden.

Protocolos

ANTES:
[01001000 01101111 01101100 01100001]

¿"Hola"? ¿4 bytes de una estructura binaria?

PROBLEMA:
- ¿Dónde empieza y termina el mensaje?
- ¿Qué representa cada parte?
- ¿Y si alguien manda basura?
- No existían formatos universales como JSON o XML.
- Cada sistema inventaba su propio protocolo propietario.

SOLUCIÓN:
Crear un PROTOCOLO: estructura fija, reglas claras, orden.

Protocolos

Un **protocolo** es un conjunto de **reglas formales y estructuras** que definen cómo dos sistemas se comunican correctamente entre sí.

¿Qué pasa exactamente cuando escribes una URL como **http://www.example.com** y presionas Enter?

Protocolos

HTTP (HyperText Transfer Protocol) es el protocolo que permite que un cliente (como un navegador) **pida recursos** (como páginas, imágenes, o datos) a un servidor.

Cliente

GET /index.html HTTP/1.1
Host: www.example.com

- **GET** es el método (acción).
- **/index.html** es el recurso.
- **HTTP/1.1** es la versión del protocolo.
- **Host** indica a qué servidor estamos hablando.

Servidor

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

```
<html>
  <body>Hola mundo</body>
</html>
```

Protocolos

HTTP (HyperText Transfer Protocol) es el protocolo que permite que un cliente (como un navegador) **pida recursos** (como páginas, imágenes, o datos) a un servidor.

Cliente

GET /index.html HTTP/1.1
Host: www.example.com

- **GET** es el método (acción).
- **/index.html** es el recurso.
- **HTTP/1.1** es la versión del protocolo.
- **Host** indica a qué servidor estamos hablando.

Servidor

HTTP/1.1 200 OK
Content-Type: text/html
Content-Length: 137

```
<html>
  <body>Hola mundo</body>
</html>
```

Cliente - Servidor

Variantes Arquitectonicas

Thin client

- Toda la lógica en el servidor.
- Cliente solo presenta datos (ej. navegador simple).

Fat client

- Parte de la lógica vive en el cliente.
- Requiere mayor sincronización con el servidor.

Multi-servidor

- Diversos servicios especializados: autenticación, recursos, datos.

Cliente - Servidor

¿Sigue siendo relevante?

Sí, pero ha **evolucionado** en múltiples direcciones:

- Backend RESTful que sirve múltiples frontends (SPA, apps móviles).
- Arquitecturas de 3 capas o n-capas (web → API Gateway → backend).
- Combinado con eventos o colas de mensajes para desacoplar componentes.

Componentes del protocolo HTTP

Metodo: Indica la acción: GET, POST, PUT, DELETE, etc.

Ruta/URI: Especifica el recurso solicitado.

Cabeceras: Añaden metadatos (tipo de contenido, codificación, autenticación…).

Cuerpo: Contiene datos que se envían (opcional).

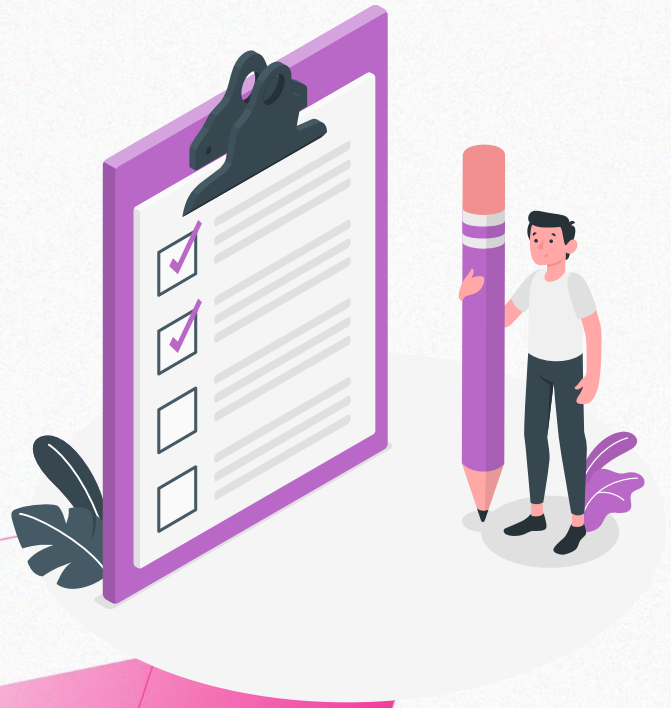
Código de estado: El servidor responde con un código que indica el resultado (200 OK, 404 Not Found, 500 Internal Server Error, etc).

