# .NET, Patrones de Arquitectura y Principios SOLID

# Lenguajes

- C#
- F#
- VB

# Implementaciones

- .NET Framework
- .NET Core
- Xamarin/Mono (mobile)
- UWP (Universal Windows Platform)

# Herramientas

- Visual Studio (Community/Enterprise/Professional)
- Visual Studio Code
- Visual Studio for Mac
- CLI

# Patrones de Arquitectura

- Son soluciones o estructuras que nos ayudan a definir la aplicación desde el nivel más grande.
- ¿Para qué sirven?

# Patrón de Arquitectura en Capas

También conocida como arquitectura multi capas o N-Capas

# Especificaciones de la arquitectura en capas

- Organizada en capas horizontales
- Cada capa tiene su propósito
- Capas de alto nivel y capas de bajo nivel
- Reglas Generales:
  - La capa de alto nivel tiene visibilidad de la capa inferior no viceversa
  - Cada capa tiene su abstracción
  - Si están bien diseñadas una capa puede ser reemplazada por otra
  - Los componentes en cada capa debe hacer tareas relacionada a esa capa

# Patrón de Arquitectura de Microservicios

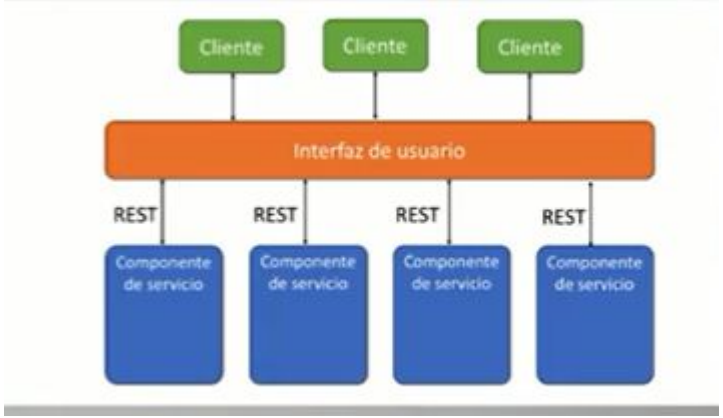
¿Qué son aplicaciones monolíticas?

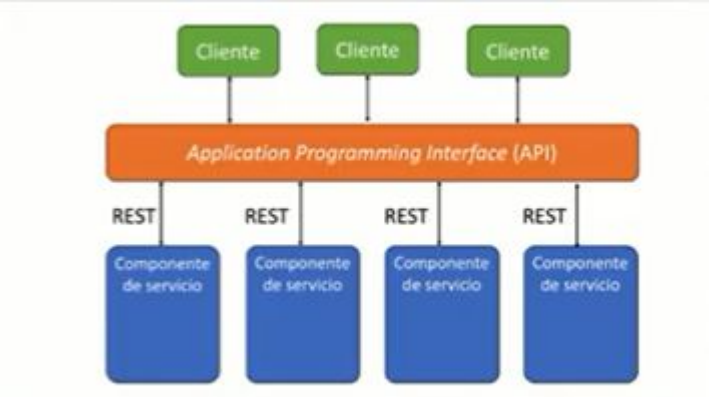
La arquitectura de microservicios habla de desarrollar tu aplicación como un conjunto de servicios más pequeños

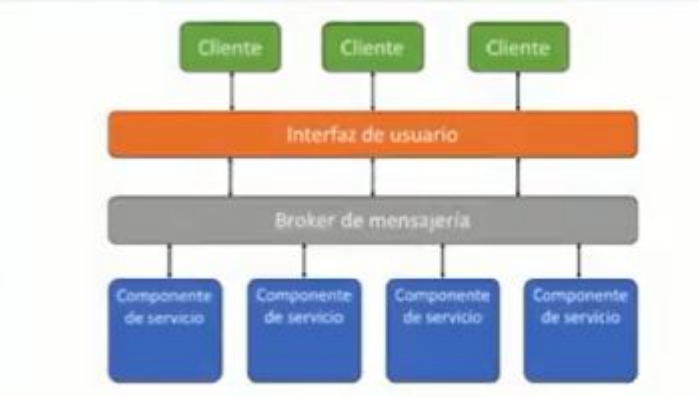
Características

- Cada microservicio corre su propio servicio;
- Se comunican por mecanismos muy livianos;
- Se desarrollan alrededor de la necesidad del negocio;
- Deben estar soportadas por un proceso de despliegue automático.

