Díganos qué opina sobre la experiencia de descarga del PDF.

Microservicios de .NET: arquitectura para aplicaciones .NET en contenedor

18/06/2025

EDITION v7.0 : actualizado a ASP.NET Core 7.0

Consulte el registro de cambios para obtener las actualizaciones del libro y las contribuciones de la comunidad.

Esta guía es una introducción al desarrollo de aplicaciones basadas en microservicios y su administración mediante contenedores. Se describen los enfoques de diseño e implementación de arquitectura mediante contenedores de .NET y Docker.

Para facilitar la introducción, la guía se centra en una aplicación en contenedores y basada en microservicios de referencia que puede explorar. La aplicación de referencia está disponible en el repositorio de GitHub eShopOnContainers .

Vínculos de acción

- Este libro electrónico también está disponible en formato PDF (solo versión en inglés) Descargar
- Clonar o bifurcar la aplicación de referencia eShopOnContainers en GitHub
- Vea el vídeo introductorio
- Conozca la arquitectura de microservicios inmediatamente

Introducción

Las empresas están aumentando el ahorro de costos, solucionando problemas de implementación y mejorando las operaciones de DevOps y producción mediante contenedores. Microsoft ha estado publicando innovaciones en contenedores para Windows y Linux mediante la creación de productos como Azure Kubernetes Service y Azure Service Fabric, y al asociarse con líderes del sector como Docker, Mesosphere y Kubernetes. Estos productos ofrecen soluciones de contenedor que ayudan a las empresas a crear e implementar aplicaciones a velocidad y escala en la nube, sea cual sea su elección de plataforma o herramientas.

Docker se está convirtiendo en el estándar de facto en el sector de contenedores, compatible con los proveedores más significativos de los ecosistemas de Windows y Linux. (Microsoft es uno de los principales proveedores de nube que admiten Docker). En el futuro, Docker probablemente estará omnipresente en cualquier centro de datos de la nube o en el entorno local.

Además, la arquitectura de microservicios está emergente como un enfoque importante para las aplicaciones críticas distribuidas. En una arquitectura basada en microservicios, la aplicación

se basa en una colección de servicios que se pueden desarrollar, probar, implementar y versionar de forma independiente.

Acerca de esta guía

Esta guía es una introducción al desarrollo de aplicaciones basadas en microservicios y su administración mediante contenedores. Se describen los enfoques de diseño e implementación de arquitectura mediante contenedores de .NET y Docker. Para facilitar la introducción a contenedores y microservicios, la guía se centra en una aplicación en contenedores y basada en microservicios de referencia que puede explorar. La aplicación de ejemplo está disponible en el repositorio de GitHub eShopOnContainers .

En esta guía se proporcionan instrucciones fundamentales sobre el desarrollo y la arquitectura principalmente en un nivel de entorno de desarrollo con un enfoque en dos tecnologías: Docker y .NET. Nuestra intención es que lea esta guía al pensar en el diseño de la aplicación sin centrarse en la infraestructura (en la nube o en el entorno local) de su entorno de producción. Tomará decisiones sobre la infraestructura más adelante cuando cree las aplicaciones listas para producción. Por lo tanto, esta guía está pensada para ser independiente de la infraestructura y más centrada en el entorno de desarrollo.

Después de haber estudiado esta guía, el siguiente paso sería obtener información sobre los microservicios listos para producción en Microsoft Azure.

Versión

Esta guía se ha revisado para cubrir la versión **de .NET 7** junto con muchas actualizaciones adicionales relacionadas con la misma "oleada" de tecnologías (es decir, Azure y tecnologías de terceros adicionales) que coinciden en el tiempo con la versión de .NET 7.

> **Nota**
>
> Se crea una nueva versión de este libro electrónico para .NET 8 y el nuevo ejemplo de eShop .

Lo que esta guía no cubre

Esta guía no se centra en el ciclo de vida de la aplicación, DevOps, canalizaciones de CI/CD ni trabajo en equipo. La guía complementaria Ciclo de vida de aplicaciones de Docker en contenedores con la plataforma y las herramientas de Microsoft se centra en ese tema. La

guía actual tampoco proporciona detalles de implementación en la infraestructura de Azure, como información sobre orquestadores específicos.

Recursos adicionales

- **Ciclo de vida de aplicaciones de Docker en contenedor con la plataforma y herramientas de Microsoft** (libro electrónico descargable) https://aka.ms/dockerlifecycleebook

Destinatarios de esta guía

Hemos escrito esta guía para desarrolladores y arquitectos de soluciones que son nuevos en el desarrollo de aplicaciones basado en Docker y en la arquitectura basada en microservicios. Esta guía es para usted si desea aprender a diseñar, diseñar e implementar aplicaciones de prueba de concepto con tecnologías de desarrollo de Microsoft (con especial enfoque en .NET) y con contenedores de Docker.

También encontrará esta guía útil si es un responsable de la toma de decisiones técnicas, como un arquitecto empresarial, que quiere una introducción a la arquitectura y la tecnología antes de decidir qué enfoque seleccionar para las aplicaciones distribuidas nuevas y modernas.

Cómo usar esta guía

La primera parte de esta guía presenta contenedores de Docker, describe cómo elegir entre .NET 7 y .NET Framework como marco de desarrollo y proporciona información general sobre los microservicios. Este contenido es para arquitectos y responsables de la toma de decisiones técnicas que desean obtener información general, pero no necesitan centrarse en los detalles de implementación del código.

La segunda parte de la guía comienza con la sección Desarrollo de aplicaciones basadas en Docker . Se centra en los patrones de desarrollo y microservicio para implementar aplicaciones mediante .NET y Docker. Esta sección será de mayor interés para los desarrolladores y arquitectos que desean centrarse en el código y en los patrones y los detalles de implementación.

Aplicación de referencia relacionada basada en microservicios y contenedores: eShopOnContainers

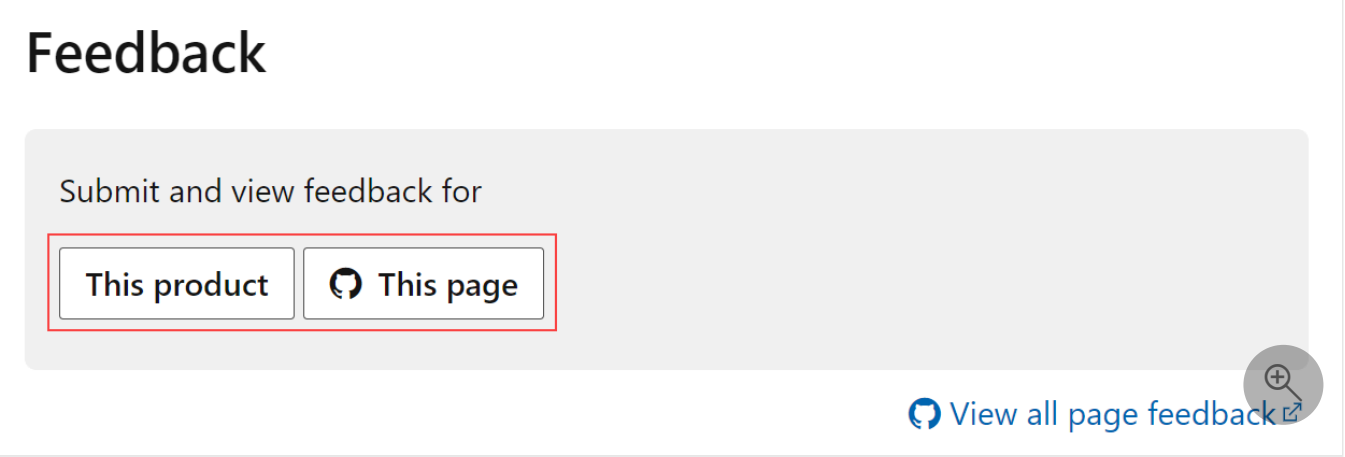
La aplicación eShopOnContainers es una aplicación de referencia de código abierto para .NET y microservicios diseñados para implementarse mediante contenedores de Docker. La aplicación

consta de varios subsistemas, incluidos varios front-end de interfaz de usuario de tienda electrónica (una aplicación MVC web, un SPA web y una aplicación móvil nativa). También incluye los microservicios y contenedores de back-end para todas las operaciones necesarias del lado servidor.

El propósito de la aplicación es mostrar patrones arquitectónicos. **PRODUCTION-READY no es una plantilla** para iniciar aplicaciones reales. De hecho, la aplicación está en un estado beta permanente, ya que también se usa para probar nuevas tecnologías potencialmente interesantes a medida que aparecen.

Enviar sus comentarios

Este libro y ejemplos relacionados evolucionan constantemente, por lo que se agradecen sus comentarios. Si tiene comentarios sobre cómo se puede mejorar este libro y está leyendo este libro en <learn.microsoft.com>, use la sección **Comentarios** en la parte inferior de la página:



Como se ha resaltado en la captura de pantalla anterior, la sección de comentarios le permite enviar comentarios para:

- **Este producto**: mediante el formulario de comentarios del producto .NET.
- **Esta página**: usando una plantilla de incidencia de GitHub con los detalles de la página.

