Diseño de una aplicación orientada a microservicios

28/09/2021 • Tiempo de lectura: 8 minutos •  

En este artículo

Esta sección se centra en desarrollar una hipotética aplicación empresarial del lado servidor.

Especificaciones de la aplicación

La aplicación hipotética controla las solicitudes mediante la ejecución de lógica de negocios, el acceso a bases de datos y, después, la devolución de respuestas HTML, JSON o XML. Diremos que la aplicación debe admitir varios clientes, incluidos exploradores de escritorio que ejecuten aplicaciones de página única (SPA), aplicaciones web tradicionales, aplicaciones web móviles y aplicaciones móviles nativas. También es posible que la aplicación exponga una API para el consumo de terceros. También debe ser capaz de integrar sus microservicios o aplicaciones externas de forma asincrónica, para que ese enfoque ayude a la resistencia de los microservicios en caso de errores parciales.

La aplicación constará de estos tipos de componentes:

- Componentes de presentación. Estos componentes son los responsables del control de la interfaz de usuario y el consumo de servicios remotos.
- Lógica de dominio o de negocios. Este componente es la lógica de dominio de la aplicación.
- Lógica de acceso a bases de datos. Este componente está formado por componentes de acceso a datos responsables de acceder a las bases de datos

(SQL o NoSQL).

- Lógica de integración de aplicaciones. Este componente incluye un canal de mensajería basado en agentes de mensajes.

La aplicación requerirá alta escalabilidad, además de permitir que sus subsistemas verticales se escalen horizontalmente de forma autónoma, porque algunos subsistemas requerirán mayor escalabilidad que otros.

La aplicación debe ser capaz de implementarse en varios entornos de infraestructura (varias nubes públicas y locales) y debe ser multiplataforma, capaz de cambiar con facilidad de Linux a Windows (o viceversa).

Contexto del equipo de desarrollo

También se supone lo siguiente sobre el proceso de desarrollo de la aplicación:

- Tiene varios equipos de desarrollo centrados en diferentes áreas de negocio de la aplicación.
- Los nuevos miembros del equipo deben ser productivos con rapidez y la aplicación debe ser fácil de entender y modificar.
- La aplicación tendrá una evolución a largo plazo y reglas de negocio cambiantes.
- Necesita un buen mantenimiento a largo plazo, lo que significa agilidad al implementar nuevos cambios en el futuro al tiempo que se pueden actualizar varios subsistemas con un impacto mínimo en el resto.
- Le interesa la integración y la implementación continuas de la aplicación.
- Le interesa aprovechar las ventajas de las nuevas tecnologías (plataformas, lenguajes de programación, etc.) durante la evolución de la aplicación. No quiere realizar migraciones completas de la aplicación al cambiar a las nuevas tecnologías, ya que eso podría generar costos elevados y afectar a la capacidad de predicción y la estabilidad de la aplicación.

Elección de una arquitectura

¿Cuál debe ser la arquitectura de implementación de la aplicación? Las especificaciones de la aplicación, junto con el contexto de desarrollo, sugieren que se debe diseñar descomponiéndola en subsistemas autónomos en forma de microservicios de colaboración y contenedores, donde un microservicio es un contenedor.

Con este enfoque, cada servicio (contenedor) implementa un conjunto de funciones coherentes y estrechamente relacionadas. Por ejemplo, es posible que una aplicación conste de servicios como el de catálogo, de pedidos, de cesta de la compra, perfiles de usuario, etc.

Los microservicios se comunican mediante protocolos como HTTP (REST), pero también de forma asincrónica (por ejemplo, mediante AMQP) siempre que sea posible, en especial al propagar actualizaciones con eventos de integración.

Los microservicios se desarrollan e implementan como contenedores de forma independiente entre ellos. Este enfoque implica que un equipo de desarrollo puede desarrollar e implementar un microservicio determinado sin afectar a otros subsistemas.