Ejemplo Servidor HTTP

```
const express = require('express');
const app = express();

// Middleware para interpretar JSON en el cuerpo
app.use(express.json());

// Ruta tipo GET
app.get('/', (req, res) => {
  res.send('Hola desde el servidor Express');
});

// Ruta tipo POST
app.post('/saludo', (req, res) => {
  const nombre = req.body.nombre;
  res.send(`Hola, ${nombre}`);
});

// Iniciar servidor
app.listen(3000, () => {
  console.log('Servidor escuchando en http://localhost:3000');
});
```

Rest / Restful

REST no es un protocolo, ni una tecnología, sino un **conjunto de principios** propuestos por Roy Fielding en su tesis doctoral (2000).



Principios de REST

1. **Cliente-servidor:** Separación de responsabilidades: el cliente consume, el servidor provee.
2. **Sin estado (Stateless):** Cada solicitud HTTP contiene toda la información necesaria. El servidor **no guarda contexto** entre llamadas.
3. **Caché:** Las respuestas deben indicar si pueden ser almacenadas o no.
4. **Interfaz uniforme** *(el más importante):* Se accede a los recursos a través de URLs y **métodos estándar de HTTP**.
5. **Sistema en capas:** Puede haber proxies, balanceadores, etc. ocultos al cliente.
6. **Código bajo demanda (opcional):** Posibilidad de enviar scripts ejecutables (rara vez se usa).

REST trabaja con recursos y colecciones

¿Qué es un recurso?

Un **recurso** es cualquier entidad **identificable** dentro del sistema.

GET /usuarios/42 Esto representa **el usuario con ID 42**.

¿Qué es una colección?

Una **colección** es un conjunto de recursos del mismo tipo.

GET /usuarios Todos los usuarios
POST /usuarios Crear un nuevo usuario

REST trabaja con recursos y colecciones

¿Qué es un recurso?

Un **recurso** es cualquier entidad **identificable** dentro del sistema.

¿Qué es una colección?

Una **colección** es un conjunto de recursos del mismo tipo.

Acción	Verbo HTTP	URL	Resultado
Listar usuarios	GET	/usuarios	Devuelve la colección
Ver un usuario	GET	/usuarios/42	Devuelve un recurso
Crear usuario	POST	/usuarios	Añade un nuevo recurso
Actualizar completo	PUT	/usuarios/42	Reemplaza todo
Actualizar parcial	PATCH	/usuarios/42	Modifica campos
Eliminar usuario	DELETE	/usuarios/42	Elimina recurso

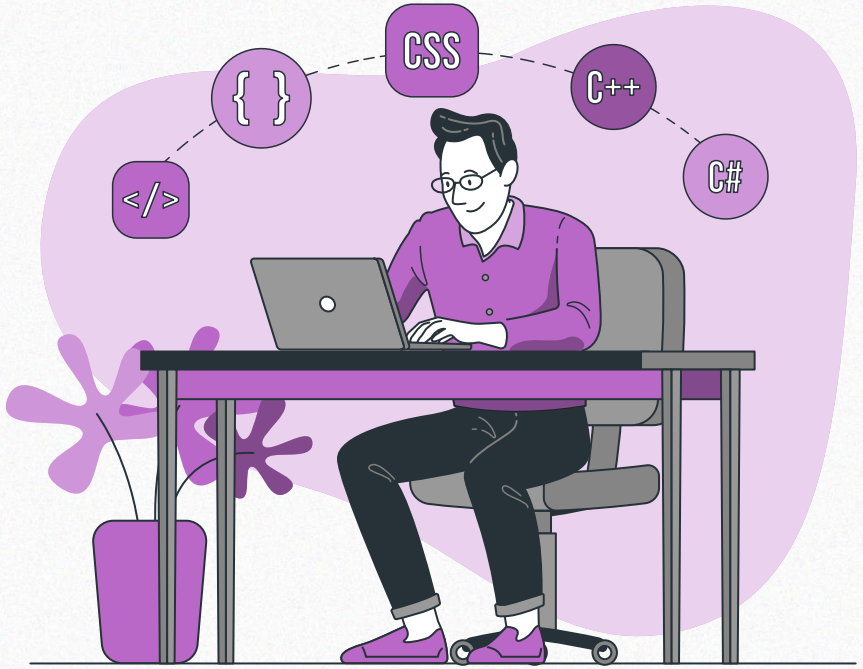
¿Por qué recursos y no funciones?