Si está leyendo este libro como un PDF, puede enviar comentarios creando un problema de GitHub en Documentos de .NET o mediante la plantilla de problemas de GitHub en Libros electrónicos de arquitectura de .NET.

Créditos

Coautores:

> **Cesar de la Torre**, Sr. PM, equipo de productos de .NET, Microsoft Corp.

> **Bill Wagner**, Sr. Content Developer, C+E, Microsoft Corp.
>
> **Mike Rousos**, ingeniero principal de software, equipo CAT de DevDiv, Microsoft

Editores:

> **Mike Pope**
>
> **Steve Hoag**

Participantes y revisores:

> **Jeffrey Richter**, Partner Software Eng, Equipo de Azure, Microsoft
>
> **Jimmy Bogard**, arquitecto jefe de Headpring
>
> **Udi Dahan**, Fundador y CEO, Software particular
>
> **Jimmy Nilsson**, co-fundador y ceo de Factor10
>
> **Glenn Condron**, director del programa, ASP.NET equipo
>
> **Mark Fussell**, principal responsable de pm, equipo de Azure Service Fabric, Microsoft
>
> **Diego Vega**, responsable de pm, equipo de Entity Framework, Microsoft
>
> **Barry Dorrans**, Director del Programa de Seguridad Sr.
>
> **Rowan Miller**, Sr. Program Manager, Microsoft
>
> **Ankit Asthana**, administrador principal de pm, equipo de .NET, Microsoft
>
> **Scott Hunter**, director asociado de administración de programas del equipo de .NET, Microsoft
>
> **Nish Anil**, Sr. Program Manager, equipo de .NET, Microsoft
>
> **Dylan Reisenberger**, arquitecto y jefe de desarrollo en Polly
>
> **Steve "ardalis" Smith -** Arquitecto de software y Entrenador - Ardalis.com
>
> **Ian Cooper**, arquitecto de codificación en Brighter
>
> **Unai Zorrilla**, arquitecto y jefe de desarrollo en Plain Concepts
>
> **Eduard Tomas**, jefe de desarrollo en Plain Concepts
>
> **Ramon Tomas**, desarrollador en Plain Concepts

David Sanz, desarrollador en Plain Concepts

Javier Valero, director de operaciones de Grupo Solutio

Pierre Millet, Consultor Senior, Microsoft

Michael Friis, Product Manager, Docker Inc

Charles Lowell, ingeniero de software, equipo de VS CAT, Microsoft

Miguel Veloso, ingeniero de desarrollo de software en Plain Concepts

Sumit Ghosh, consultor principal de Neudesic

Derechos de autor

PUBLICADO POR

Equipos de producto de Microsoft Developer Division, .NET y Visual Studio

Una división de Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2023 de Microsoft Corporation

Todos los derechos reservados. No se podrá reproducir ni transmitir ninguna parte del contenido de este libro en cualquier forma ni por ningún medio sin el permiso por escrito del editor.

Este libro se proporciona “tal cual” y expresa las opiniones del autor. Las vistas, opiniones e información expresadas en este libro, incluidas las direcciones URL y otras referencias a sitios web de Internet, pueden cambiar sin previo aviso.

Algunos ejemplos que se muestran aquí se proporcionan solo para la ilustración y son ficticios. No se pretende ni debe deducirse ninguna asociación o conexión real.

Microsoft y las marcas registradas que aparecen en https://www.microsoft.com la página web "Marcas comerciales" son marcas comerciales del grupo de empresas de Microsoft.

Mac y macOS son marcas comerciales de Apple Inc.

El logotipo de la ballena de Docker es una marca registrada de Docker, Inc. Usado por permiso.

Todas las demás marcas y logotipos son propiedad de sus respectivos propietarios.

Próximo

Introducción a contenedores y Docker

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

La inclusión en contenedores es un enfoque de desarrollo de software en el que una aplicación o un servicio, sus dependencias y su configuración (extraídos como archivos de manifiesto de implementación) se empaquetan como una imagen de contenedor. La aplicación en contenedor se puede probar como una unidad e implementarse como una instancia de imagen de contenedor en el sistema operativo host (SO).

Al igual que los contenedores de envío permiten transportar mercancías por barco, tren o camión independientemente de la carga dentro, los contenedores de software actúan como una unidad estándar de implementación de software que puede contener diferentes código y dependencias. El software en contenedores de esta manera permite a los desarrolladores y profesionales de TI implementarlos en entornos con poca o ninguna modificación.

Los contenedores también aíslan las aplicaciones entre sí en un sistema operativo compartido. Las aplicaciones en contenedores se ejecutan sobre un host de contenedor que, a su vez, se ejecuta en el sistema operativo (Linux o Windows). Por lo tanto, los contenedores tienen una superficie significativamente menor que las imágenes de máquina virtual (VM).

Cada contenedor puede ejecutar una aplicación web completa o un servicio, como se muestra en la figura 2-1. En este ejemplo, el host de Docker es un host de contenedor y App1, App2, Svc 1 y Svc 2 son aplicaciones o servicios contenedorizados.

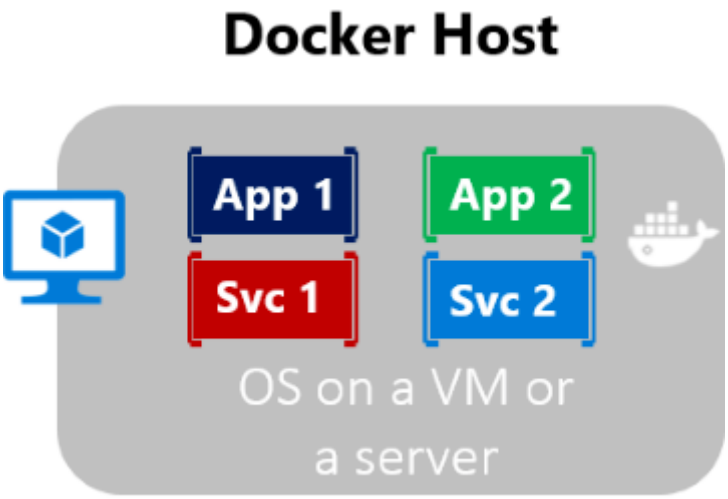


Figura 2-1. Varios contenedores que se ejecutan en un host de contenedor

Otra ventaja de la inclusión en contenedores es la escalabilidad. La creación de contenedores para tareas a corto plazo permite escalar horizontalmente con gran rapidez. Desde un punto de vista de la aplicación, crear instancias de una imagen (crear un contenedor) es similar a crear instancias de un proceso como un servicio o una aplicación web. Sin embargo, para la confiabilidad, al ejecutar varias instancias de la misma imagen en varios servidores host, normalmente quiere que cada contenedor (instancia de imagen) se ejecute en un servidor host o máquina virtual diferente en dominios de error diferentes.

En resumen, los contenedores ofrecen las ventajas de aislamiento, portabilidad, agilidad, escalabilidad y control en todo el flujo de trabajo del ciclo de vida de la aplicación. La ventaja más importante es el aislamiento del entorno proporcionado entre Dev y Ops.

Siguiente anterior

¿Qué es Docker?

Artículo • 30/01/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Docker es un proyecto de código abierto para automatizar la implementación de aplicaciones como contenedores portátiles y autoadministrados que se pueden ejecutar en la nube o en el entorno local. Docker también es una empresa de que promueve y evoluciona esta tecnología, trabajando en colaboración con proveedores en la nube, Linux y Windows, incluido Microsoft.

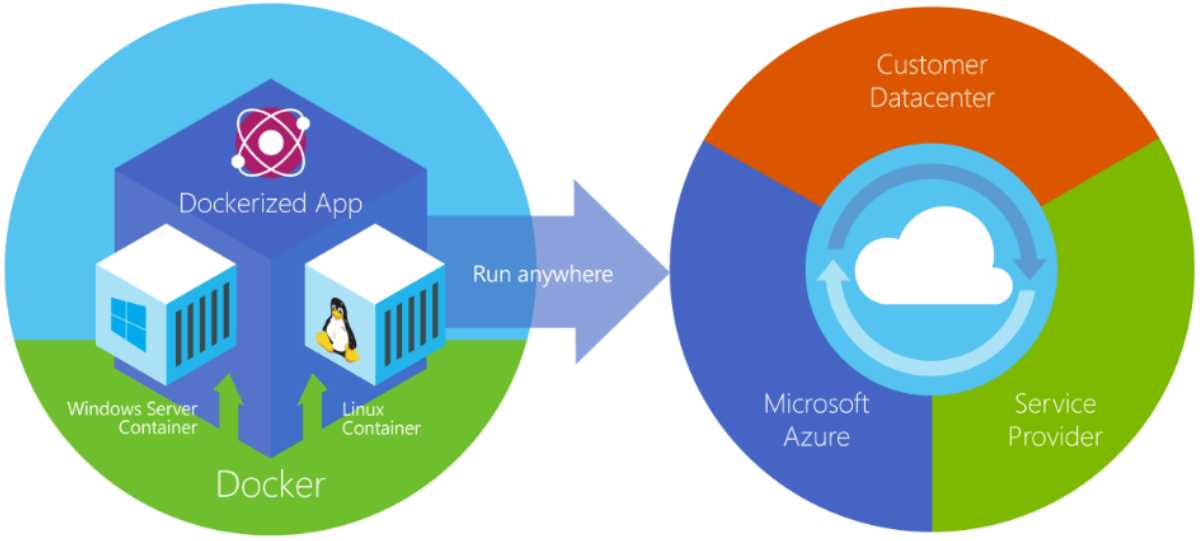


Figura 2-2. Docker implementa contenedores en todas las capas de la nube híbrida.

