

MicroInformatico 2

Documento de Arquitectura de Software

El presente documento presenta una visión general de la arquitectura de software aplicado en la construcción de la aplicación web MicroInformatico 2, el mismo que sigue un enfoque estructurado para el diseño de aplicaciones web modernas que sean escalables, resistentes y con una alta disponibilidad. Se basa en prácticas probadas y utilizadas en compañías líderes en la industria de software actual (Microsoft, Netflix, Amazon, Facebook, etc).

Elaborado por

Giancarlo Villena

Christian Cabrera

**Introducción**

La nube está cambiando la forma en que se diseñan las aplicaciones. En lugar de ser monolitos, las aplicaciones se descomponen en servicios menores y descentralizados. Estos servicios se comunican a través de API o mediante el uso de eventos o de mensajería asincrónica. Las aplicaciones se escalan horizontalmente, agregando nuevas instancias, tal y como exigen las necesidades.

Estas tendencias agregan nuevos desafíos. El estado de las aplicaciones se distribuye. Las operaciones se realizan en paralelo y de forma asincrónica. Las aplicaciones deben ser resistentes cuando se produzcan errores. Las implementaciones deben estar automatizadas y ser predecibles. La supervisión y la telemetría son fundamentales para obtener una visión general del sistema. MicroInformatico 2 se orienta a cubrir estas nuevas necesidades y retos de forma progresiva y ágil.

|  |  |
| --- | --- |
| **ENFOQUE TRADICIONAL** | **ENFOQUE NUBE** |
| * Monolítica * Diseñada para una escalabilidad predecible * Base de datos relacional * Procesamiento sincronizado * Diseño para evitar errores (MTBF) * Actualizaciones grandes ocasionales * Administración manual * Servidores de copo de nieve | * Descompuesto * Diseñado para un escalado elástico * Persistencia de Polyglot (combinación de tecnologías de almacenamiento) * Procesamiento asincrónico * Diseño para errores (MTTR) * Pequeñas actualizaciones frecuentes * Administración automatizada * Infraestructura inmutable |

**Sobre los estilos de arquitectura de software**

Un estilo de arquitectura de software es una familia de arquitecturas que comparten determinadas características. Por ejemplo, de ***n niveles*** es un estilo de arquitectura común. Últimamente, las arquitecturas de ***microservicios*** se han empezado a hacer más populares. Los estilos de arquitectura no requieren el uso de tecnologías concretas, pero algunas tecnologías son adecuadas para ciertas arquitecturas. Por ejemplo, los contenedores son una opción natural para los microservicios.

Hemos identificado un conjunto de estilos de arquitectura que se suelen encontrar en aplicaciones en la nube.

**N niveles**

N niveles es una arquitectura tradicional para aplicaciones empresariales. Las dependencias se administran mediante la división de la aplicación en capas que realizan funciones lógicas como presentaciones, lógica de negocios y acceso a datos. Una capa solo puede llamar a las capas que se encuentran por debajo de ella. Sin embargo, este sistema de capas horizontales puede tener otros efectos. Puede ser difícil introducir cambios en una parte de la aplicación sin tocar el resto de esta. Eso hace que las actualizaciones frecuentes sean un problema que limita la frecuencia con la que se agregan nuevas características.

La arquitectura de n niveles es una opción habitual para migrar aplicaciones existentes que ya usan una arquitectura en capas. Por ese motivo, la arquitectura de n niveles es la que se encuentra más frecuentemente en soluciones de infraestructura como servicio (IaaS) o en aplicaciones que usan una combinación de IaaS y servicios administrados.

**Web-Cola-Trabajo**

Para una solución puramente PaaS, considere la posibilidad de usar una arquitectura Web-Cola-Trabajo . En este estilo, la aplicación tiene un front-end web que controla las solicitudes HTTP y un trabajo de back-end que realiza tareas de uso intensivo de la CPU u operaciones de larga duración. El front-end se comunica con el trabajo a través de una cola de mensajes asincrónicos.

La arquitectura Web-Cola-Trabajo es adecuada para dominios relativamente sencillos con algunas tareas que consumen muchos recursos. Al igual que en la de n niveles, la arquitectura es fácil de entender. El uso de servicios administrados simplifica la implementación y las operaciones. Pero con dominios complejos, puede que sea difícil administrar las dependencias. El front-end y el trabajo se pueden convertir fácilmente en componentes grandes y monolíticos que son difíciles de mantener y actualizar. Al igual que sucede con la de n niveles, esto puede reducir la frecuencia de las actualizaciones y limitar la innovación.