Cada microservicio tiene su propia base de datos, lo que permite separarlo totalmente de otros microservicios. Cuando sea necesario, la coherencia entre las bases de datos de los diferentes microservicios se logra mediante eventos de integración de nivel de aplicación (a través de un bus de eventos lógicos), como se controla en Command and Query Responsibility Segregation (CQRS). Por ese motivo, las restricciones de negocio deben adoptar la coherencia final entre los múltiples microservicios y bases de datos relacionadas.

eShopOnContainers: aplicación de referencia para .NET y microservicios implementados mediante contenedores

Para que pueda centrarse en la arquitectura y las tecnologías en lugar de pensar en un dominio de negocio hipotético que es posible que no conozca, se ha seleccionado un dominio de negocio conocido: una aplicación de comercio electrónico simplificada (e-shop) que presenta un catálogo de productos, recibe pedidos de los clientes, comprueba el inventario y realiza otras funciones de negocio. El código fuente basado en contenedores de esta aplicación está disponible en el repositorio de GitHub eShopOnContainers .

La aplicación consta de varios subsistemas, incluidos varios front-end de interfaz de usuario de tienda (una aplicación web y una aplicación móvil nativa), junto con los microservicios de back-end y los contenedores para todas las operaciones necesarias del lado servidor con varias puertas de enlace de API como puntos de entrada consolidados a los microservicios internos. En la figura 6-1 se muestra la arquitectura de la aplicación de referencia.

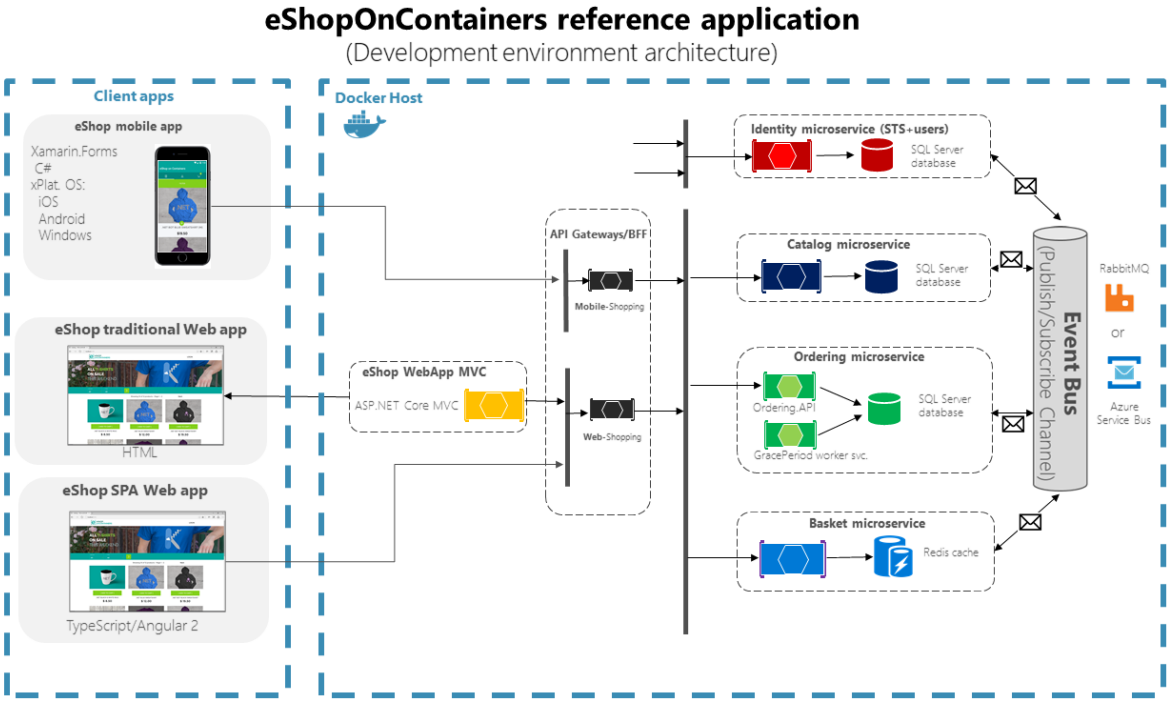

Figura 6-1. La arquitectura de aplicación de referencia de eShopOnContainers para el entorno de desarrollo

En el diagrama anterior se muestra que los clientes móviles y SPA se comunican con los puntos de conexión de puerta de enlace de API única y, a continuación, se comunican con los microservicios. Los clientes web tradicionales se comunican con el microservicio MVC, que se comunica con microservicios mediante la puerta de enlace de API.

Entorno de hospedaje. En la figura 6-1 se pueden ver varios contenedores implementados dentro de un único host de Docker. Ese sería el caso al implementar en un único host de Docker con el comando docker-compose up. Pero si se usa un clúster de orquestadores o contenedores, cada contenedor podría ejecutarse en otro host (nodo) y cualquier nodo podría ejecutar cualquier número de contenedores, como se explicó anteriormente en la sección sobre arquitectura.