Los contenedores de Docker se pueden ejecutar en cualquier lugar, local en el centro de datos del cliente, en un proveedor de servicios externo o en la nube, en Azure. Los contenedores de imágenes de Docker se pueden ejecutar de forma nativa en Linux y Windows. Sin embargo, las imágenes de Windows solo se pueden ejecutar en hosts de Windows, mientras que las imágenes de Linux se pueden ejecutar tanto en hosts de Linux como en hosts de Windows, utilizando una máquina virtual de Linux de Hyper-V hasta ahora. Aquí, "host" se refiere a un servidor o una máquina virtual.

Los desarrolladores pueden usar entornos de desarrollo en Windows, Linux o macOS. En el equipo de desarrollo, el desarrollador ejecuta un host de Docker donde se implementan imágenes de Docker, incluida la aplicación y sus dependencias. Los desarrolladores que trabajan en Linux o en macOS usan un host de Docker basado en Linux y solo pueden crear imágenes para contenedores de Linux. (Los desarrolladores que trabajan en macOS pueden editar código o ejecutar la CLI de Docker desde macOS, pero a partir del momento en que se escribe, los contenedores no se ejecutan directamente en macOS). Los desarrolladores que trabajan en Windows pueden crear imágenes para contenedores de Linux o Windows.

Para hospedar contenedores en entornos de desarrollo y proporcionar herramientas de desarrollo adicionales, Docker incluye Docker Desktop para Windows o para macOS. Estos productos instalan la máquina virtual necesaria (el host de Docker) para hospedar los contenedores.

Para ejecutar contenedores de Windows, hay dos tipos de tiempos de ejecución:

- Los contenedores de Windows Server proporcionan aislamiento de aplicaciones mediante tecnología de aislamiento de procesos y espacios de nombres. Un contenedor de Windows Server comparte un kernel con el host de contenedor y con todos los contenedores que se ejecutan en el host.

- Hyper-V Los contenedores amplían el aislamiento proporcionado por los contenedores de Windows Server al ejecutar cada contenedor en una máquina virtual altamente optimizada. En esta configuración, el kernel del host de contenedor no se comparte con los contenedores de Hyper-V, lo que proporciona un mejor aislamiento.

Las imágenes de estos contenedores se crean de la misma manera y funcionan igual. La diferencia está en cómo se crea el contenedor a partir de la imagen; ejecutar un contenedor Hyper-V requiere un parámetro adicional. Para más información, vea Contenedores de Hyper-V.

Comparación de contenedores de Docker con máquinas virtuales

En la figura 2-3 se muestra una comparación entre las máquinas virtuales y los contenedores de Docker.

Expandir tabla

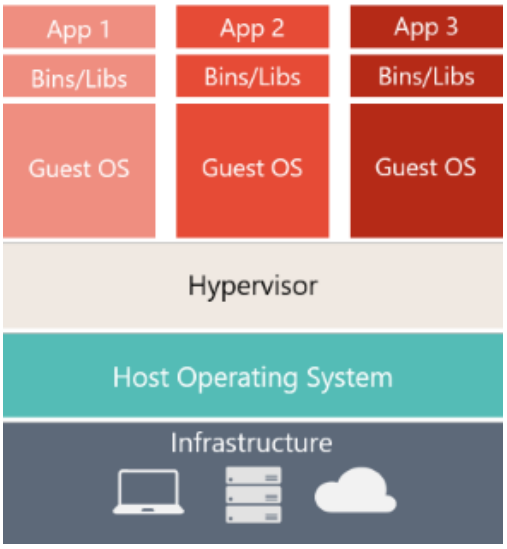
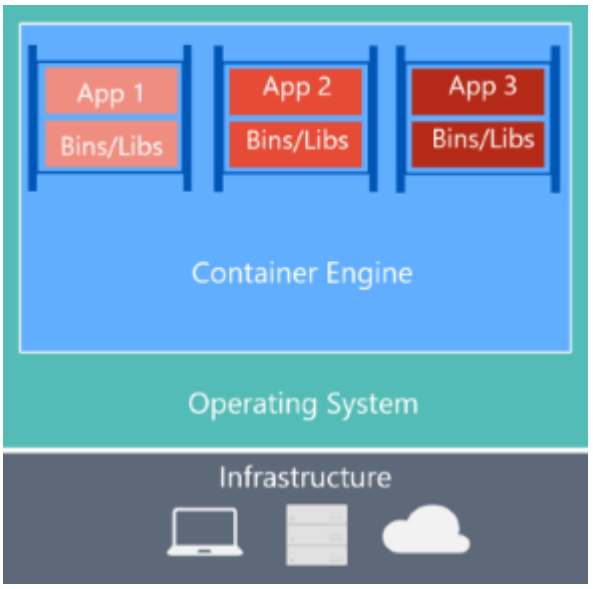
Máquinas virtuales	Contenedores de Docker
	
Las máquinas virtuales incluyen la aplicación, las bibliotecas o archivos binarios necesarios y un sistema operativo invitado completo. La virtualización completa requiere más recursos que la contenedorización.	Los contenedores incluyen la aplicación y todas sus dependencias. Sin embargo, comparten el kernel del sistema operativo con otros contenedores, ejecutándose como procesos aislados en el espacio de usuario en el sistema operativo host. (Excepto en contenedores Hyper-V, donde cada contenedor se ejecuta dentro de una máquina virtual especial para cada contenedor).

Figura 2-3. Comparación de máquinas virtuales tradicionales con contenedores de Docker

En el caso de las máquinas virtuales, hay tres capas base en el servidor host, desde la parte inferior: infraestructura, sistema operativo host y un hipervisor y encima de todo lo que cada máquina virtual tiene su propio sistema operativo y todas las bibliotecas necesarias. Para Docker, el servidor host solo tiene la infraestructura y el sistema operativo y encima de eso, el motor de contenedor, que mantiene el contenedor aislado pero comparte los servicios del sistema operativo base.

Dado que los contenedores requieren mucho menos recursos (por ejemplo, no necesitan un sistema operativo completo), son fáciles de implementar y comienzan

rápidamente. Esto le permite tener una mayor densidad, lo que significa que permite ejecutar más servicios en la misma unidad de hardware, lo que reduce los costos.

Como efecto secundario de la ejecución en el mismo kernel, obtendrá menos aislamiento que las máquinas virtuales.

El objetivo principal de una imagen es que el entorno (dependencias) sea el mismo en distintas implementaciones. Esto significa que puede depurarlo en su equipo y, a continuación, implementarlo en otra máquina con el mismo entorno garantizado.

Una imagen de contenedor es una manera de empaquetar una aplicación o servicio e implementarla de forma confiable y reproducible. Podría decir que Docker no es solo una tecnología, sino también una filosofía y un proceso.

Al usar Docker, no escuchará a los desarrolladores decir: "Funciona en mi máquina, ¿por qué no en producción?" Simplemente pueden decir: "Se ejecuta en Docker", ya que la aplicación de Docker empaquetada se puede ejecutar en cualquier entorno de Docker compatible y ejecuta la forma en que estaba pensada para en todos los destinos de implementación (como Dev, QA, staging y production).

Una analogía simple

Quizás una analogía simple puede ayudar a comprender el concepto principal de Docker.

Volvamos en el tiempo a los años 1950 por un momento. No había procesadores de texto, y además, las fotocopiadoras se usaban en todas partes (más o menos).

Imagine que es responsable de emitir rápidamente lotes de cartas según sea necesario, para enviarlos a los clientes, utilizando papel y sobres reales, para que se entreguen físicamente a la dirección de cada cliente (no había ningún correo electrónico entonces).

En algún momento, te das cuenta de que las cartas son simplemente una composición de un gran conjunto de párrafos, que se seleccionan y organizan según el propósito de la carta, por lo que desarrollas un sistema para emitir cartas rápidamente, esperando obtener un buen aumento.

El sistema es sencillo:

1. Comienza con una baraja de hojas transparentes que contienen un párrafo cada uno.

2. Para preparar un conjunto de cartas, elige las hojas con los párrafos que necesitas, luego apílalas y alinéalas para que se vean bien y sean legibles.

3. Por último, coloca el conjunto en la fotocopiadora y presiona el inicio para producir tantas letras como sea necesario.

Por lo tanto, simplificando, esa es la idea principal de Docker.

En Docker, cada capa es el conjunto resultante de cambios que se producen en el sistema de archivos después de ejecutar un comando, como instalar un programa.

Por lo tanto, cuando miras el sistema de archivos después de copiar la capa, verás todos los archivos que están en la capa cuando se instaló el programa.

Puede pensar en una imagen como un disco duro auxiliar de solo lectura listo para instalarse en un "equipo" donde el sistema operativo ya está instalado.

