

Tipos de arquitectura

¿Qué son las arquitecturas de software ?

Las arquitecturas de *software* son el diseño o estructura organizativa fundamental de un sistema de *software*.

Incluyen la disposición de sus componentes, las relaciones entre ellos y los principios que guían su desarrollo y evolución a lo largo del tiempo.

En esencia, representan una visión de alto nivel del sistema que busca garantizar que el software sea eficiente, mantenible, escalable y adecuado para sus objetivos.



Componentes clave de la arquitectura de software:

Componentes o módulos	Conexiones o interfaces	Estilos arquitectónicos
Partes funcionales independientes del sistema, como servicios, bibliotecas o clases.	Formas en que los componentes interactúan entre sí, como API, protocolos o mensajes.	Patrones o enfoques establecidos para resolver problemas comunes, como arquitectura en capas, microservicios o orientada a eventos.



Arquitecturas de software

- **Arquitectura monolítica:**

Todas las funcionalidades de una aplicación se construyen y despliegan como una sola unidad. Pueden tener llamadas REST como parte integral de su software.

- **Arquitectura de microservicios:**

Esta arquitectura divide una aplicación en servicios pequeños e independientes, cada uno enfocado en una funcionalidad específica del negocio. En líneas generales, suelen tener varias llamadas REST.

- **Arquitectura de capas (N-Tier):**

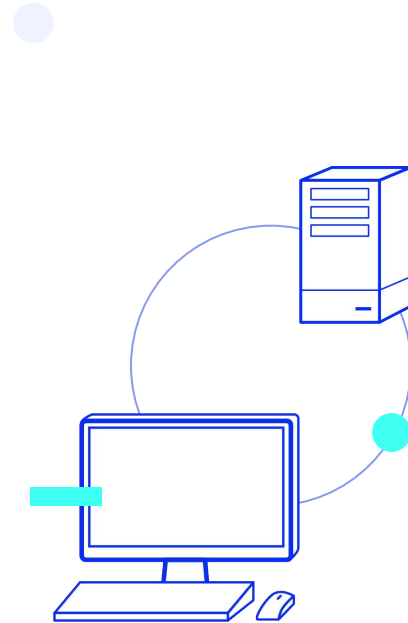
Organiza la aplicación en capas lógicas separadas, como presentación, lógica de negocio y acceso a datos. Pueden tener llamadas REST como parte integral de su *software*.

- **Arquitectura orientada a servicios (SOA):**

Similar a microservicios, SOA organiza la funcionalidad en servicios independientes, pero típicamente más grandes y menos especializados que los microservicios. En líneas generales, suelen tener varias llamadas REST.

- **Arquitectura *Event-Driven* (basada en eventos):**

La comunicación entre componentes se realiza a través de eventos. Los componentes emiten eventos y reaccionan a eventos de otros componentes. Si bien la idea de esta arquitectura es llamar a otro servicio de manera asincrónica, a través de una cola de mensajería, potencialmente pueden tener llamadas REST como parte integral de su *software*.



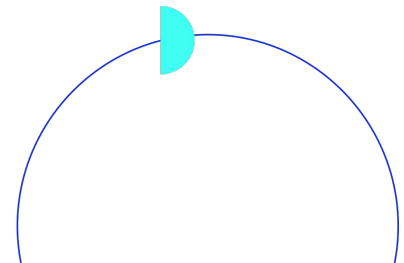
Arquitectura monolítica

La arquitectura monolítica es un tipo de diseño de *software* en el que todas las funcionalidades del sistema están integradas en una única aplicación o unidad.

Esto significa que el código, la lógica de negocio, la interfaz de usuario y el acceso a datos se encuentran juntos en un único proyecto o despliegue.

Características principales:

- Todo el sistema se desarrolla, implementa y ejecuta como una sola pieza.
- Los módulos del sistema están fuertemente acoplados, lo que implica que dependen unos de otros.



Ventajas

- **Simplicidad:** fácil de desarrollar y probar en proyectos pequeños.
- **Menor complejidad inicial:** no requiere configuraciones complicadas ni herramientas especializadas.
- **Despliegue sencillo:** se despliega todo en un solo servidor o entorno.