En REST **no se invocan funciones** tipo getUser().

En su lugar, **se accede a representaciones de recursos** usando las rutas.

Esto hace que las APIs sean más:

- **Predecibles**
- **Cacheables**
- **Compatibles con la Web**



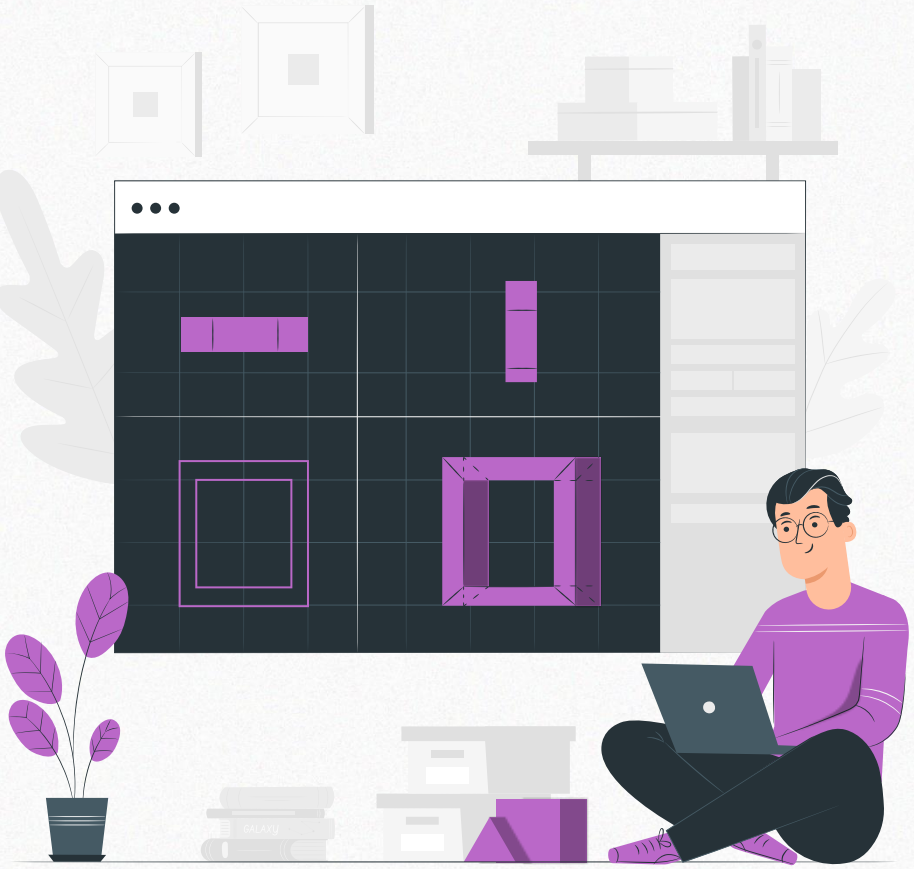
Representaciones

REST se llama así porque devuelve una **“Representación”** del recurso.

Puede ser en formato:

- application/json
- text/html
- application/xml

```
{
  "id": 42,
  "nombre": "Joel",
  "rol": "Administrador"
}
```



Restful

REST es como una dieta saludable; RESTful es seguirla bien: con disciplina y sin trampas.