Arquitectura de comunicación. En la aplicación eShopOnContainers se usan dos tipos de comunicación, según el tipo de la acción funcional (consultas frente a transacciones y actualizaciones):

- Comunicación de cliente a microservicio de HTTP a través de puertas de enlace de API. Este enfoque se sigue para las consultas y al aceptar los comandos transaccionales o de actualización desde las aplicaciones cliente. El enfoque que usa puertas de enlace de API se explica con detalle en secciones posteriores.

- Comunicación asincrónica basada en eventos. Esta comunicación se realiza mediante un bus de eventos para propagar las actualizaciones en los microservicios o para la integración con aplicaciones externas. El bus de eventos se puede implementar con cualquier tecnología de infraestructura de agente de

mensajería como RabbitMQ, o bien mediante Service Bus de nivel superior (nivel de abstracción) como Azure Service Bus, NServiceBus, MassTransit o Brighter.

La aplicación se implementa como un conjunto de microservicios en forma de contenedores. Las aplicaciones cliente pueden comunicarse con esos microservicios que se ejecuten como contenedores a través de las direcciones URL públicas publicadas por las puertas de enlace de API.

Propiedad de los datos por microservicio

En la aplicación de ejemplo, cada microservicio posee su propia base de datos u origen de datos, aunque todas las bases de datos de SQL Server se implementan como un contenedor único. Esta decisión de diseño se tomó solo para facilitar a los desarrolladores la obtención del código desde GitHub, clonarlo y abrirlo en Visual Studio o Visual Studio Code. También facilita la compilación de las imágenes de Docker personalizadas mediante la CLI de .NET y la de Docker, y la implementación y ejecución posteriores en un entorno de desarrollo de Docker. En cualquier caso, el uso de contenedores para orígenes de datos permite a los desarrolladores compilar e implementar en cuestión de minutos sin tener que aprovisionar una base de datos externa o cualquier otro origen de datos con dependencias en la infraestructura (en la nube o locales).

En un entorno de producción real, para alta disponibilidad y escalabilidad, las bases de datos deberían basarse en servidores de base de datos en la nube o locales, pero no en contenedores.

Por tanto, las unidades de implementación de los microservicios (e incluso de las bases de datos de esta aplicación) son contenedores de Docker y la aplicación de referencia es una aplicación de varios contenedores que se rige por los principios de los microservicios.

Recursos adicionales

- **Repositorio de GitHub de eShopOnContainers. Código fuente de la aplicación de referencia**
  https://aka.ms/eShopOnContainers/

Ventajas de una solución basada en microservicios

Una solución basada en microservicios como esta tiene muchas ventajas:

Cada microservicio es relativamente pequeño, fácil de administrar y desarrollar. De manera específica:

- Es fácil para los desarrolladores entender y empezar a trabajar rápidamente con buena productividad.
- Los contenedores se crean con rapidez, lo que permite que los desarrolladores sean más productivos.
- Un IDE como Visual Studio puede cargar proyectos más pequeños con rapidez, aumentando la productividad de los desarrolladores.
- Cada microservicio se puede diseñar, desarrollar e implementar con independencia de otros microservicios. Esto aporta agilidad, dado que es más fácil implementar nuevas versiones de los microservicios con frecuencia.

Es posible escalar horizontalmente áreas individuales de la aplicación. Por ejemplo, es posible que sea necesario escalar horizontalmente el servicio de catálogo o el de cesta de la compra, pero no el proceso de pedidos. Una infraestructura de microservicios será mucho más eficaz con respecto a los recursos que se usan durante el escalado horizontal que una arquitectura monolítica.

El trabajo de desarrollo se puede dividir entre varios equipos. Cada servicio puede ser propiedad de un único equipo de desarrollo. Cada equipo puede administrar, desarrollar, implementar y escalar su servicio de forma independiente a los demás equipos.

Los problemas son más aislados. Si se produce un problema en un servicio, inicialmente solo se ve afectado ese servicio (excepto cuando se usa un diseño incorrecto, con dependencias directas entre los microservicios) y los demás servicios pueden continuar con el control de las solicitudes. Por el contrario, un componente en mal estado en una arquitectura de implementación monolítica puede colapsar todo el sistema, especialmente si hay recursos implicados, como una fuga de memoria. Además, cuando se resuelve un problema en un microservicio, se puede implementar el microservicio afectado sin afectar al resto de la aplicación.