Desventajas

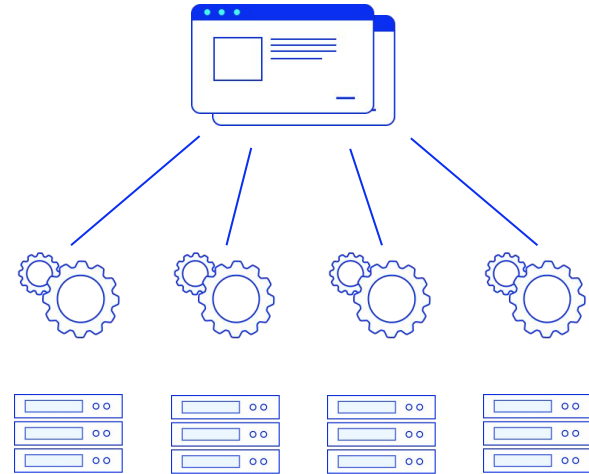
- **Escalabilidad limitada:** dificultad para escalar módulos de forma independiente.
- **Mantenimiento complicado:** a medida que crece, los cambios en una parte pueden afectar a todo el sistema.
- **Falta de flexibilidad:** dificultad para adoptar nuevas tecnologías en módulos específicos.



Arquitectura de microservicios

Los microservicios son una arquitectura de *software* en la que una aplicación se construye como un conjunto de servicios pequeños e independientes que se comunican entre sí a través de API bien definidas.

Cada microservicio se centra en una funcionalidad específica del negocio y puede ser desarrollado, desplegado y escalado de forma independiente.



Características principales:

- **Independencia:** cada microservicio es un componente autónomo que se puede desarrollar, desplegar y escalar de manera independiente de los demás servicios.
- **Especialización:** cada microservicio está diseñado para realizar una única tarea o conjunto de tareas relacionadas.
- **Comunicación a través de API:** los microservicios se comunican entre sí mediante API, generalmente usando protocolos HTTP/REST, aunque también pueden usar otros métodos como gRPC o colas de mensajería.
- **Flexibilidad tecnológica:** diferentes microservicios pueden estar escritos en diferentes lenguajes de programación y usar distintas tecnologías de almacenamiento y herramientas.
- **Despliegue independiente:** el despliegue de un microservicio no afecta a los demás servicios.



Ventajas

- **Escalabilidad:** permite escalar solo los componentes que requieren más recursos.
- **Resiliencia:** los fallos en un microservicio no necesariamente afectan a toda la aplicación.
- **Desarrollo ágil:** facilita el desarrollo continuo e integración continua (CI/CD), permitiendo que diferentes



Desventajas

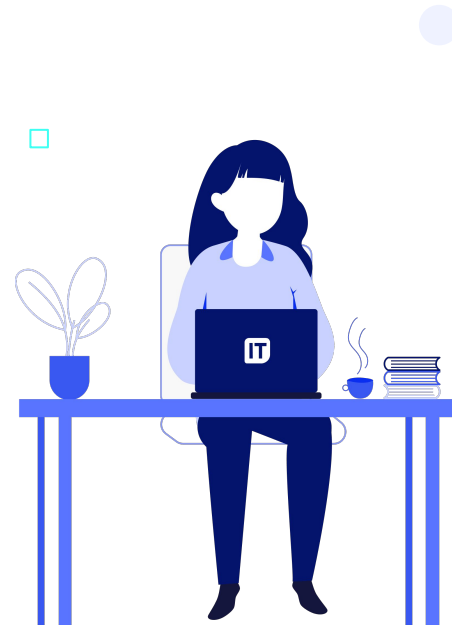
- **Complejidad:** la gestión de múltiples servicios independientes puede aumentar la complejidad del sistema.
- **Comunicación:** requiere un manejo eficiente de la comunicación y coordinación entre servicios.
- **Despliegue:** la orquestación y despliegue de múltiples servicios pueden ser más complicados.



Arquitectura de capas (N-Tier)

La arquitectura de capas (N-Tier) es un estilo de diseño de software en el que las funcionalidades del sistema se organizan en capas separadas, cada una con responsabilidades específicas y comunicándose únicamente con las capas adyacentes.

