

# **Introducción a los servicios web en C++**

Antonio Espín Herranz

# Contenidos

- Fundamentos de microservicios
  - Ventajas frente a arquitecturas monolíticas
  - Desafíos específicos al trabajar con lenguajes de bajo nivel como C/C++.
  - Casos de uso en sistemas embebidos, sistemas de alto rendimiento, y telecomunicaciones.
- Comparación con otros lenguajes de programación en arquitecturas de microservicios:
  - Cuando elegir C/C++ frente a lenguajes como Java, Python o Go.

# Los servicios web

- Los servicios web permiten que aplicaciones se comuniquen entre sí a través de la web, usando protocolos como HTTP y formatos como XML o JSON. Los más comunes son:
- **RESTful services:** Basados en HTTP, usan métodos como GET, POST, PUT, DELETE.
- **SOAP services:** Más estructurados, usan XML y requieren WSDL para definir la interfaz.

# Arquitectura Monolítica vs Microservicios

Característica	Monolítica	Microservicios
Estructura	Una sola unidad de software	Conjunto de servicios independientes
Comunicación interna	Directa, dentro del mismo código	A través de APIs o protocolos como HTTP/REST
Despliegue	Se despliega todo el sistema a la vez	Cada servicio se puede desplegar por separado

# Ventajas

- **Monolítica:**

- Fácil de desarrollar y probar en etapas iniciales.
- Menor complejidad en la gestión de dependencias.
- Ideal para proyectos pequeños.

- **Microservicios:**

- Escalabilidad granular: puedes escalar solo lo que necesitas.
- Flexibilidad tecnológica: cada servicio puede usar su propio stack.
- Mayor resiliencia: fallos en un servicio no afectan a todo el sistema

# Desventajas

- **Monolítica:**

- Difícil de escalar parcialmente.
- Actualizaciones lentas y riesgosas: cualquier cambio requiere recompilar todo.
- Poca flexibilidad para adoptar nuevas tecnologías.

- **Microservicios:**

- Mayor complejidad inicial: requiere planificación y diseño cuidadoso.
- Gestión más compleja de la comunicación entre servicios.
- Necesidad de herramientas para monitoreo, orquestación y despliegue continuo.

# Arquitectura de microservicios

- La arquitectura de microservicios consiste en dividir una aplicación en múltiples servicios pequeños, independientes y especializados, cada uno encargado de una funcionalidad específica.
- Estos servicios se comunican entre sí a través de APIs, generalmente usando HTTP y formatos como JSON o XML2.
- Características clave:
  - **Desacoplamiento:** Cada microservicio puede desarrollarse, desplegarse y escalarse por separado.
  - **Especialización:** Cada servicio cumple una función concreta (por ejemplo, autenticación, pagos, notificaciones).
  - **Independencia tecnológica:** Puedes usar diferentes lenguajes o bases de datos en cada microservicio.
  - **Escalabilidad:** Puedes escalar solo los servicios que lo necesiten, no toda la aplicación.
  - **Resiliencia:** Si un servicio falla, los demás pueden seguir funcionando.

# Ventajas

- **Desarrollo ágil:** Equipos pequeños pueden trabajar en paralelo.
- **Entrega continua:** Puedes actualizar servicios sin afectar al resto.
- **Mejor mantenimiento:** Más fácil localizar y corregir errores.
- **Escalabilidad granular:** Solo escalas lo que realmente necesita más recursos.



# Servicios en C++

- Aunque C++ no es el lenguaje más popular para desarrollo web, tiene ventajas únicas:
  - **Rendimiento superior:** Ideal para aplicaciones que requieren alta velocidad y bajo consumo de recursos.
  - **Control detallado de memoria y recursos:** Fundamental en sistemas embebidos o aplicaciones críticas.
  - **Interoperabilidad con sistemas existentes:** Muchos sistemas legacy (sistemas heredados, antiguos) están escritos en C++.