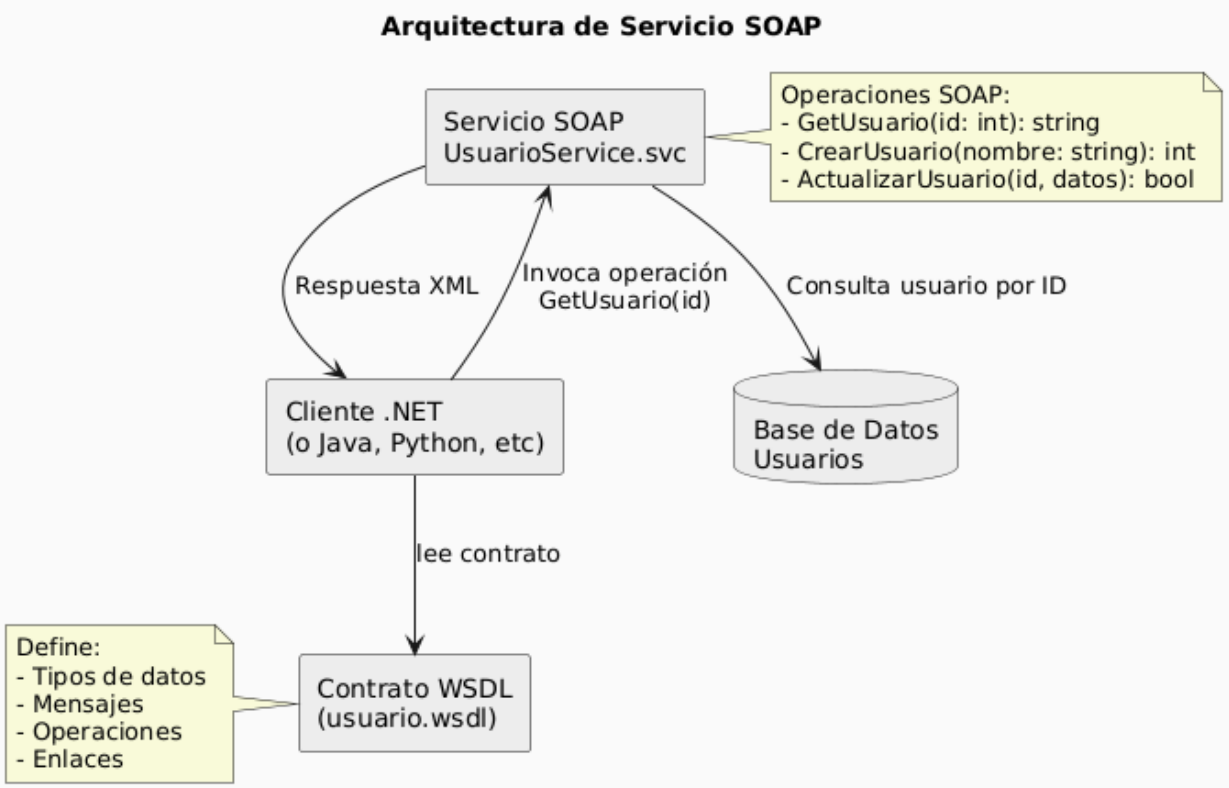
¿REST será *la única* forma correcta de permitir la comunicación cliente-servidor en el mundo moderno?

¿Hay casos donde REST no alcanza… y necesitamos algo más o diferente?

SOAP

Simple Object Access Protocol es un **protocolo de mensajería** basado en **XML**, diseñado para permitir la comunicación entre sistemas distribuidos a través de redes, especialmente en contextos **empresariales y heterogéneos**.

- Formato definido (XML)
- Estructura fija (envelope, header, body)
- Contratos (WSDL)
- Reglas de validación



WSDL

WSDL (Web Services Description Language) es un **documento XML** que describe **de manera formal y estandarizada** cómo interactuar con un servicio web SOAP.

- Establece un **acuerdo estricto** entre el **cliente y el servidor**.
- Si ambos respetan ese contrato, pueden comunicarse sin ambigüedad, incluso si están escritos en distintos lenguajes (Java, .NET, Python).
- Es como un “manual técnico” automático, generado desde el servicio y **consumido por herramientas** que crean clientes automáticamente.

```xml
<!-- Tipos de datos -->
<types>
  <xsd:schema targetNamespace="http://ejemplo.com/usuario">
    <xsd:element name="GetUsuarioRequest">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="id" type="xsd:int"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>

    <xsd:element name="GetUsuarioResponse">
      <xsd:complexType>
        <xsd:sequence>
          <xsd:element name="nombre" type="xsd:string"/>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</types>

<!-- Mensajes -->
<message name="GetUsuarioInput">
  <part name="parameters" element="tns:GetUsuarioRequest"/>
</message>
<message name="GetUsuarioOutput">
  <part name="parameters" element="tns:GetUsuarioResponse"/>
</message>

<!-- PortType: define las operaciones -->
<portType name="UsuarioPortType">
  <operation name="GetUsuario">
    <input message="tns:GetUsuarioInput"/>
    <output message="tns:GetUsuarioOutput"/>
  </operation>
</portType>

<!-- Binding: cómo se hace la llamada -->
<binding name="UsuarioBinding" type="tns:UsuarioPortType">
  <soap:binding style="document"
                transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="GetUsuario">
    <soap:operation soapAction="http://ejemplo.com/usuario/GetUsuario"/>
    <input>
```

¿Dónde se usa SOAP?