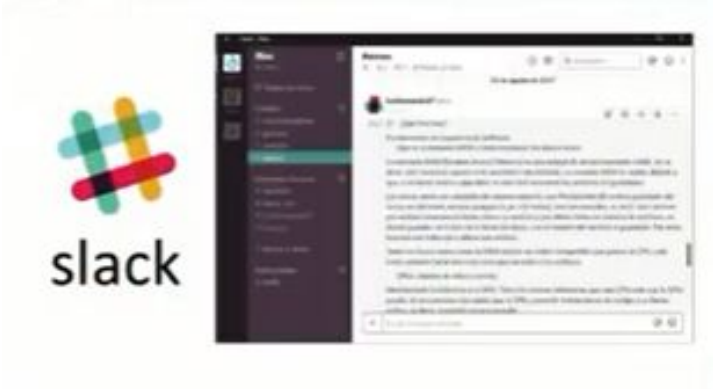
# Topologías


Cliente
Cliente
Cliente
Interfaz de usuario
REST
REST
REST
REST
Componente de servicio
Componente de servicio
Componente de servicio
Componente de servicio


Cliente
Cliente
Cliente
Application Programming Interface (API)
REST
REST
REST
REST
Componente de servicio
Componente de servicio
Componente de servicio
Componente de servicio

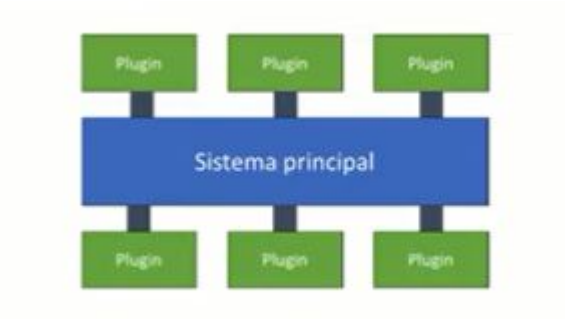

Cliente
Cliente
Cliente
Interfaz de usuario
Broker de mensajería
Componente de servicio
Componente de servicio
Componente de servicio
Componente de servicio

# Patrón de Arquitectura de Microkernel/Plugins



# Características

- Funcionalidad principal/básica (lógica de dominio independiente y acceso a los recursos globales del sistema )
- Pluggins (reglas de negocio especializadas)



# Decisiones de diseño

Registro:

- Poniendo pluggins en una carpeta
- modificando archivos de configuración

La interacción

- Por eventos
- Invocando los objetos
- Rest/Soap llamando servidores diferentes

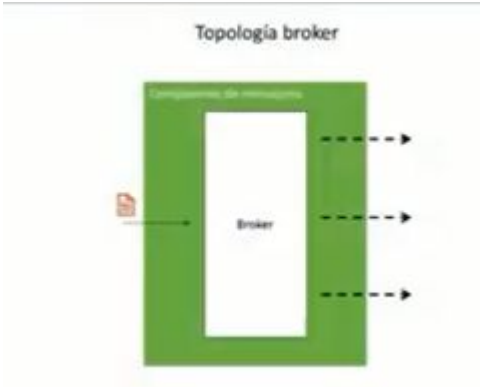
# Patrón de Arquitectura Orientada a Eventos

# Componentes

- Generadores/productores
- Mensajes/Eventos
- Componente de Mensajería
- Canales
  - Cola de Mensajes
  - Patrón publicador/suscriptor
- Consumidores/procesadores

# Topologías


Topología mediador
Mediador
Cola de eventos


Topología broker
Broker

# Arquitectura de 3 capas

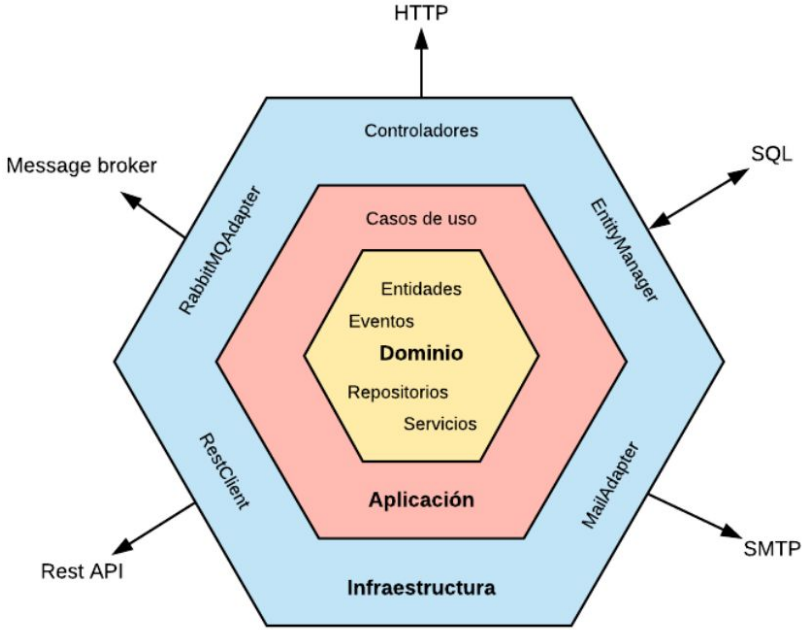
Presentación: maneja interacción entre cliente (humano o programa externo) y la aplicación.