Se pueden usar las tecnologías más recientes. Como es posible empezar a desarrollar los servicios de forma independiente y ejecutarlos en paralelo (gracias a los contenedores y .NET), se pueden usar las tecnologías y plataformas más modernas de forma oportuna en lugar de atascarse en una pila o marco antiguo para toda la aplicación.

Desventajas de una solución basada en microservicios

Una solución basada en microservicios como esta también tiene algunas desventajas:

Aplicación distribuida. La distribución de la aplicación agrega complejidad para los desarrolladores cuando diseñen y creen los servicios. Por ejemplo, los desarrolladores deben implementar la comunicación entre servicios mediante protocolos como HTTP o AMPQ, lo que agrega complejidad a efectos de pruebas y control de excepciones. También agrega latencia al sistema.

Complejidad de la implementación. Una aplicación que tiene docenas de tipos de microservicios y que necesita alta escalabilidad (debe ser capaz de crear varias instancias por cada servicio y equilibrarlos entre varios hosts) supone un alto grado de complejidad de implementación para las operaciones de TI y administración. Si no se usa una infraestructura orientada a microservicios (por ejemplo, un orquestador y un programador), esa complejidad adicional puede requerir muchos más esfuerzos de desarrollo que la propia aplicación empresarial.

Transacciones atómicas. Normalmente, no se pueden realizar transacciones atómicas entre varios microservicios. Los requisitos de negocio deben adoptar la coherencia final entre varios microservicios.

Aumento de las necesidades de recursos globales (total de memoria, unidades y recursos de red para todos los hosts o servidores). En muchos casos, al reemplazar una aplicación monolítica con un enfoque de microservicios, la cantidad de recursos globales inicial necesaria para la nueva aplicación basada en microservicios será mayor que las necesidades de infraestructura de la aplicación monolítica original. Este enfoque se debe a que un mayor grado de granularidad y servicios distribuidos requiere más recursos globales. Sin embargo, dado el bajo costo de los recursos en general y la ventaja de poder escalar horizontalmente determinadas áreas de la aplicación en comparación con los costos a largo plazo a la hora de desarrollar aplicaciones monolíticas, el aumento en el uso de recursos normalmente es una ventaja para las grandes aplicaciones a largo plazo.

Problemas de comunicación directa de cliente a microservicio. Cuando la aplicación es grande, con docenas de microservicios, hay problemas y limitaciones si la aplicación requiere comunicaciones directas del cliente al microservicio. Un problema es un error de coincidencia potencial entre las necesidades del cliente y las API expuestas por cada uno de los microservicios. En algunos casos, es posible que la aplicación cliente tenga que realizar varias solicitudes independientes para crear la interfaz de usuario, lo que puede resultar ineficaz a través de Internet y poco práctico a través de una red móvil.

Por tanto, se deben minimizar las solicitudes de la aplicación cliente al sistema back-end.

Otro problema con las comunicaciones directas entre el cliente y el microservicio es la posibilidad de que algunos microservicios usen protocolos que no sean aptos para la web. Es posible que un servicio use un protocolo binario, mientras que otro use mensajería de AMQP. Estos protocolos no son compatibles con firewall y resultan más útiles cuando se usan internamente. Normalmente, una aplicación debería usar protocolos como HTTP y WebSockets para la comunicación fuera del firewall.

Otra desventaja con este enfoque directo de cliente a servicio es que resulta difícil refactorizar los contratos para esos microservicios. Con el tiempo, es posible que a los desarrolladores les interese cambiar la forma en que el sistema se divide en servicios. Por ejemplo, es posible que combinen dos servicios o dividan uno en dos o más servicios. Pero si los clientes se comunican directamente con los servicios, realizar este tipo de refactorización puede interrumpir la compatibilidad con las aplicaciones cliente.

Como se mencionó en la sección sobre arquitectura, al diseñar y crear una aplicación compleja basada en microservicios, podría considerar el uso de varias puertas de enlace de API específicas en lugar del enfoque más sencillo de comunicación directa entre el cliente y el microservicio.