Servicios de **banca, seguros, gobierno**, donde se necesita:

- Precisión formal
- Firmas digitales
- Esquemas bien definidos
- Compatibilidad entre lenguajes (Java ↔ .NET)

¿Cómo se define un servicio SOAP?

Mediante un archivo **WSDL** que describe:

- Operaciones disponibles
- Tipos de datos
- Estructura de entrada/salida
- Puertos y ubicaciones

```
<definitions name="UsuarioService" ...>
  <message name="getUsuarioRequest">
    <part name="id" type="xsd:int"/>
  </message>
  <message name="getUsuarioResponse">
    <part name="nombre" type="xsd:string"/>
  </message>
  ...
</definitions>
```

REST:

Dame el recurso *usuario 42*.
GET /usuarios/42

SOAP:

Ejecuta la operación *getUsuario(id=42)*

(dentro del SOAP Body, no en la URL)

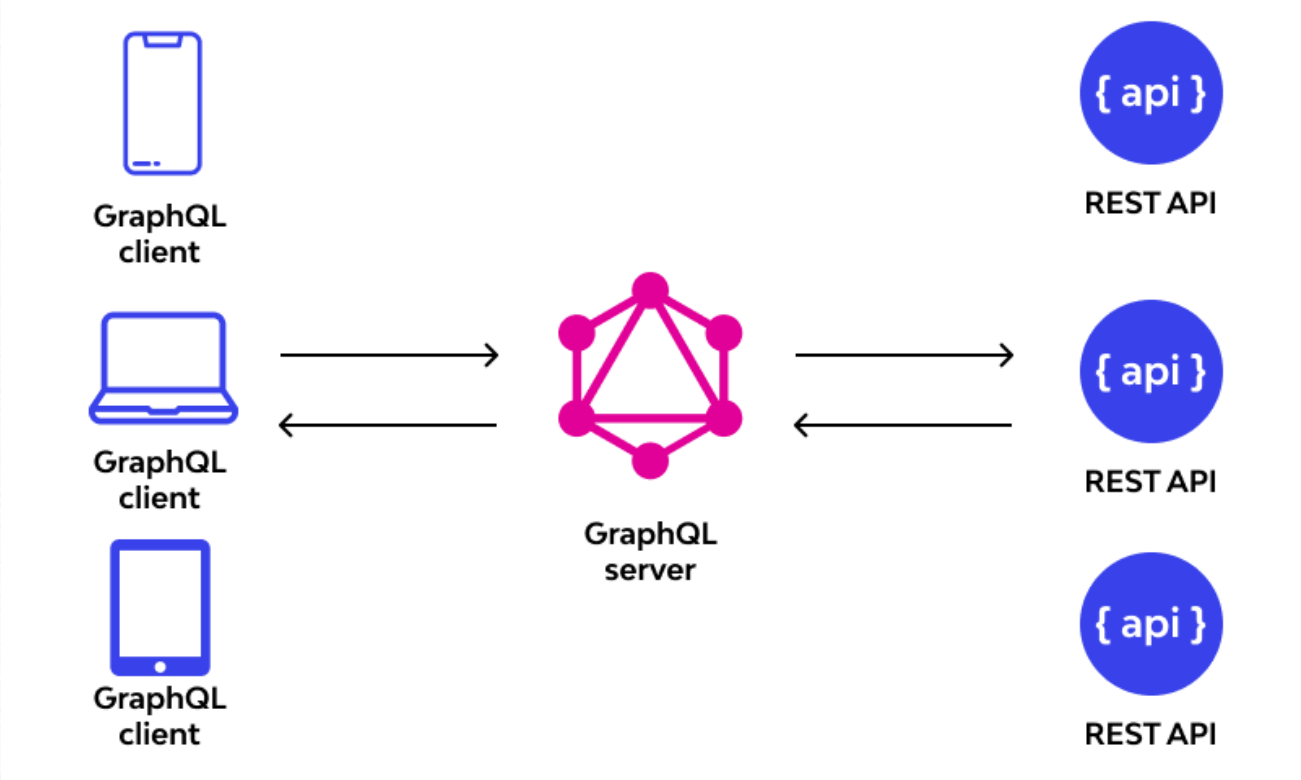
```
var cliente = new UsuarioServiceRef.UsuarioServiceClient();
var resultado = cliente.GetUsuario(42);
Console.WriteLine("Respuesta del servidor: " + resultado);
```

¿Qué pasa si una app móvil solo necesita el nombre del usuario… pero nuestra API REST devuelve 30 campos?