Del mismo modo, puede considerar un contenedor como el "equipo" con el disco duro de imagen instalado. El contenedor, al igual que un equipo, puede encenderse o apagarse.

siguiente

Terminología de Docker

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En esta sección se enumeran los términos y definiciones con los que debe estar familiarizado antes de profundizar en Docker. Para obtener más definiciones, consulte el glosario extenso proporcionado por Docker.

Imagen de contenedor: un paquete con todas las dependencias e información necesarias para crear un contenedor. Una imagen incluye todas las dependencias (como marcos) más la configuración de implementación y ejecución que usará un entorno de ejecución de contenedor. Normalmente, una imagen se deriva de varias imágenes base que se apilan entre sí para formar el sistema de archivos del contenedor. Una imagen es inmutable una vez creada.

Dockerfile: un archivo de texto que contiene instrucciones para compilar una imagen de Docker. Es como un script por lotes, la primera línea indica la imagen base para comenzar y, a continuación, siga las instrucciones para instalar los programas necesarios, copiar archivos, etc., hasta que obtenga el entorno de trabajo que necesita.

Compilación: la acción de crear una imagen de contenedor basada en la información y el contexto proporcionados por su Dockerfile, además de archivos adicionales en la carpeta donde se compila la imagen. Puede compilar imágenes con el siguiente comando de Docker:

```
Bash
```

```
docker build
```

Contenedor: una instancia de una imagen de Docker. Un contenedor representa la ejecución de una sola aplicación, proceso o servicio. Consta del contenido de una imagen de Docker, un entorno de ejecución y un conjunto estándar de instrucciones. Al escalar un servicio, se crean varias instancias de un contenedor a partir de la misma imagen. O bien, un trabajo por lotes puede crear varios contenedores a partir de la misma imagen, pasando parámetros diferentes a cada instancia.

Volúmenes: ofrezca un sistema de archivos grabable que el contenedor pueda usar. Dado que las imágenes son de solo lectura, pero la mayoría de los programas necesitan escribir en el sistema de archivos, los volúmenes agregan una capa grabable, sobre la imagen de contenedor, por lo que los programas tienen acceso a un sistema de archivos grabable. El programa no sabe que está accediendo a un sistema de archivos en capas, es solo el sistema de archivos como de costumbre. Los volúmenes residen en el sistema host y se administran mediante Docker.

Etiqueta: una marca o etiqueta que se puede aplicar a las imágenes para que se puedan identificar diferentes imágenes o versiones de la misma imagen (según el número de versión o el entorno de destino).

Compilación en varias fases: es una característica, ya que Docker 17.05 o posterior, que ayuda a reducir el tamaño de las imágenes finales. Por ejemplo, una imagen base grande, que contiene el SDK se puede usar para compilar y publicar y, a continuación, se puede usar una imagen base de solo tiempo de ejecución pequeña para hospedar la aplicación.

Repositorio (repo): una colección de imágenes Docker relacionadas, marcadas con una etiqueta que indica la versión de la imagen. Algunos repositorios contienen varias variantes de una imagen específica, como una imagen que contiene SDK (más pesado), una imagen que solo contiene entornos de ejecución (más ligeros), etc. Esas variantes se pueden marcar con etiquetas. Un solo repositorio puede contener variantes de plataforma, como una imagen de Linux y una imagen de Windows.

Registro: un servicio que proporciona acceso a los repositorios. El registro predeterminado para la mayoría de las imágenes públicas es Docker Hub (propiedad de Docker como organización). Normalmente, un registro contiene repositorios de varios equipos. Las empresas suelen tener registros privados para almacenar y administrar imágenes que han creado. Azure Container Registry es otro ejemplo.

Imagen de varios arcos: para la arquitectura múltiple (o multiplataforma), es una característica de Docker que simplifica la selección de la imagen adecuada, según la plataforma en la que se ejecuta Docker. Por ejemplo, cuando un Dockerfile solicita una imagen base **FROM**

mcr.microsoft.com/dotnet/sdk:8.0 del registro, en realidad obtiene **8.0-nanoserver-ltsc2022**, **8.0-nanoserver-1809** o **8.0-bullseye-slim**, en función del sistema operativo y la versión donde se ejecuta Docker.

Docker Hub: registro público para cargar imágenes y trabajar con ellas. Docker Hub proporciona hospedaje de imágenes de Docker, registros públicos o privados, desencadenadores de compilación y enlaces web e integración con GitHub y Bitbucket.

Azure Container Registry: un recurso público para trabajar con imágenes de Docker y sus componentes en Azure. Esto proporciona un registro que está cerca de las implementaciones en Azure y que proporciona control sobre el acceso, lo que permite usar los grupos y permisos de Azure Active Directory.

Registro de confianza de Docker (DTR): un servicio de registro de Docker (desde Docker) que se puede instalar localmente para que resida dentro del centro de datos y la red de la organización. Es conveniente para la gestión de imágenes privadas dentro de la empresa. El Registro de confianza de Docker se incluye como parte del producto Docker Datacenter.

Docker Desktop: herramientas de desarrollo para Windows y macOS para compilar, ejecutar y probar contenedores localmente. Docker Desktop para Windows proporciona entornos de desarrollo para contenedores de Linux y Windows. El host de Docker de Linux en Windows se basa en una máquina virtual de Hyper-V . El host para contenedores de Windows se basa directamente en Windows. Docker Desktop para Mac se basa en el marco de Hypervisor de Apple y en el hipervisor xhyve, que proporciona una máquina virtual host de Docker linux en macOS. Docker Desktop para Windows y para Mac reemplaza Docker Toolbox, que se basaba en Oracle VirtualBox.

Compose: Una herramienta de línea de comandos y un formato de archivo YAML con metadatos para definir y ejecutar aplicaciones con múltiples contenedores. Se define una sola aplicación basada en varias imágenes con uno o varios archivos .yml que pueden invalidar valores en función del entorno. Después de crear las definiciones, puede implementar toda la aplicación de varios contenedores con un solo comando (docker-compose up) que cree un contenedor por imagen en el host de Docker.

Clúster: una colección de hosts de Docker expuestos como si fuera un único host de Docker virtual para que la aplicación pueda escalar a varias instancias de los servicios distribuidos entre varios hosts dentro del clúster. Los clústeres de Docker se pueden crear con Kubernetes, Azure Service Fabric, Docker Swarm y Mesosphere DC/OS.

Orchestrator: herramienta que simplifica la administración de clústeres y hosts de Docker. Los orquestadores permiten administrar sus imágenes, contenedores y hosts a través de una interfaz de línea de comandos (CLI) o una interfaz gráfica de usuario. Puede administrar redes de contenedores, configuraciones, equilibrio de carga, detección de servicios, alta

disponibilidad, configuración del host de Docker, etc. Un orquestador se encarga de ejecutar, distribuir, escalar y reparar las cargas de trabajo a través de una colección de nodos. Normalmente, los productos de orquestador son los mismos productos que proporcionan infraestructura de clúster, como Kubernetes y Azure Service Fabric, entre otras ofertas del mercado.

Siguiente anterior

Contenedores, imágenes y registros de Docker

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Cuando se usa Docker, un desarrollador crea una aplicación o servicio y la empaqueta y sus dependencias en una imagen de contenedor. Una imagen es una representación estática de la aplicación o el servicio y de su configuración y las dependencias.

Para ejecutar la aplicación o el servicio, se crea una instancia de la imagen de la aplicación para crear un contenedor, que se ejecutará en el host de Docker. Los contenedores se prueban inicialmente en un entorno de desarrollo o pc.

Los desarrolladores deben almacenar imágenes en un registro, que actúa como una biblioteca de imágenes y es necesaria al implementar en orquestadores de producción. Docker mantiene un registro público a través de Docker Hub; otros proveedores proporcionan registros para diferentes colecciones de imágenes, incluido Azure Container Registry. Como alternativa, las empresas pueden tener un registro privado local para sus propias imágenes de Docker.

En la figura 2-4 se muestra cómo se relacionan las imágenes y los registros en Docker con otros componentes. También muestra las ofertas de servicios de registro de varios proveedores.