Esto crea una estructura modular y facilita la gestión y el mantenimiento del software.



Ventajas

- **Modularidad:** cada capa tiene responsabilidades claras y separadas.
- **Facilidad de mantenimiento:** cambios en una capa no afectan directamente a las demás.
- **Escalabilidad:** permite escalar partes específicas del sistema según sea necesario.
- **Reutilización:** las capas pueden ser reutilizadas en otros sistemas.



Desventajas

- **Mayor complejidad inicial:** puede requerir más esfuerzo para diseñar e implementar.
- **Latencia:** la comunicación entre capas puede agregar retrasos al sistema.
- **Sobrecarga:** en sistemas pequeños, la separación en capas puede ser innecesaria.



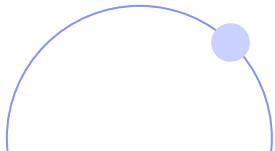
Arquitectura orientada a servicios (SOA)

La arquitectura orientada a servicios (SOA) es un enfoque de diseño de software en el que las funcionalidades del sistema se dividen en servicios independientes que se comunican entre sí mediante protocolos estándar (como HTTP o mensajes XML/JSON).

Cada servicio está diseñado para realizar una función específica y puede ser reutilizado en diferentes aplicaciones o contextos.

Componentes claves

- **Servicio:** una unidad funcional independiente que realiza una tarea específica.
- **Bus de servicios (ESB):** facilita la comunicación entre servicios y asegura que estén desacoplados.
- **Orquestación:** define cómo interactúan los servicios para cumplir procesos más complejos.



Ventajas

- **Flexibilidad:** es fácil agregar, modificar o reemplazar servicios sin afectar a todo el sistema.
- **Escalabilidad:** los servicios pueden escalarse de manera independiente según la demanda.
- **Reutilización:** permite usar los mismos servicios en diferentes aplicaciones o proyectos.
- **Integración:** ideal para conectar sistemas heterogéneos.



Desventajas

- **Complejidad:** la implementación requiere herramientas adicionales, como un bus de servicios (ESB) y monitoreo avanzado.
- **Sobrecarga:** la comunicación entre servicios puede introducir latencia.
- **Costos iniciales:** diseñar y desplegar una arquitectura SOA puede ser costoso y llevar tiempo.



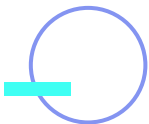
Arquitectura Event-Driven (basada en eventos)

La arquitectura Event-Driven (basada en eventos) es un estilo de diseño de software donde los sistemas están organizados en torno a la producción, detección y reacción a eventos.

Un evento es cualquier cambio significativo en el estado del sistema, como una acción del usuario, un mensaje recibido o una actualización de datos.

Características principales

- **Desacoplamiento:** los productores y consumidores no necesitan conocerse entre sí directamente.
- **Asincronía:** los eventos son procesados de forma independiente, lo que mejora la escalabilidad y la resiliencia.
- **Reactividad:** el sistema reacciona en tiempo real a los cambios o acciones.



Ventajas

- **Escalabilidad:** permite manejar grandes volúmenes de eventos distribuyendo la carga.
- **Resiliencia:** el sistema puede seguir funcionando aunque algunos consumidores estén temporalmente inactivos.
- **Flexibilidad:** es fácil agregar nuevos consumidores sin afectar el productor o viceversa.



Desventajas

- **Complejidad:** requiere herramientas y una gestión cuidadosa de los eventos, como evitar duplicaciones.
- **Depuración difícil:** consiste en identificar errores, puede ser complicado debido a la asincronía.
- **Latencia:** la comunicación asincrónica puede introducir retrasos en el procesamiento.



Conclusión

En la comunicación entre aplicaciones, REST se ha convertido en un estándar ampliamente adoptado debido a su simplicidad, flexibilidad y eficacia.

Independientemente de la arquitectura de software elegida, ya sea monolítica, de microservicios o basada en eventos, REST proporciona un medio robusto y coherente para la interacción entre sistemas.