GraphQL

Es un lenguaje de consulta para APIs y un entorno de ejecución del lado del servidor, que permite a los clientes **definir exactamente qué datos necesitan**.

```
query {
  usuario(id: 42) {
    nombre
    email
    direccion {
      ciudad
    }
  }
}
```



GraphQL

Es un lenguaje de consulta para APIs y un entorno de ejecución del lado del servidor, que permite a los clientes **definir exactamente qué datos necesitan**.

Componente	Función
Schema	Define los tipos, relaciones y operaciones disponibles
Query	Consultas de lectura
Mutation	Operaciones que modifican datos
Resolver	Función que ejecuta la lógica detrás de cada campo

```
type Usuario {
  id: ID
  nombre: String
  email: String
  direccion: Direccion
}

type Direccion {
  ciudad: String
  pais: String
}

type Query {
  usuario(id: ID!): Usuario
}

type Mutation {
  crearUsuario(nombre: String, email: String): Usuario
}
```

GraphQL sobre APIs

El servidor GraphQL consulta una API REST existente

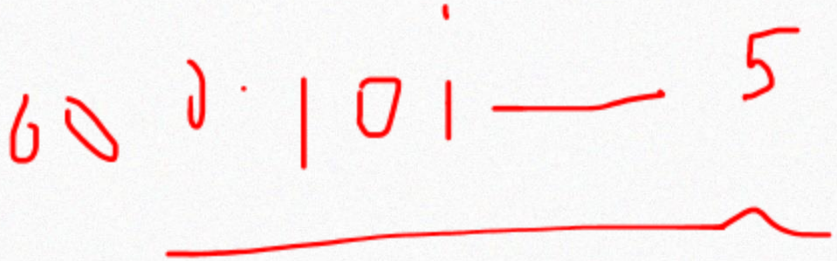
```javascript
// servidor.js
const { ApolloServer, gql } = require('apollo-server');
const fetch = require('node-fetch');

const typeDefs = gql`
  type Usuario {
    id: ID
    nombre: String
  }

  type Query {
    usuario(id: ID!): Usuario
  }
`;

const resolvers = {
  Query: {
    usuario: async (_, { id }) => {
      const res = await fetch(`https://api.ejemplo.com/usuarios/${id}`);
      const data = await res.json();
      return {
        id: data.id,
        nombre: data.nombre
      };
    }
  }
};

const server = new ApolloServer({ typeDefs, resolvers });
server.listen().then(({ url }) => {
  console.log(`Servidor GraphQL en ${url}`);
});
```

GraphQL sobre Modelos o DB

El servidor GraphQL accede directamente a los modelos o DB

```javascript
// usuarioModel.js
const usuariosDB = [
  { id: '42', nombre: 'Joel' },
  { id: '1', nombre: 'Andrea' }
];

function buscarUsuario(id) {
  return usuariosDB.find((u) => u.id === id);
}

module.exports = { buscarUsuario };
```

```javascript
// servidor.js
const { ApolloServer, gql } = require('apollo-server');
const { buscarUsuario } = require('./usuarioModel');

const typeDefs = gql`
  type Usuario {
    id: ID
    nombre: String
  }

  type Query {
    usuario(id: ID!): Usuario
  }
`;

const resolvers = {
  Query: {
    usuario: (_, { id }) => buscarUsuario(id)
  }
};
```