Creación de particiones de los microservicios. Por último, independientemente del enfoque que se adopte para la arquitectura del microservicio, otro desafío consiste en decidir cómo dividir una aplicación integral en varios microservicios. Como se indicó en la sección sobre arquitectura de la guía, se pueden adoptar varias técnicas y enfoques. Básicamente, debe identificar las áreas de la aplicación que se separan del resto y que tienen un número reducido de dependencias fuertes. En muchos casos, este enfoque se alinea con la creación de particiones de los servicios por caso de uso. Por ejemplo, en la aplicación de tienda electrónica, hay un servicio de pedidos que se encarga de toda la lógica de negocios relacionada con el proceso de pedidos. También hay un servicio de catálogo y otro de cesta de la compra que implementan otras funciones. Idealmente, cada servicio solo debería tener un conjunto reducido de responsabilidades. Este enfoque es similar al principio de responsabilidad única (SRP) aplicado a las clases, que indica que una clase solo debe tener un motivo para cambiar. Pero en este caso, se trata de microservicios, por lo que el ámbito será mayor que el de una sola clase. Sobre todo, un microservicio tiene que ser autónomo de principio a fin, incluida la responsabilidad de sus propios orígenes de datos.

Diferencias entre patrones de arquitectura y diseño externos e internos

La arquitectura externa es la arquitectura de microservicio compuesta por varios servicios, siguiendo los principios descritos en la sección sobre arquitectura de esta guía. Pero en función de la naturaleza de cada microservicio y con independencia de la arquitectura general de microservicios que elija, es habitual y a veces aconsejable tener distintas arquitecturas internas, cada una basada en patrones diferentes, para los distintos microservicios. Los microservicios incluso pueden usar tecnologías y lenguajes de programación diferentes. En la figura 6-2 se ilustra esta diversidad.