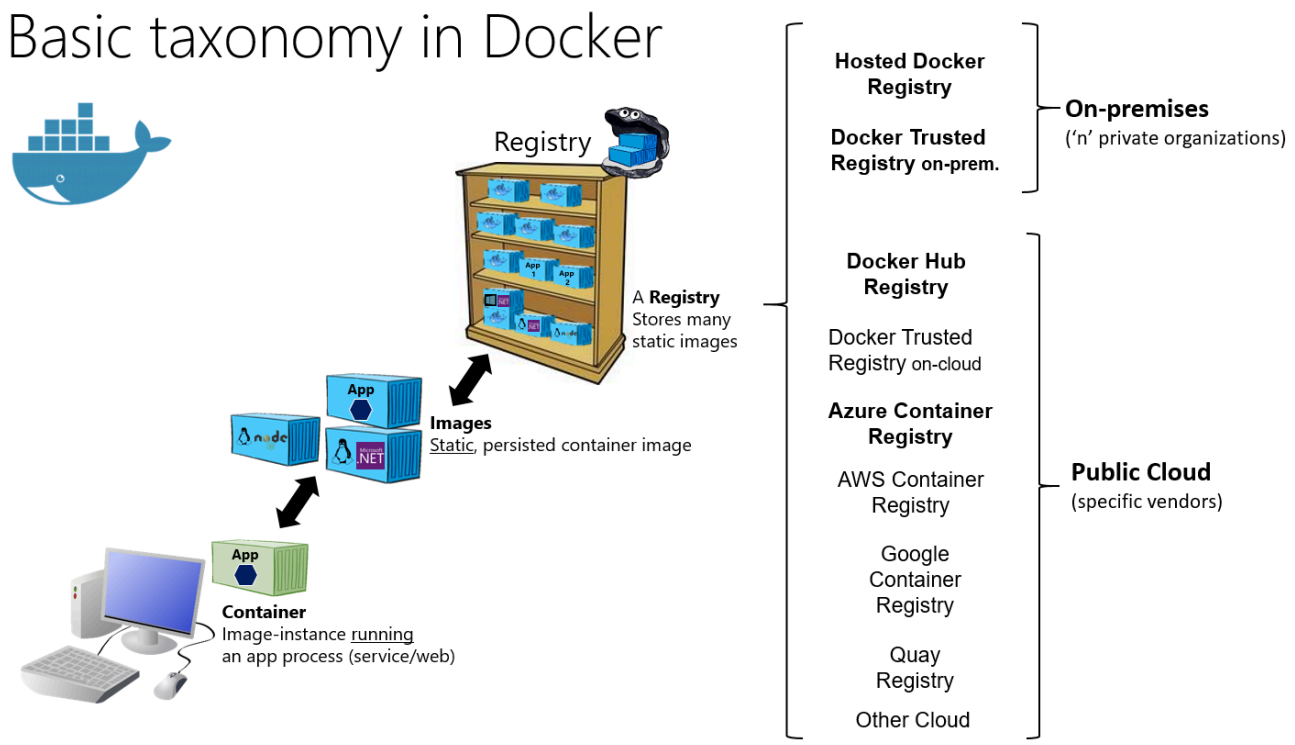


Figura 2-4. Taxonomía de los términos y conceptos de Docker

el registro es como una estantería donde las imágenes se almacenan y están disponibles para extraerlas con el fin de compilar contenedores que ejecuten servicios o aplicaciones web. Hay registros privados de Docker locales y en la nube pública. Docker Hub es un registro público mantenido por Docker, junto con el Registro de Confianza de Docker, una solución de nivel empresarial, Azure ofrece Azure Container Registry. AWS, Google y otros también tienen registros de contenedor.

Colocar imágenes en un registro le permite almacenar bits de aplicación estáticos e inmutables, incluidas todas sus dependencias en un nivel de marco. Esas imágenes pueden versionarse y implementarse en varios entornos y, por lo tanto, proporcionar una unidad de implementación coherente.

Los registros de imágenes privadas, hospedados en el entorno local o en la nube, se recomiendan cuando:

- Las imágenes no deben compartirse públicamente debido a la confidencialidad.
- Quiere tener una latencia de red mínima entre las imágenes y el entorno de implementación elegido. Por ejemplo, si el entorno de producción es la nube de Azure, probablemente quiera almacenar las imágenes en Azure Container Registry para que la latencia de red sea mínima. De forma similar, si el entorno de producción es local, es posible que desee tener un registro de confianza de Docker local disponible en la misma red local.

Elección entre .NET y .NET Framework para contenedores de Docker

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Hay dos marcos compatibles para compilar aplicaciones de Docker en contenedor del lado servidor con .NET: .NET Framework y .NET 8. Comparten muchos componentes de la plataforma .NET y puede compartir código entre los dos. Sin embargo, existen diferencias fundamentales entre ellos y qué marco usará dependerá de lo que quiera lograr. En esta sección se proporcionan instrucciones sobre cuándo elegir cada marco.

Siguiente anterior

Guía general

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En esta sección se proporciona un resumen de cuándo elegir .NET 8 o .NET Framework. Se proporcionan más detalles sobre estas opciones en las secciones siguientes.

Use .NET 8, con contenedores de Linux o Windows, para la aplicación de servidor docker en contenedores cuando:

- Tenga necesidades multiplataforma. Por ejemplo, quiere utilizar tanto contenedores de Linux como de Windows.
- La arquitectura de la aplicación se basa en microservicios.
- Necesite iniciar contenedores rápidamente y quiera que una pequeña superficie por contenedor alcance una mejor densidad o más contenedores por unidad de hardware con el fin de reducir costos.

En resumen, al crear nuevas aplicaciones .NET en contenedor, debe considerar .NET 8 como opción predeterminada. Tiene muchas ventajas y se adapta mejor a la filosofía y estilo de trabajo de los contenedores.

Una ventaja adicional de usar .NET 8 es que puede ejecutar versiones de .NET en paralelo para aplicaciones dentro del mismo equipo. Esta ventaja es más importante para los servidores o máquinas virtuales que no usan contenedores, ya que los contenedores aíslan las versiones de

.NET que necesita la aplicación. (Siempre que sean compatibles con el sistema operativo subyacente).

Use .NET Framework para la aplicación de servidor docker en contenedor cuando:

- La aplicación usa actualmente .NET Framework y tiene dependencias sólidas en Windows.
- Debe usar las API de Windows que no son compatibles con .NET 8.
- Debe usar bibliotecas de .NET de terceros o paquetes NuGet que no están disponibles para .NET 8.

El uso de .NET Framework en Docker puede mejorar las experiencias de implementación minimizando los problemas de implementación. Este escenario de "lift and shift" es importante para contenerizar aplicaciones heredadas que fueron desarrolladas originalmente con el .NET Framework, como ASP.NET WebForms, aplicaciones web MVC o servicios WCF (Windows Communication Foundation).

Recursos adicionales

- **Libro electrónico: Modernización de aplicaciones de .NET Framework existentes con contenedores de Azure y Windows**
  https://aka.ms/liftandshiftwithcontainersebook
- **Aplicaciones de ejemplo: modernización de aplicaciones web de ASP.NET heredadas mediante contenedores de Windows**
  https://aka.ms/eshopmodernizing

Siguiente anterior

Cuándo elegir .NET para contenedores de Docker

Artículo • 31/10/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

La naturaleza modular y ligera de .NET 8 resulta perfecta para contenedores. Al implementar e iniciar un contenedor, su imagen es mucho más pequeña con .NET 8 que con .NET Framework. En cambio, al usar .NET Framework para un contenedor, es necesario basar la imagen en la de Windows Server Core, que es mucho más pesada que las imágenes de Windows Nano Server o Linux que se usan para .NET 8.

Además, .NET 8 es multiplataforma, por lo que es posible implementar aplicaciones de servidor con imágenes de contenedor de Linux o Windows. Pero, si está utilizando el tradicional .NET Framework, solo puede implementar imágenes basadas en Windows Server Core.

A continuación, se ofrece una explicación más detallada sobre por qué elegir .NET 8.

Desarrollo e implementación multiplataforma

Obviamente, si su objetivo es tener una aplicación (servicio o aplicación web) que se pueda ejecutar en diferentes plataformas compatibles con Docker (Linux y Windows), la

opción adecuada es .NET 8, puesto que .NET Framework solo es compatible con Windows.

.NET 8 también admite macOS como plataforma de desarrollo. Pero, al implementar contenedores en un host Docker, el host debe estar (actualmente) basado en Linux o Windows. Por ejemplo, en un entorno de desarrollo, podría utilizar una máquina virtual Linux que se ejecutara en un equipo Mac.

Visual Studio proporciona un entorno de desarrollo integrado (IDE) para Windows y admite el desarrollo en Docker.

También puede usar Visual Studio Code en macOS, Linux y Windows.
Visual Studio Code es totalmente compatible con .NET 8, incluyendo IntelliSense y la depuración. Como VS Code es un editor ligero, puede usarlo para desarrollar aplicaciones en contenedor en la máquina junto con la interfaz de la línea de comandos de Docker y la CLI de .NET. También es posible utilizar .NET 8 con la mayoría de editores de terceros, como Sublime, Emacs, vi y el proyecto OmniSharp de código abierto, que también es compatible con IntelliSense.

Además de los editores e IDE, puede utilizar la CLI de .NET en todas las plataformas admitidas.

Uso de contenedores para nuevos proyectos (green field)

Normalmente los contenedores se usan en combinación con una arquitectura de microservicios, aunque también se pueden usar para contener servicios o aplicaciones web que siguen cualquier patrón de arquitectura. Aunque se puede usar .NET Framework en contenedores de Windows, la modularidad y ligereza de .NET 8 lo convierten en la opción perfecta para los contenedores y las arquitecturas de microservicio. Al crear e implementar un contenedor, su imagen es mucho más pequeña con .NET 8 que con .NET Framework.

Creación e implementación de microservicios en contenedores

Puede utilizar el tradicional .NET Framework para crear aplicaciones basadas en microservicios (sin contenedores) mediante el uso de procesos sin formato. De este modo, como .NET Framework ya está instalado y se comparte entre procesos, los procesos son ligeros y rápidos al iniciarse. Pero, si usa contenedores, la imagen del

tradicional .NET Framework también se basa en Windows Server Core y esto hace que sea demasiado pesada para una opción de microservicios en contenedores. Sin embargo, los equipos también han estado buscando oportunidades para mejorar la experiencia de los usuarios de .NET Framework. Recientemente, el tamaño de las imágenes de contenedor de Windows Server Core se ha >reducido un 40 % .