- Entrega y solicita información
- Interpreta solicitudes del usuario en acciones para las capas de persistencia y negocio

Dominio: lógica de negocio del sistema, tareas o trabajo para las cuales el sistema está hecho. Esconde el acceso a datos.

Acceso a datos: se comunica con la base de datos u otros sistemas de persistencia de datos.

# Arquitectura Hexagonal/Onion



# Que patron usar: Atributos de Calidad

- Escalabilidad
- Desplegabilidad
- Rendimiento
- Agilidad
- Testeabilidad
- Facilidad de Desarrollo

# Escalabilidad

+ Microservicios y Orientadas a Eventos
- Microkernel y Por Capas

# Desplegabilidad

+ MicroKernel
+ Relativo: Microservicios y Orientada a Eventos
- Capas

# Rendimiento

+ Orientada a Eventos
+ Relativo: microkernel y por capas
- Microservicios

# Agilidad

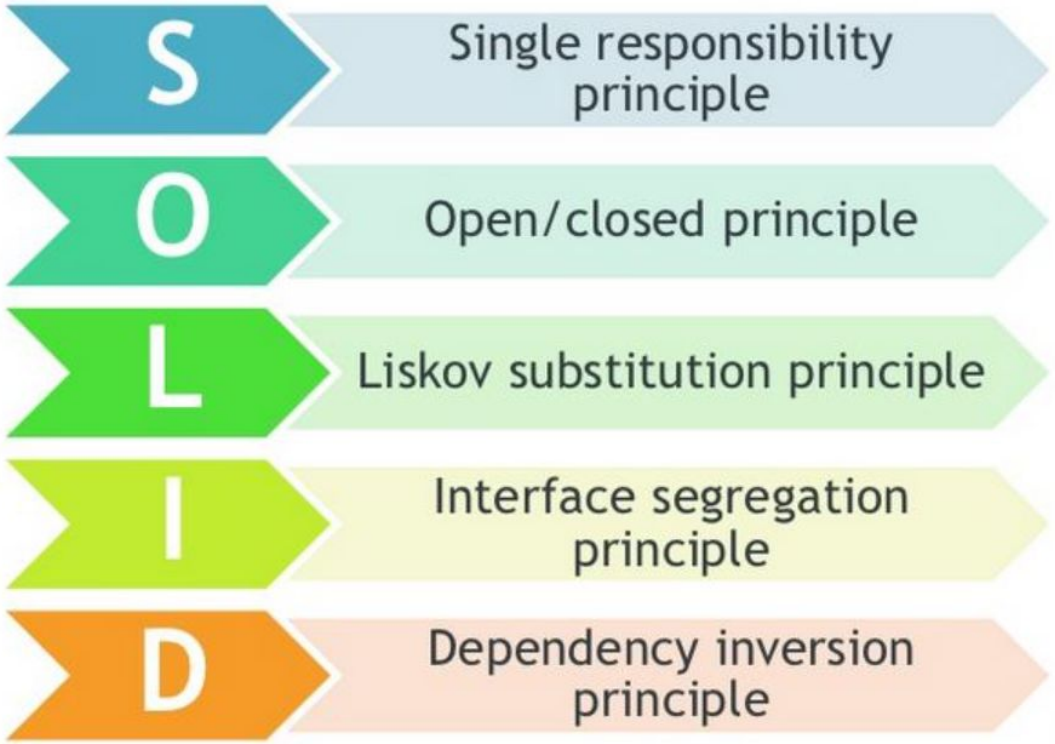
+ Microkernel, eventos y microservicios
- Capas

# Testeabilidad

+ Capas, Microkernel y Microservicios
- Orientadas a eventos

# Facilidad de Desarrollo

+ Capas
+ Relativo: Microservicios
- Microkernel y orientada a eventos


S
Single responsibility principle
O
Open/closed principle
L
Liskov substitution principle
I
Interface segregation principle
D
Dependency inversion principle

# Principio de Responsabilidad Unica

***“A class should have one, and only one, reason to change.”***

***Ejemplo: navaja china, una clase Cliente encargada de mandar un Mail.***

# Principio Abierto/Cerrado

***“You should be able to extend a classes behavior, without modifying it.”***

***Ejemplo: Crear una clase abstracta que permita ser heredada y extendida***

# Principio de Sustitución de Liskov

***“Derived classes must be substitutable for their base classes.”***

***Ejemplo: dejar de usar una clase derivada y todo debería funcionar con su clase padre.***

# Principio de Segregación de la Interfaz

***“Make fine grained interfaces that are client specific.”***

***Ejemplo: mantener las interfaces específicas para cada cliente. No forzar a definir elementos que no necesitan.***

# Principio de Inversión de Dependencias

***“Depend on abstractions, not on concretions.”***

***Ejemplo: Un auto posee frenos, motor, pedales, palanca.***