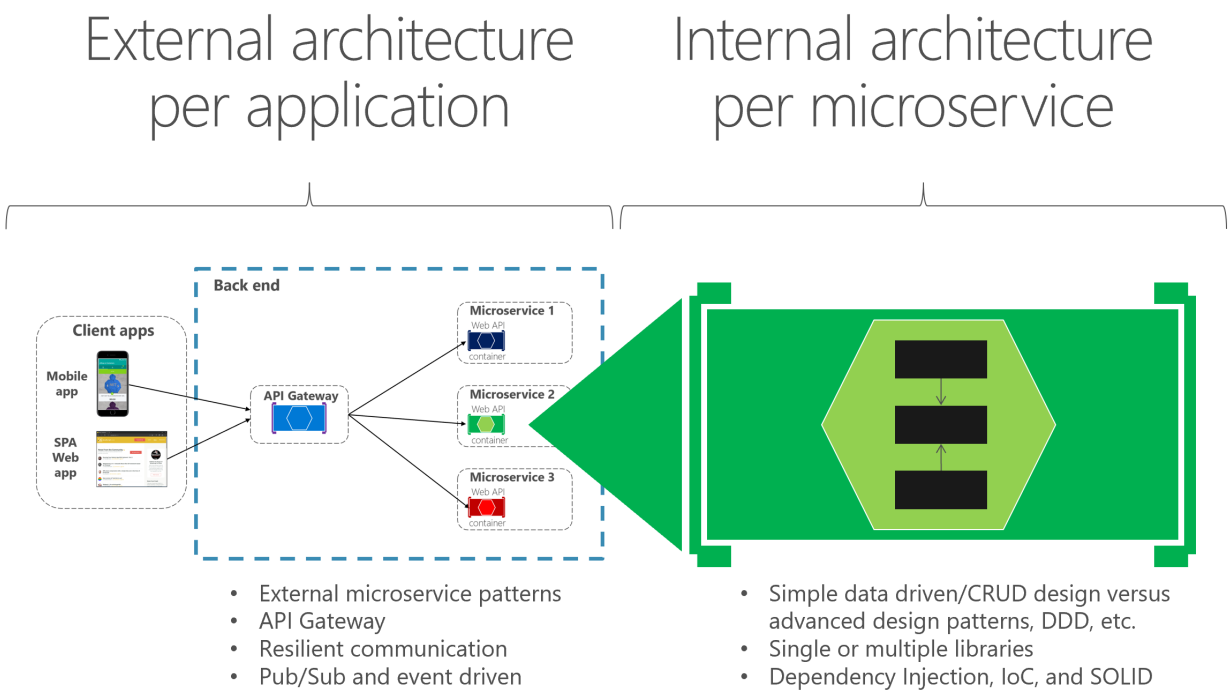

Figura 6-2. Diferencias entre arquitectura y diseño externos e internos

En el ejemplo *eShopOnContainers*, los microservicios de catálogo, cesta de la compra y perfil de usuario son simples (básicamente subsistemas de CRUD). Por tanto, su arquitectura y diseño internos son sencillos. Pero es posible que tenga otros microservicios, como el de pedidos, que sean más complejos y representen las reglas de negocios cambiantes con un alto grado de complejidad del dominio. En estos casos, es posible que le interese implementar modelos más avanzados dentro de un microservicio determinado, como los que se definen con los enfoques de diseño controlado por dominios (DDD), como se hace en el microservicio de pedidos de *eShopOnContainers*. (Estos patrones de DDD se describirán más adelante en la sección en la que se explica la implementación del microservicio de pedidos de *eShopOnContainers*).

Otra razón para usar una tecnología distinta por microservicio podría ser la naturaleza de cada microservicio. Por ejemplo, podría ser mejor usar un lenguaje de programación funcional como F#, o incluso un lenguaje como R si los dominios de destino son de IA y aprendizaje automático, en lugar de un lenguaje de programación más orientado a objetos como C#.

La conclusión es que cada microservicio puede tener una arquitectura interna diferente basada en patrones de diseño diferentes. Para evitar la ingeniería excesiva de los microservicios, no todos deben implementarse mediante patrones de DDD avanzados. Del mismo modo, los microservicios complejos con lógica de negocios cambiante no deberían implementarse como componentes CRUD o el resultado sería código de baja calidad.

El nuevo mundo: varios modelos arquitectónicos y microservicios políglotas

Los desarrolladores y arquitectos de software usan muchos modelos arquitectónicos. Los siguientes son algunos de ellos (se combinan estilos y modelos arquitectónicos):

- CRUD simple, de un nivel y una capa.
- Tradicional de N capas.
- Diseño controlado por dominios de N capas .
- Arquitectura limpia (como se usa con eShopOnWeb)
- Segregación de responsabilidades de consultas de comandos (CQRS).
- Arquitectura controlada por eventos (EDA).

También se pueden compilar microservicios con muchas tecnologías y lenguajes, como las API web de ASP.NET Core, NancyFx, ASP.NET Core SignalR (disponible con .NET Core 2 o versiones posteriores), F#, Node.js, Python, Java, C++, GoLang y otros.

Lo importante es que ningún modelo o estilo arquitectónico determinado, ni ninguna tecnología concreta, es adecuado para todas las situaciones. En la figura 6-3 se muestran algunos enfoques y tecnologías (aunque en ningún orden concreto) que se pueden usar en otros microservicios.