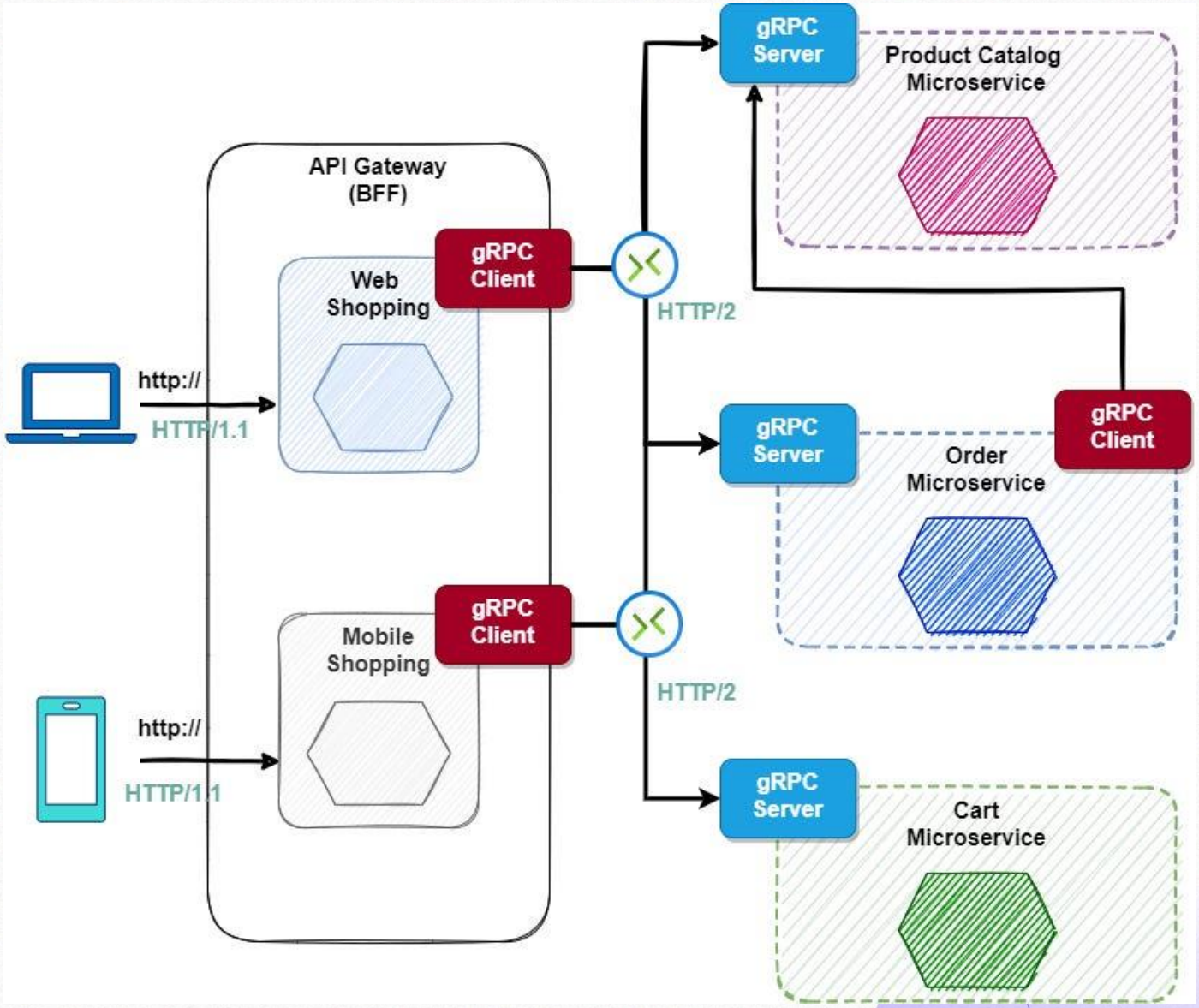


¿Qué pasaría si una empresa con cientos de microservicios necesitara que todos se comuniquen millones de veces al día… y rápido?

¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

- HTTP/2 (soporte nativo para streams, multiplexing)
- **Protocol Buffers (Protobuf)** como formato de serialización
- Código fuertemente tipado generado automáticamente



¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder

```
// usuario.proto
syntax = "proto3";

service UsuarioService {
  rpc GetUsuario (UsuarioRequest) returns (UsuarioResponse);
}

message UsuarioRequest {
  int32 id = 1;
}

message UsuarioResponse {
  string nombre = 1;
}
```

¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder

```js
// servidor.js
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

const paqueteDef = protoLoader.loadSync('usuario.proto');
const grpcObj = grpc.loadPackageDefinition(paqueteDef);
const servidor = new grpc.Server();

servidor.addService(grpcObj.UsuarioService.service, {
  GetUsuario: (call, callback) => {
    const id = call.request.id;
    const nombre = id === 42 ? "Joel" : "Desconocido";
    callback(null, { nombre });
  }
});

servidor.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createIns
  servidor.start();
});
```