**Microservicios**

Si la aplicación tiene un dominio más complejo, considere la posibilidad de pasar a una arquitectura de Microservicios. Una aplicación de microservicios se compone de muchos servicios pequeños e independientes. Cada servicio implementa una sola función empresarial. Los servicios están acoplados de forma flexible y se comunican a través de contratos de API.

Un pequeño equipo de desarrollo con dedicación puede compilar cada servicio. Los servicios individuales se pueden implementar sin necesidad de mucha coordinación entre los equipos, lo cual fomenta unas actualizaciones frecuentes. Una arquitectura de microservicios es más compleja a la hora de compilar y administrar que la de n niveles o la de web-cola-trabajo. Requiere un desarrollo perfeccionado y una cultura de DevOps. Pero si se hace correctamente, este estilo puede dar lugar una velocidad de lanzamiento mayor, una innovación más rápida y una arquitectura más resistente.

**Arquitectura basada en eventos**

Las arquitecturas basadas en eventos usan un modelo de publicación-suscripción (pub-sub), en el que los productores publican eventos y los consumidores de suscriben a ellos. Los productores son independientes de los consumidores y estos, a su vez, son independientes entre sí.

Considere la posibilidad de implementar una arquitectura basada en eventos para las aplicaciones que ingieren y procesan un gran volumen de datos con una latencia muy baja como en el caso de las soluciones de IoT. Este estilo también es útil cuando diferentes subsistemas deben realizar distintos tipos de procesamiento en los mismos datos de evento.

**Big Data, Big Compute**

Big Data y Big Compute son los estilos de arquitectura especializados en cargas de trabajo que resultan más adecuados para determinados perfiles específicos. Big Data permite dividir un conjunto de datos muy grande en fragmentos, realizando un procesamiento paralelo en todo el conjunto, con fines de análisis y creación de informes. Big compute, también denominada informática de alto rendimiento (HPC), realiza cálculos en paralelo en un gran número (miles) de núcleos. Los dominios incluyen simulaciones, modelado y representaciones 3-D.

**Sobre los estilos de arquitectura de software como restricciones**

Un estilo de arquitectura puede crear restricciones en el diseño, incluido en el conjunto de elementos que pueden aparecer, y en las relaciones permitidas entre los elementos. Las restricciones contribuyen a dar "forma" a una arquitectura restringiendo el universo de opciones. Cuando una arquitectura cumple con las restricciones de un estilo determinado, surgen determinadas propiedades deseables.

Por ejemplo, entre las restricciones de los microservicios se incluyen:

* Un servicio representa una única responsabilidad.
* Cada servicio es independiente de los demás.
* Los datos son privados para el servicio al que pertenecen. Los servicios no comparten datos.

Mediante la adhesión a estas restricciones, lo que surge es un sistema en el que los servicios se pueden implementar independientemente, los errores se aíslan, las actualizaciones frecuentes son posibles y resulta fácil introducir nuevas tecnologías en la aplicación.

Antes de elegir un estilo de arquitectura, es importante asegurarse de que exista una comprensión en los principios subyacentes y las restricciones del estilo elegido. En caso contrario, puede acabar con un diseño que se adapta al estilo en un nivel superficial pero que no utiliza todas las posibilidades de ese estilo. También es importante ser práctico. A veces es mejor ser menos exigentes con una restricción en lugar de incidir en la pureza de una arquitectura.

En la tabla siguiente se resume cómo administra las dependencias cada estilo y qué tipos de dominios son los más adecuados para cada uno.

|  |  |  |
| --- | --- | --- |
| **Estilo de arquitectura** | **Administración de dependencias** | **Tipo de dominio** |
| N niveles | Niveles horizontales divididos por subred | Dominio empresarial tradicional. La frecuencia de las actualizaciones es baja. |
| Web-Cola-Trabajo | Trabajos de front-end y back-end, desacoplados mediante mensajería asincrónica. | Dominio relativamente sencillo con algunas tareas de uso intensivo de recursos. |
| Microservicios | Servicios descompuestos verticalmente (funcionalmente) que se llaman mutuamente mediante API. | Dominio complicado. Actualizaciones frecuentes. |
| Arquitectura basada en eventos. | Productores y consumidores. Vista independiente por cada subsistema. | IoT y sistemas en tiempo real |
| Macrodatos | Divide un conjunto de datos grande en fragmentos pequeños. Procesamiento en paralelo en los conjuntos de datos locales. | Análisis de datos por lotes y en tiempo real. Análisis predictivo mediante Machine Learning. |
| Big Compute | Asignación de datos a miles de núcleos. | Dominios con procesos intensivos como simulaciones. |

**Sobre el proceso de análisis y selección de arquitectura**

A continuación, se listan algunos de los criterios y razones que se tuvieron en cuenta en la selecciona de la arquitectura para el sistema MicroInformatico 2.