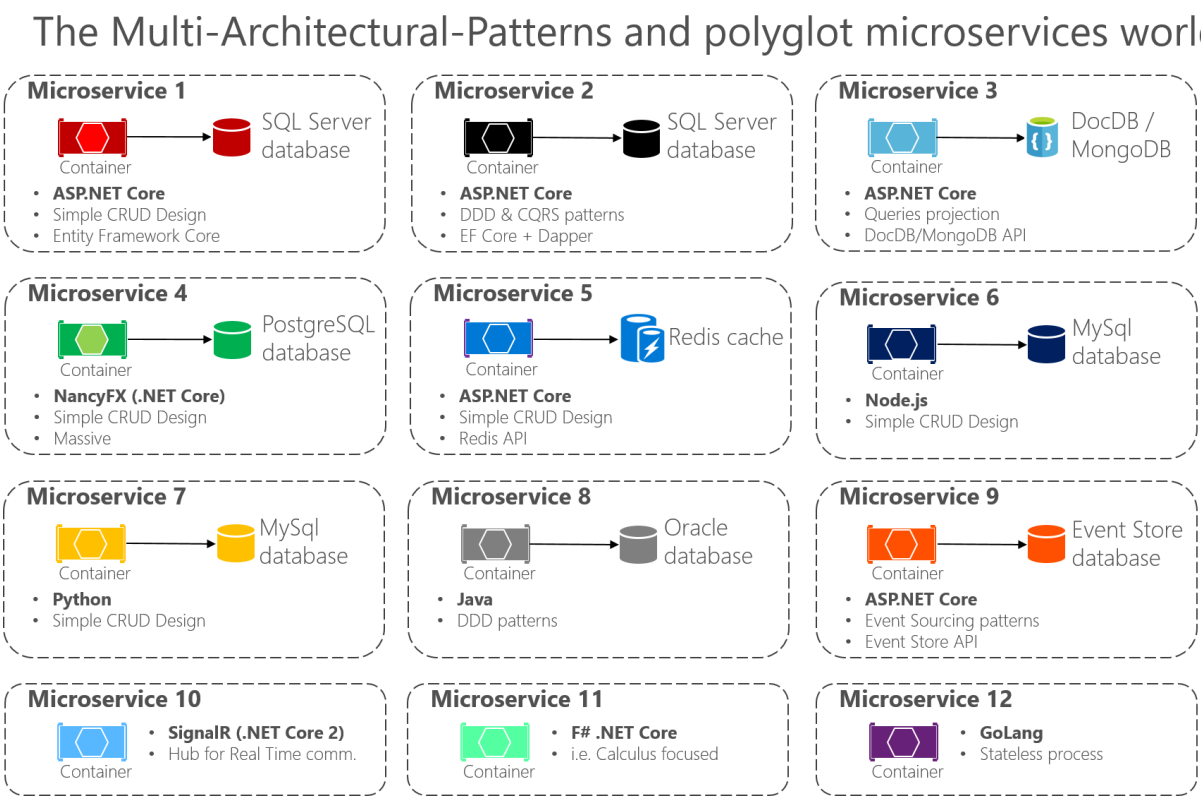

Figura 6-3. Modelos arquitectónicos múltiples y el mundo de los microservicios políglotas

Los patrones de varias arquitecturas y los microservicios políglotas implican que puede mezclar y adaptar lenguajes y tecnologías a las necesidades de cada microservicio y permitir que se sigan comunicando entre sí. Como se muestra en la figura 6-3, en las aplicaciones formadas por muchos microservicios (contextos delimitados en terminología del diseño controlado por dominios, o simplemente "subsistemas" como microservicios autónomos), podría implementar cada microservicio de forma diferente. Cada uno de ellos podría tener un modelo arquitectónico diferente y usar otros lenguajes y bases de datos según la naturaleza de la aplicación, los requisitos empresariales y las prioridades. En algunos casos, es posible que los microservicios sean similares. Pero eso no es lo habitual, porque el límite del contexto y los requisitos de cada subsistema suelen ser diferentes.

Por ejemplo, para una aplicación de mantenimiento CRUD simple, es posible que no tenga sentido diseñar e implementar patrones de DDD. Pero para el dominio o el negocio principal, es posible que tenga que aplicar patrones más avanzados para abordar la complejidad empresarial con reglas de negocio cambiantes.

Especialmente cuando se trabaja con aplicaciones de gran tamaño compuestas por varios subsistemas, no se debe aplicar una única arquitectura de nivel superior basada en un único modelo arquitectónico. Por ejemplo, no se debe aplicar CQRS como arquitectura de nivel superior para una aplicación completa, pero podría ser útil para un conjunto específico de servicios.

No hay ninguna solución mágica ni un modelo arquitectónico correcto para cada caso concreto. No se puede tener "un modelo arquitectónico para dominarlos a todos". Según las prioridades de cada microservicio, tendrá que elegir un enfoque diferente para cada uno, como se explica en las secciones siguientes.

Anterior Siguiente

Contenido recomendado

Creación de interfaces de usuario compuestas basadas en microservicios

La arquitectura de microservicios no es solo para el back-end. Obtenga una vista de inspección usándola en el front-end.

Microservicios de .NET. Arquitectura para aplicaciones .NET en contenedor

Arquitectura de Microservicios de .NET para aplicaciones .NET en contenedores | Los microservicios son servicios modulares que se pueden implementar de forma independiente. Los contenedores de Docker (para Linux y Windows) simplifican la…

Comunicación asincrónica basada en mensajes

Arquitectura de Microservicios de .NET para aplicaciones .NET en contenedor | Las comunicaciones asincrónicas basadas en mensajes son un concepto fundamental en la arquitectura de microservicios, porque es la mejor manera de mantener los microservicios…

Comunicación en una arquitectura de microservicio

Explore distintas formas de comunicación entre microservicios, y comprenda las implicaciones de formas sincrónicas y asincrónicas.

Mostrar más