Por otra parte, .NET 8 es la mejor opción si adoptó un sistema orientado a microservicios basado en contenedores, puesto que .NET 8 es ligero. Además, sus imágenes de contenedor relacionadas, ya sean de Linux o de Windows Nano Server, son eficientes y pequeñas, lo que hace que los contenedores sean ligeros y rápidos al iniciarse.

Un microservicio está pensado para ser lo más pequeño posible: que sea ligero al acelerar, que tenga una superficie pequeña, que tenga un pequeño contexto limitado (comprobar DDD, diseño basado en dominios), que represente una pequeña área de problemas y que se pueda iniciar y detener rápidamente. Para cumplir con estos requisitos, le recomendamos que utilice imágenes de contenedor pequeñas y que permitan crear una instancia rápidamente, como la imagen de contenedor de .NET 8.

Una arquitectura de microservicios también le permite mezclar tecnologías en un límite de servicio. Este enfoque permite migrar gradualmente a .NET 8 microservicios nuevos que funcionen junto con otros microservicios o con servicios desarrollados con Node.js, Python, Java, GoLang u otras tecnologías.

Implementación de alta densidad en sistemas escalables

Cuando en un sistema basado en contenedores se necesita la mejor densidad, granularidad y rendimiento posibles, .NET y ASP.NET Core son las mejores opciones. ASP.NET Core es hasta 10 veces más rápido que ASP.NET en el componente .NET Framework tradicional y conduce a otras tecnologías conocidas del sector de los microservicios, como servlets de Java, Go y Node.js.

Este enfoque es especialmente importante para las arquitecturas de microservicios, donde podría tener cientos de microservicios (contenedores) en funcionamiento. Con imágenes de ASP.NET Core (basadas en el entorno de ejecución de .NET) en Linux o Windows Nano, puede ejecutar el sistema con un número mucho menor de servidores o máquinas virtuales, con lo que, en última instancia, ahorra en costos de infraestructura y hospedaje.

Anterior Siguiente

Cuándo elegir .NET Framework para contenedores de Docker

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Aunque .NET 8 ofrece ventajas significativas para las nuevas aplicaciones y patrones de aplicación, .NET Framework seguirá siendo una buena opción para muchos escenarios existentes.

Migración de aplicaciones existentes directamente a un contenedor de Windows Server

Es posible que quiera usar contenedores de Docker solo para simplificar la implementación, aunque no cree microservicios. Por ejemplo, quizás quiera mejorar el flujo de trabajo de DevOps con Docker: los contenedores pueden proporcionar entornos de prueba aislados mejores y también pueden eliminar los problemas de implementación causados por dependencias que faltan al pasar a un entorno de producción. En casos como estos, incluso si va a implementar una aplicación monolítica, tiene sentido usar Contenedores de Docker y Windows para las aplicaciones actuales de .NET Framework.

En la mayoría de los casos de este escenario, no tendrá que migrar las aplicaciones existentes a .NET 8; Puede usar contenedores de Docker que incluyan .NET Framework tradicional. Sin

embargo, un enfoque recomendado es usar .NET 8 a medida que extiende una aplicación existente, como escribir un nuevo servicio en ASP.NET Core.

Uso de bibliotecas de .NET de terceros o paquetes NuGet no disponibles para .NET 8

Las bibliotecas de terceros adoptan rápidamente .NET Standard, lo que permite el uso compartido de código en todos los tipos de .NET, incluido .NET 8. Con .NET Standard 2.0 y versiones posteriores, la compatibilidad de la superficie de API en diferentes marcos se ha vuelto significativamente mayor. Además, las aplicaciones .NET Core 2.x y más recientes también pueden hacer referencia directamente a las bibliotecas existentes de .NET Framework (consulte .NET Framework 4.6.1 que admite .NET Standard 2.0).

Además, el paquete de compatibilidad de Windows amplía la superficie de API disponible para .NET Standard 2.0 en Windows. Este paquete permite volver a compilar el código existente en .NET Standard 2.x con poca o ninguna modificación, para ejecutarse en Windows.

Sin embargo, incluso con esa progresión excepcional desde .NET Standard 2.0 y .NET Core 2.1 o posterior, puede haber casos en los que determinados paquetes NuGet necesitan Windows para ejecutarse y es posible que no admitan .NET Core o posterior. Si esos paquetes son críticos para la aplicación, deberá usar .NET Framework en contenedores de Windows.

Uso de tecnologías de .NET no disponibles para .NET 8

Algunas tecnologías de .NET Framework no están disponibles en .NET 8. Algunos de ellos podrían estar disponibles en versiones posteriores, pero otros no se ajustan a los nuevos patrones de aplicación destinados a .NET Core y es posible que nunca estén disponibles.

En la lista siguiente se muestran la mayoría de las tecnologías que no están disponibles en .NET 8:

- ASP.NET Formularios web. Esta tecnología solo está disponible en .NET Framework. Actualmente no hay planes para llevar ASP.NET Web Forms a .NET o posterior.
- Servicios relacionados con el flujo de trabajo. Windows Workflow Foundation (WF), Workflow Services (WCF + WF en un solo servicio) y WCF Data Services (anteriormente conocido como ADO.NET Data Services) solo están disponibles en .NET Framework. Actualmente no hay planes para traerlos a .NET 8.

Además de las tecnologías enumeradas en la hoja de ruta oficial de .NET, se pueden migrar otras características a la nueva plataforma unificada de .NET. Es posible que considere la posibilidad de participar en las discusiones en GitHub para que su voz se pueda escuchar. Y si cree que falta algo, abra un nuevo problema en el repositorio dotnet/runtime de GitHub.

Uso de una plataforma o API que no admite .NET 8

Algunas plataformas de Microsoft y de terceros no admiten .NET 8. Por ejemplo, algunos servicios de Azure proporcionan un SDK que aún no está disponible para su consumo en .NET 8. La mayoría del SDK de Azure se deben migrar finalmente a .NET 8/.NET Standard, pero es posible que algunos no por varias razones. Puede ver los SDK de Azure disponibles en la página Versiones más recientes del SDK de Azure .

Mientras tanto, si alguna plataforma o servicio de Azure todavía no admite .NET 8 con su API de cliente, puede usar la API REST equivalente desde el servicio de Azure o el SDK de cliente en .NET Framework.

Migración de una aplicación de ASP.NET existente a .NET 8

.NET Core es un revolucionario paso adelante de .NET Framework. Ofrece una serie de ventajas sobre .NET Framework a lo largo de todo el panel, desde la productividad hasta el rendimiento, y desde el soporte multiplataforma hasta la satisfacción del desarrollador.

Recursos adicionales

- **Aspectos básicos de .NET**
  https://learn.microsoft.com/dotnet/fundamentals
- **Migración de proyectos a .NET 5**
  https://learn.microsoft.com/events/dotnetconf-2020/porting-projects-to-net-5
- **Guía de .NET en Docker**
  https://learn.microsoft.com/dotnet/core/docker/introduction

Siguiente anterior

Tabla de decisión: implementaciones de .NET que se van a usar para Docker

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En la tabla de decisión siguiente se resume si se va a usar .NET Framework o .NET 8. Recuerde que para los contenedores de Linux, necesita hosts de Docker basados en Linux (máquinas virtuales o servidores) y que para contenedores de Windows, necesita hosts de Docker basados en Windows Server (máquinas virtuales o servidores).

Importante

Las máquinas de desarrollo ejecutarán un host de Docker, ya sea Linux o Windows. Todos los microservicios relacionados que desea ejecutar y probar juntos en una solución tendrán que ejecutarse en la misma plataforma de contenedor.

Expandir tabla

Arquitectura o tipo de aplicación	Contenedores de Linux	Contenedores de Windows
Microservicios en contenedores	.NET 8	.NET 8

Arquitectura o tipo de aplicación	Contenedores de Linux	Contenedores de Windows
Aplicación monolítica	.NET 8	.NET Framework .NET 8
Rendimiento líder en su clase y escalabilidad	.NET 8	.NET 8
Migración de aplicaciones heredadas de Windows Server ("campo marrón") a contenedores	--	.NET Framework
Nuevo desarrollo basado en contenedores ("campo verde")	.NET 8	.NET 8
ASP.NET Core	.NET 8	.NET 8 (recomendado) .NET Framework
ASP.NET 4 (MVC 5, Web API 2 y Web Forms)	--	.NET Framework
Servicios de SignalR	.NET Core 2.1 o una versión posterior	.NET Framework .NET Core 2.1 o una versión posterior
WCF, WF y otros marcos heredados	WCF en .NET Core (solo biblioteca cliente) o CoreWCF	.NET Framework WCF en .NET 8 (solo biblioteca cliente) o CoreWCF
Consumo de servicios de Azure	.NET 8 (en última instancia, la mayoría de los servicios de Azure proporcionarán SDK de cliente para .NET 8)	.NET Framework .NET 8 (en última instancia, la mayoría de los servicios de Azure proporcionarán SDK de cliente para .NET 8)

Siguiente anterior

Qué sistema operativo se va a establecer como destino con contenedores de .NET

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Dada la diversidad de sistemas operativos compatibles con Docker y las diferencias entre .NET Framework y .NET 8, debe tener como destino un sistema operativo específico y versiones específicas en función del marco que use.