# Fundamentos de Microservicios

- Los **microservicios** son una **arquitectura** de software que divide una aplicación en **módulos independientes**, cada uno con una funcionalidad específica, que se comunican entre sí mediante APIs.
- Esto permite:
  - **Desarrollo paralelo** por equipos distintos.
  - **Despliegue independiente** de cada servicio.
  - **Escalabilidad granular**.
  - **Resiliencia** ante fallos parciales.

# Desafíos al usar C++ en Microservicios

- C++ ofrece rendimiento y control excepcionales, trabajar con ellos en una arquitectura distribuida presenta los siguientes retos:
  - Concurrencia y sincronización
  - Comunicación entre servicios
  - Serialización de datos
  - Testing y mantenimiento
  - Despliegue y contenedores

# Concurrencia

- **C++ no tiene un modelo de concurrencia tan seguro como otros lenguajes modernos.**
- Requiere **gestión manual de hilos**, mutexes y semáforos, lo que puede provocar condiciones de carrera si no se maneja bien.
- La librería **Boost** nos proporciona APIs enfocadas a la concurrencia.

# Testing y mantenimiento

- Las pruebas unitarias y de integración son más complejas.
- Herramientas como Google Test ayudan, pero no son tan integradas como en otros ecosistemas.

# Despliegue y Contenedores

- Posibilidad de utilización de contenedores en **Docker**:
  - Creación de imágenes, contenedores, redes y volúmenes
- **Kubernetes**:
  - Escalado y replicación de los contenedores.

# Casos de uso

- **Sistemas embebidos**

- **Microservicios en el borde (edge computing):** sensores, controladores, IoT.
  - Ejemplo: un sistema de control de tráfico donde cada microservicio gestiona un semáforo.

- **Sistemas de alto rendimiento**

- **Trading financiero, simulación científica, renderizado gráfico.**
- Ejemplo: un microservicio que calcula precios de derivados financieros en tiempo real.

- **Telecomunicaciones**

- **Procesamiento de paquetes, gestión de redes, protocolos de señalización.**
- Ejemplo: microservicios que manejan la señalización SIP en una red VoIP.

# Algunas herramientas en C++

- **CppREST SDK (Casablanca)**
  - Biblioteca moderna para crear clientes y servicios REST
  - ***Actualmente no es una buena elección para nuevos proyectos porque está en revisión.***
- **Pistache:** para sistemas Linux. ***No se aconseja para Windows, suele dar problemas.***
- **Boost.Beast / Boost.Asio:** Manejar HTTP y TCP
- **crow:** Servicios REST



# Comunicación entre Servicios

- La comunicación entre microservicios puede ser síncrona / asíncrona.
- Tenemos varios métodos de comunicación entre microservicios:
  - **HTTP/REST**
    - La librería más famosa: **Boost.beast** para la implementación de clientes y servidores HTTP
  - **gRPC:**
    - Basado en HTTP/2 y protocols buffers
    - Eficiente y rápida
    - Soporta **streaming** bidireccional, comunicación síncrona y asíncrona
  - **Mensajería asíncrona:**
    - Utiliza RabbitMQ o ZeroMQ (soporta los dos tipos)
  - **Sockets TCP/UDP:**
    - Comunicación de bajo nivel entre procesos
    - Útil si se requiere alto rendimiento y control total.
    - Se puede utilizar **Boost.Asio** para facilitar la implementación.

# Comunicación entre Servicios

- Consideraciones:
  - **Síncrona vs asíncrona:** HTTP/gRPC → síncrono (gRPC soporta ambas formas de comunicación) y RabbitMQ para asíncrona.
  - **Desacoplamiento:** Los sistemas de mensajería permiten independencia entre servicios.
  - **Escalabilidad:** Los brókers (por ejemplo: RabbitMQ) de mensajes facilitan la escalabilidad.
  - **Latencia:** gRPC y sockets ofrecen menor latencia que HTTP.
- Un buen esquema de comunicación puede ser gRPC para servicios internos y HTTP para APIs públicas.