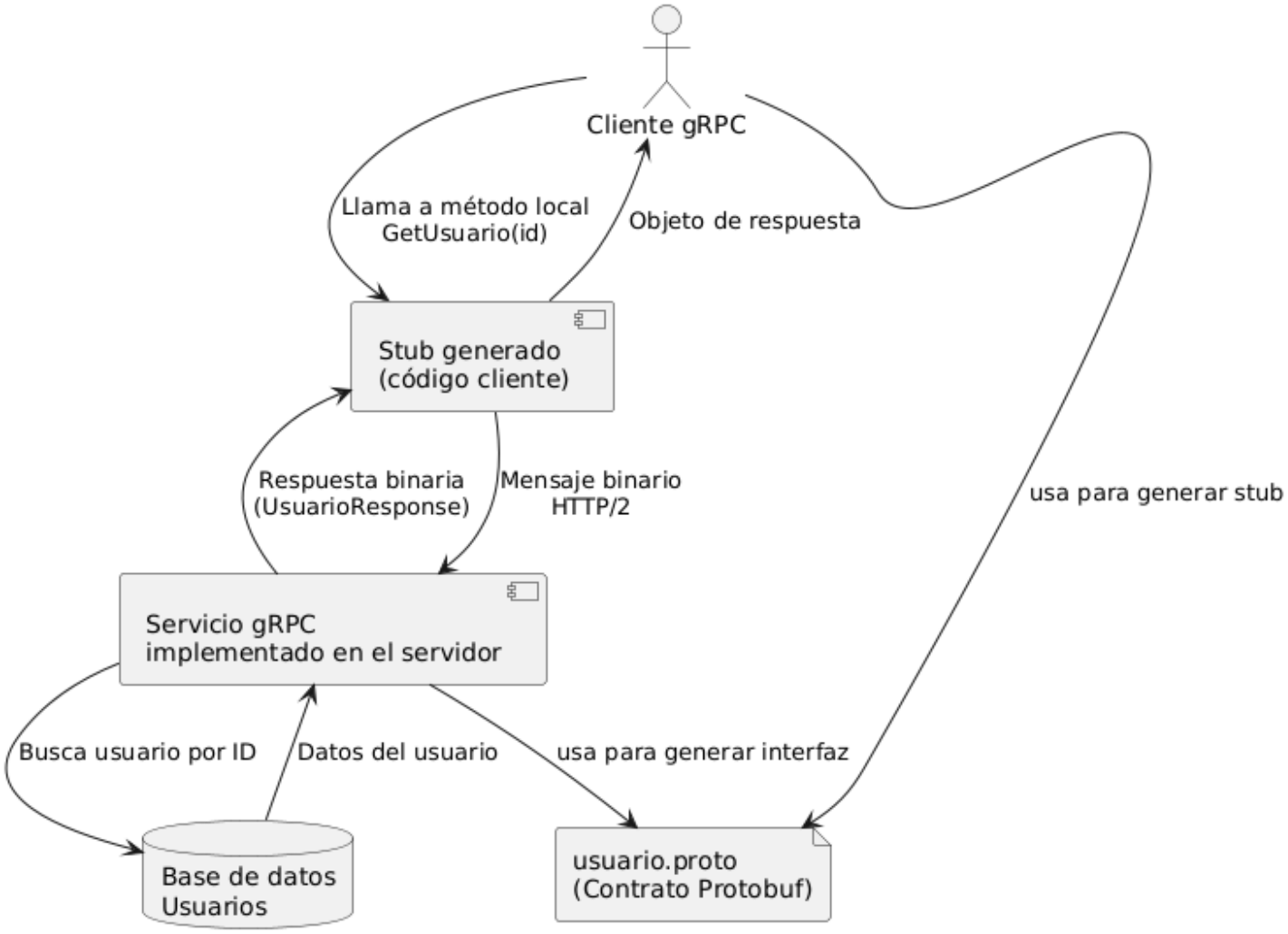
¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder

```
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

const paqueteDef = protoLoader.loadSync('usuario.proto');
const grpcObj = grpc.loadPackageDefinition(paqueteDef);

const cliente = new grpcObj.UsuarioService('localhost:50051',

cliente.GetUsuario({ id: 42 }, (err, respuesta) => {
  console.log("Nombre:", respuesta.nombre);
});
```

¿Qué es gRPC?

gRPC es un sistema de **llamadas a procedimientos remotos** (RPC) moderno y de alto rendimiento desarrollado por Google, basado en:

Componente	Descripción
.proto	Archivo donde defines mensajes y servicios
protoc	Compilador que genera código cliente/servidor
Stub	Clase generada que representa el cliente
Service	Clase implementada en el servidor para responder