Para Windows, puede usar Windows Server Core o Windows Nano Server. Estas versiones de Windows proporcionan características diferentes (IIS en Windows Server Core frente a un servidor web autohospedado como Kestrel en Nano Server) que podría necesitar .NET Framework o .NET 8, respectivamente.

Para Linux, hay varias distribuciones disponibles y se admiten en imágenes oficiales de Docker de .NET (como Debian).

En la figura 3-1, puede ver la posible versión del sistema operativo en función de .NET Framework usada.

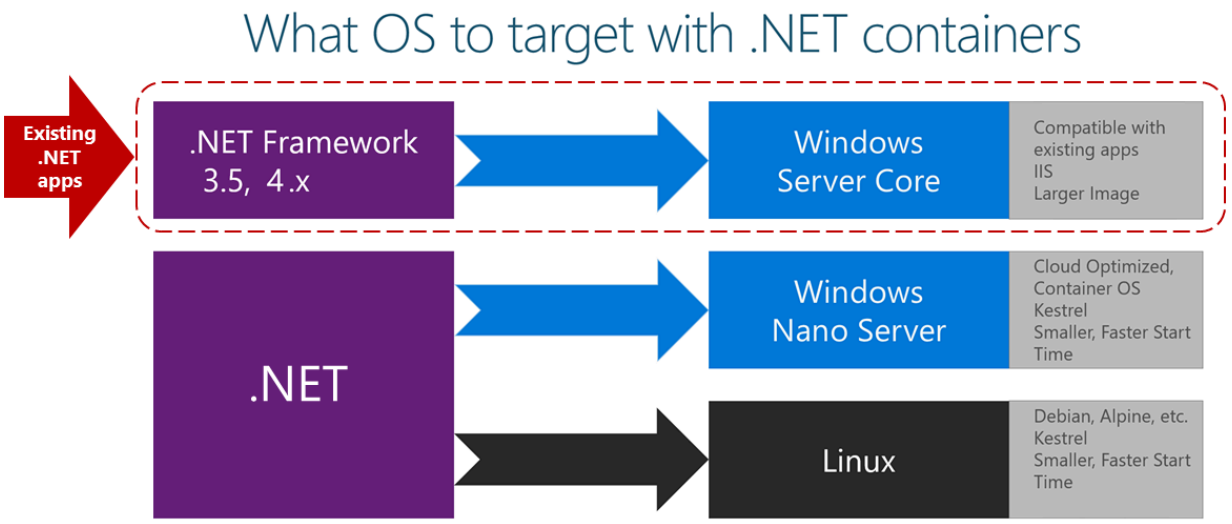


Figura 3-1. Sistemas operativos de destino en función de las versiones de .NET Framework

Al implementar aplicaciones heredadas de .NET Framework, tiene que elegir como destino Windows Server Core, que es compatible con aplicaciones heredadas e IIS, pero tiene una imagen más grande. Al implementar aplicaciones de .NET 8, puede tener como destino Windows Nano Server, que está optimizado para la nube, usa Kestrel y es más pequeño e inicia más rápido. También puede tener como destino Linux, que es compatible con Debian, Alpine y otros.

También puede crear su propia imagen de Docker en los casos en los que quiera usar una distribución de Linux diferente o donde quiera una imagen con versiones no proporcionadas por Microsoft. Por ejemplo, puede crear una imagen con ASP.NET Core que se ejecuta en .NET Framework tradicional y Windows Server Core, que es un escenario no tan común para Docker.

Al agregar el nombre de la imagen al archivo Dockerfile, puede seleccionar el sistema operativo y la versión en función de la etiqueta que use, como en los ejemplos siguientes:

Expandir tabla

Imagen	Comentarios
mcr.microsoft.com/dotnet/runtime:8.0	Arquitectura múltiple de .NET 8: admite Linux y Windows Nano Server en función del host de Docker.
mcr.microsoft.com/dotnet/aspnet:8.0	La arquitectura múltiple de ASP.NET Core 8.0: es compatible con Linux y Windows Nano Server en función del host de Docker.

Imagen	Comentarios
	La imagen aspnetcore tiene algunas optimizaciones para ASP.NET Core.
mcr.microsoft.com/dotnet/aspnet:8.0-bullseye-slim	Solo entorno de ejecución de .NET 8 en una distribución de Linux Debian
mcr.microsoft.com/dotnet/aspnet:8.0-nanoserver-1809	Entorno de ejecución exclusivamente de .NET 8 en Windows Nano Server (versión 1809 de Windows Server)

Siguiente anterior

Imágenes oficiales de Docker de .NET

Artículo • 15/10/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en **Documentación de .NET** o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Las imágenes oficiales de Docker de .NET son imágenes de Docker que Microsoft ha creado y optimizado. Están disponibles públicamente en el Registro de artefactos Microsoft. Puede buscar en el catálogo para encontrar todos los repositorios de imágenes de .NET, por ejemplo, el repositorio del SDK de .NET.

Cada repositorio puede contener varias imágenes, según las versiones de .NET y según el sistema operativo y las versiones (Linux Debian, Linux Alpine, Windows Nano Server, Windows Server Core, etc.). Los repositorios de imágenes ofrecen un etiquetado exhaustivo con el que es más fácil elegir no solo la versión de un marco concreto, sino también un sistema operativo (distribución de Linux o versión de Windows).

Optimizaciones de imágenes de .NET y Docker para desarrollo frente a para producción

Al compilar imágenes de Docker para desarrolladores, Microsoft se centró en los siguientes escenarios principales:

- Imágenes que se usan para *desarrollar* y compilar aplicaciones de .NET.
- Imágenes que se usan para *ejecutar* aplicaciones de .NET.

¿Por qué varias imágenes? Al desarrollar, compilar y ejecutar aplicaciones en contenedor, normalmente hay prioridades diferentes. Al proporcionar diferentes imágenes para estas tareas independientes, con Microsoft es más fácil optimizar los procesos independientes de desarrollo, creación e implementación de aplicaciones.

Durante el desarrollo y la compilación

Durante el desarrollo, lo importante es la velocidad con que se pueden iterar los cambios y la capacidad para depurar los cambios. El tamaño de la imagen no es tan importante como la capacidad de realizar cambios en el código y ver rápidamente los cambios. Algunas herramientas y "contenedores de agente de compilación" usan la imagen de .NET de desarrollo (*mcr.microsoft.com/dotnet/sdk:8.0*) durante el proceso de desarrollo y compilación. Al compilar dentro de un contenedor de Docker, los aspectos importantes son los elementos necesarios para compilar la aplicación. Esto incluye el compilador y cualquier otra dependencia de .NET.

Otra opción excelente son los contenedores de desarrollo. Estos contenedores son entornos de desarrollo precompilados que están listos para usarse; no tiene que preocuparse por las dependencias y las configuraciones. También se pueden personalizar fácilmente para incluir herramientas o dependencias adicionales. Los contenedores de desarrollo proporcionan una configuración coherente y reproducible que es fácil de compartir con su equipo. Los contenedores de desarrollo se ajustan a la especificación de contenedor de desarrollo y muchas herramientas de desarrollo populares, como Visual Studio Code y GitHub Codespaces, las admiten. Los contenedores de desarrollo de .NET se basan en la imagen del SDK de .NET e incluyen el SDK de .NET, el entorno de ejecución y otras herramientas que necesita para desarrollar aplicaciones .NET.

¿Por qué es importante este tipo de imagen de compilación? Esta imagen no se implementa para producción, sino que es una imagen que se usa para compilar el contenido que se coloca en una imagen de producción. Esta imagen se usaría en el entorno de integración continua (CI) o el entorno de compilación al utilizar compilaciones de Docker de varias fases.

En producción

Lo importante en producción es la rapidez con la que se pueden implementar e iniciar los contenedores según una imagen de .NET de producción. Por tanto, la imagen solo en entorno de ejecución basada en *mcr.microsoft.com/dotnet/aspnet:8.0* es pequeña, de modo que puede viajar rápidamente a través de la red desde el registro de Docker hasta los hosts de Docker. El contenido está listo para ejecutarse, lo que agiliza el proceso que

va desde iniciar el contenedor hasta procesar los resultados. En el modelo de Docker, no es necesario compilar desde el código C#, como cuando se ejecuta dotnet build o dotnet publish al usar el contenedor de compilación.

En esta imagen optimizada solo se colocan los archivos binarios y otros contenidos necesarios para ejecutar la aplicación. Por ejemplo, el contenido que crea `dotnet publish` solo contiene los archivos binarios de .NET compilados, las imágenes y los archivos .js y .css. Con el tiempo, verá imágenes que contienen paquetes anteriores a la compilación (la compilación del lenguaje intermedio al nativo que se produce en tiempo de ejecución).