***Complejidad***

¿Está justificada la complejidad de la arquitectura para el dominio? O, por el contrario, ¿es demasiado simple? En ese caso, se arriesga a terminar con un sistema "sin arquitectura alguna observable", ya que esta no le ayuda a administrar las dependencias correctamente.

***Mensajería asincrónica y coherencia final***

La mensajería asincrónica puede usarse para desacoplar servicios y aumentar la confiabilidad (porque se pueden recuperar los mensajes) y la escalabilidad. Sin embargo, esto también crea desafíos en el control de la coherencia final, así como la posibilidad de que se produzcan mensajes duplicados.

***Comunicación entre servicios***

Cuando descompone una aplicación en servicios independientes, existe el riesgo de que la comunicación entre estos provoque una latencia inaceptable o cree una congestión en la red (por ejemplo, en una arquitectura de microservicios).

***Manejabilidad***

¿Es difícil administrar la aplicación y supervisar e implementar actualizaciones, etc?

**Sobre el estilo de arquitectura microservicios**

Los microservicios son pequeños e independientes, y están acoplados de forma imprecisa. Un único equipo reducido de programadores puede escribir y mantener un servicio.

Cada servicio es un código base independiente, que puede administrarse por un equipo de desarrollo pequeño.

Los servicios pueden implementarse de manera independiente. Un equipo puede actualizar un servicio existente sin tener que volver a generar e implementar toda la aplicación.

Los servicios son los responsables de conservar sus propios datos o estado externo. Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.

Los servicios se comunican entre sí mediante API bien definidas. Los detalles de la implementación interna de cada servicio se ocultan frente a otros servicios.

No es necesario que los servicios compartan la misma pila de tecnología, las bibliotecas o los marcos de trabajo.

**Sobre las ventajas del estilo de arquitectura microservicios**

***Agilidad.*** Dado que microservicios se implementan de forma independiente, resulta más fácil de administrar las correcciones de errores y las versiones de características. Puede actualizar un servicio sin volver a implementar toda la aplicación y revertir una actualización si algo va mal. En muchas aplicaciones tradicionales, un error en una parte de la aplicación puede bloquear todo el proceso de lanzamiento. Es posible que se requieran nuevas características a la espera de que se integre, pruebe y publique una corrección de errores.

***Equipos pequeños y centrados.*** Un microservicio debe ser lo suficientemente pequeño como para que un solo equipo de características lo pueda compilar, probar e implementar. Los equipos pequeños favorecen la agilidad. Los equipos grandes suelen ser menos productivos, porque la comunicación es más lenta, aumenta la sobrecarga de administración y la agilidad disminuye.

***Base de código pequeña.*** En las aplicaciones monolíticas, con el paso del tiempo se da la tendencia de que las dependencias del código se acaben enredarse, por lo que para agregar una nueva característica, es preciso tocar el código en muchos lugares. Al no compartir el código ni los almacenes de datos, la arquitectura de microservicios minimiza las dependencias y resulta más fácil agregar nuevas características.

***Mezcla de tecnologías.*** Los equipos pueden elegir la tecnología que mejor se adapte al servicio de una combinación de pilas de tecnología, según corresponda.

***Aislamiento de errores.*** Si un microservicio individual no está disponible, no interrumpe toda la aplicación, siempre que los microservicios de nivel superior estén diseñados para controlar los errores correctamente (por ejemplo, mediante la implementación de disyuntores).

***Escalabilidad.*** Los servicios se pueden escalar de forma independiente, lo que permite escalar horizontalmente los subsistemas que requieren más recursos, sin tener que escalar horizontalmente toda la aplicación. Mediante un orquestador como Kubernetes o Service Fabric se puede empaquetar una mayor densidad de servicios en un solo host, lo que aumenta la eficacia en el uso de los recursos.

***Aislamiento de los datos.*** Al verse afectado solo un microservicio, es mucho más fácil realizar actualizaciones del esquema. En una aplicación monolítica, las actualizaciones del esquema pueden ser muy complicadas, ya que las distintas partes de la aplicación pueden tocar los mismos datos, por lo que realizar modificaciones en el esquema resulta peligroso.

**Sobre los desafíos del estilo de arquitectura microservicios**

Las ventajas de los microservicios tienen un "precio". Estos son algunos de los aspectos que deben tenerse en cuenta antes de embarcarse en una arquitectura de microservicios.