# Serialización de datos

- No existe una solución estándar como `JSON.stringify` en JavaScript.
- Se puede realizar mediante librerías externas como son:  
**nlohmann-json**
- Instalable con el gestor de paquetes **vcpkg**

# Cuando elegir C++

- **Máximo rendimiento**
  - Video juegos multijugador
  - Comercio financiero
- Integración con sistemas **Legacy** (sistemas heredados), siguen dentro de la organización a pesar de estar obsoletas.
  - Características de un sistema **Legacy**:
    - Tecnología obsoleta
    - Difícil de modificar
    - Poca documentación
    - Dependencia crítica: no se pueden retirar
      - Por ejemplo, COBOL

# Comparativa con otros lenguajes

Característica	C++	Rust	Go	Python
 Rendimiento	Muy alto	Muy alto	Alto	Medio
 Concurrencia	Compleja (manual)	Avanzada y segura	Sencilla (goroutines)	Limitada (GIL)
 Facilidad de desarrollo	Baja (complejo y verboso)	Media (sintaxis exigente)	Alta (simple y directa)	Muy alta (ideal para prototipos)
 Ecosistema para microservicios	Limitado (Boost, REST SDK)	Emergente (Axum, Tokio)	Muy sólido (Gin, gRPC)	Amplio (FastAPI, Flask)
 Testing y mantenimiento	Complejo	Seguro pero exigente	Sencillo	Muy sencillo
 Contenerización	Requiere configuración manual	Mejor soporte que C++	Muy fácil	Muy fácil
 Seguridad	Manual	Muy fuerte (sin null, sin data races)	Buena	Media
 Interoperabilidad	Excelente con sistemas legacy	Buena con C/C++	Media	Alta

# C++, Java, Go, Python

- **C++**

- Necesitas **máximo rendimiento** y control del hardware.
- Trabajas en **sistemas embebidos, telecomunicaciones, juegos, o procesamiento en tiempo real.**
- Tu aplicación requiere **uso intensivo de memoria**, CPU o GPU.

- **Evítalo si:**

- Buscas rapidez en el desarrollo o facilidad de mantenimiento.
- No tienes experiencia en gestión manual de recursos.

# C++, Java, Go, Python

- **Java**

- Quieres construir **aplicaciones empresariales, backend robusto, o sistemas distribuidos**.
- Necesitas **portabilidad y compatibilidad multiplataforma**.
- Buscas un ecosistema maduro con herramientas como **Spring Boot, Kafka**, etc.

- **Evítalo si:**

- El rendimiento extremo es crítico (aunque Java es bastante rápido).
- Algo más ligero para microservicios simples.

# C++, Java, Go, Python

- **Go**

- Estás desarrollando **microservicios, infraestructura cloud, o DevOps tools**.
- Necesitas **conurrencia sencilla** (goroutines) y **despliegue rápido**.
- Si buscas un lenguaje moderno, minimalista y eficiente.

- **Evítalo si:**

- Necesitas programación orientada a objetos avanzada.
- Equipo está más familiarizado con otros ecosistemas.



# C++, Java, Go, Python

- **Python:**

- Quieres desarrollar **prototipos rápidos, APIs REST, o servicios de IA / Machine Learning.**
- Buscas **simplicidad, legibilidad** y una curva de aprendizaje baja.
- Necesitas acceso a librerías como **TensorFlow, Pandas, FastAPI**, etc.

- **Evítalo si:**

- El rendimiento es crítico (aunque puedes usar extensiones en C/C++ o Rust).
- Trabajas en sistemas con recursos limitados o tiempo real.

# Comparativa

Proyecto	Lenguaje ideal(es)
Sistema embebido	C++
Backend empresarial	Java
Microservicios en la nube	Go, Java
Prototipo de IA	Python
Procesamiento en tiempo real	C++, Rust
Herramientas de infraestructura	Go
Aplicaciones multiplataforma	Java