***Complejidad.*** Una aplicación de microservicios tiene más partes en movimiento que la aplicación monolítica equivalente. Cada servicio es más sencillo, pero el sistema como un todo es más complejo.

***Desarrollo y pruebas.*** La escritura de un servicio pequeño que utilice otros servicios dependientes requiere un enfoque que no sea escribir una aplicación tradicional monolítica o en capas. Las herramientas existentes no siempre están diseñadas para trabajar con dependencias de servicios. La refactorización en los límites del servicio puede resultar difícil. También supone un desafío probar las dependencias de los servicios, especialmente cuando la aplicación está evolucionando rápidamente.

***Falta de gobernanza.*** El enfoque descentralizado para la generación de microservicios tiene ventajas, pero también puede causar problemas. Puede acabar con tantos lenguajes y marcos de trabajo diferentes que la aplicación puede ser difícil de mantener. Puede resultar útil establecer algunos estándares para todo el proyecto sin restringir excesivamente la flexibilidad de los equipos. Esto se aplica especialmente a las funcionalidades transversales, como el registro.

***Congestión y latencia de red.*** El uso de muchos servicios pequeños y detallados puede dar lugar a más comunicación interservicios. Además, si la cadena de dependencias del servicio se hace demasiado larga (el servicio A llama a B, que llama a C...), la latencia adicional puede constituir un problema. Tendrá que diseñar las API con atención. Evite que las API se comuniquen demasiado, piense en formatos de serialización y busque lugares para utilizar patrones de comunicación asincrónica.

***Integridad de datos.*** Cada microservicio es responsable de la conservación de sus propios datos. Como consecuencia, la coherencia de los datos puede suponer un problema. Adopte una coherencia final cuando sea posible.

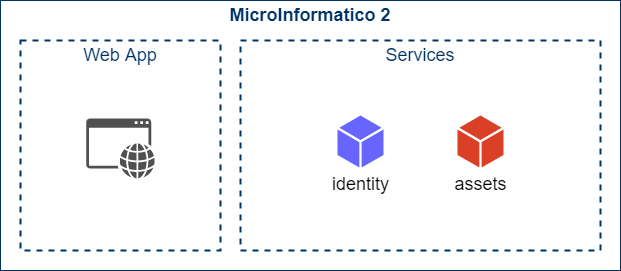
***Administración.*** Para tener éxito con los microservicios se necesita una cultura de DevOps consolidada. El registro correlacionado entre servicios puede resultar un desafío. Normalmente, el registro debe correlacionar varias llamadas de servicio para una sola operación de usuario.

***Control de versiones.*** Las actualizaciones de un servicio no deben interrumpir servicios que dependen de él. Es posible que varios servicios se actualicen en cualquier momento; por lo tanto, sin un cuidadoso diseño, podrían surgir problemas con la compatibilidad con versiones anteriores o posteriores.

***Conjunto de habilidades.*** Los microservicios son sistemas muy distribuidos. Evalúe cuidadosamente si el equipo tiene los conocimientos y la experiencia para desenvolverse correctamente.

**Sobre el estilo de arquitectura de MicroInformatico 2**

MicroInformatico 2 sigue un estilo de arquitectura de microservicios. Una arquitectura de microservicios consta de una colección de servicios autónomos y pequeños. Los servicios son independientes entre sí y cada uno debe implementar una funcionalidad de negocio individual.

*Diagrama lógico de la arquitectura de microservicios de Microinformatico*

A continuación, se listan algunas prácticas que se siguieron en la construcción del sistemas microinformatico 2 con un estilo de arquitectura de microservicios.

* Alineacion de los servicios del sistema al dominio empresarial.
* Descentralizacion de las responsabilidades, los equipos individuales son responsables de diseñar y compilar servicios.
* El almacenamiento de datos es privado para el servicio que posee los datos.
* Los servicios se comunican a través de API bien diseñadas.
* Las API deben modelar el dominio, no la implementación interna del servicio.
* Evitar el acoplamiento entre servicios. Entre las causas de acoplamiento se encuentran los protocolos de comunicación rígidos y los esquemas de bases de datos compartidos.
* Los servicios deben tener un acoplamiento flexible y una alta cohesión funcional.
* Las funciones que es probable que cambien juntas se deben empaquetar e implementar en conjunto. Si residen en distintos servicios, estos terminan estrechamente acoplados, porque un cambio en un servicio requerirá la actualización del otro servicio.
* La comunicación demasiado intensa entre dos servicios puede ser un síntoma de un acoplamiento estrecho y una cohesión baja.
* Aíslamiento de errores. Utilizacion de estrategias de resistencia para impedir que los errores dentro de un servicio se reproduzcan en cascada.