Aunque hay varias versiones de las imágenes de .NET y ASP.NET Core, todas ellas comparten una o más capas, incluida la capa base. Por tanto, la cantidad de espacio en disco necesaria para almacenar una imagen es pequeña; consiste únicamente en las diferencias entre la imagen personalizada y su imagen base. El resultado es que es rápido extraer la imagen desde el Registro.

Al explorar los repositorios de imágenes de .NET en el Registro de artefactos Microsoft, encontrará varias versiones de imágenes clasificadas o marcadas con etiquetas. Estas etiquetas ayudan a decidir cuál usar, según la versión que necesite, como las de la tabla siguiente:

Expandir tabla

Imagen	Comentarios
mcr.microsoft.com/dotnet/aspnet:**8.0**	ASP.NET Core, solo con tiempo de ejecución y las optimizaciones de ASP.NET Core, en Linux y Windows (multiarquitectura)
mcr.microsoft.com/dotnet/sdk:**8.0**	.NET 8, con los SDK incluidos, en Linux y Windows (arquitectura múltiple)

Puede encontrar todas las imágenes de Docker disponibles en dotnet-docker y también puede consultar las versiones preliminares más recientes mediante la compilación nocturna `mcr.microsoft.com/dotnet/nightly/*`.

Anterior Siguiente

Diseño de aplicaciones basadas en contenedores y microservicios

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Los microservicios ofrecen grandes ventajas, pero también generan grandes desafíos nuevos. Los patrones de arquitectura de microservicios son pilares fundamentales al crear una aplicación basada en microservicios.

Anteriormente en esta guía, ha aprendido conceptos básicos sobre contenedores y Docker. Esa información era el mínimo necesario para empezar a trabajar con contenedores. Aunque los contenedores son habilitadores de y una excelente opción para los microservicios, no son obligatorios para una arquitectura de microservicios. Muchos conceptos arquitectónicos de esta sección de arquitectura se podrían aplicar sin contenedores. Sin embargo, esta guía se centra en la intersección de ambos debido a la importancia ya introducida de los contenedores.

Las aplicaciones empresariales pueden ser complejas y a menudo se componen de varios servicios en lugar de una sola aplicación basada en servicios. Para esos casos, debe comprender otros enfoques arquitectónicos, como los microservicios y ciertos patrones de diseño de Domain-Driven (DDD), además de conceptos de orquestación de contenedores. Tenga en cuenta que en este capítulo se describen no solo microservicios en contenedores, sino también cualquier aplicación en contenedor.

Principios de diseño de contenedores

En el modelo de contenedor, una instancia de imagen de contenedor representa un único proceso. Al definir una imagen de contenedor como límite de proceso, puede crear primitivos que se pueden usar para escalar o procesar por lotes el proceso.

Al diseñar una imagen de contenedor, visualizarás una definición ENTRYPOINT en el Dockerfile. Esta definición define el proceso cuya duración controla la duración del contenedor. Cuando se completa el proceso, finaliza el ciclo de vida del contenedor. Los contenedores pueden representar procesos de ejecución prolongada como servidores web, pero también pueden representar procesos de corta duración, como trabajos por lotes, que anteriormente podrían haberse implementado como Azure WebJobs.

Si se produce un error en el proceso, el contenedor finaliza y lo sustituye el orquestador. Si el orquestador se configuró para mantener cinco instancias en ejecución y se produce un error, el orquestador creará otra instancia de contenedor para reemplazar el proceso con errores. En un trabajo por lotes, el proceso se inicia con parámetros. Cuando el proceso finaliza, el trabajo está terminado. Más adelante en esta guía se exploran en profundidad los orquestadores.

Es posible que encuentre un escenario en el que quiera que varios procesos se ejecuten en un único contenedor. En ese escenario, dado que solo puede haber un punto de entrada por contenedor, podría ejecutar un script dentro del contenedor que inicie tantos programas como sea necesario. Por ejemplo, puede usar Supervisor o una herramienta similar para encargarse de iniciar varios procesos dentro de un único contenedor. Sin embargo, aunque puede encontrar arquitecturas que contienen varios procesos por contenedor, ese enfoque no es muy común.

Siguiente anterior

Contenedorización de aplicaciones monolíticas

Artículo • 27/05/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Es posible que quiera compilar una única aplicación web o un servicio web implementados de forma monolítica e implementarla como un contenedor. Es posible que la propia aplicación no sea monolítica internamente, pero estructurada como varias bibliotecas, componentes o incluso capas (capa de aplicación, capa de dominio, capa de acceso a datos, etc.). Sin embargo, externamente es un único contenedor: un único proceso, una sola aplicación web o un único servicio.

Para administrar este modelo, implemente un único contenedor para representar la aplicación. Para aumentar la capacidad, escale horizontalmente, es decir, agregue más copias con un equilibrador de carga delante. La simplicidad procede de administrar una sola implementación en un único contenedor o máquina virtual.

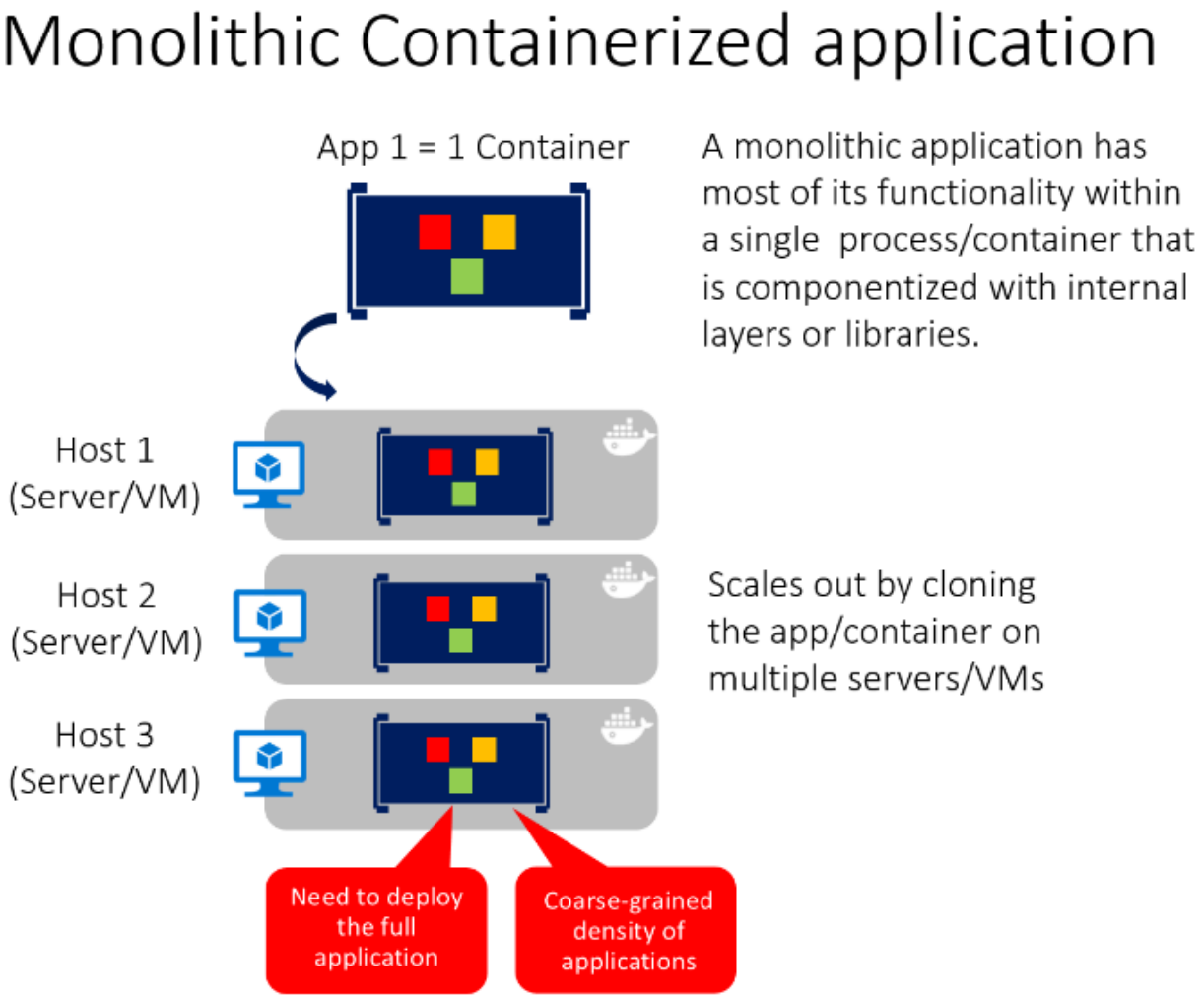


Figura 4-1. Ejemplo de la arquitectura de una aplicación monolítica en contenedor

Puede incluir varios componentes, bibliotecas o capas internas en cada contenedor, como se muestra en la figura 4-1. Una aplicación en contenedor monolítica tiene la mayor parte de su funcionalidad dentro de un único contenedor, con capas internas o bibliotecas, y se escala horizontalmente mediante la clonación del contenedor en varios servidores o máquinas virtuales. Sin embargo, este patrón monolítico podría entrar en conflicto con el principio de contenedor "un contenedor hace una cosa y lo hace en un proceso", pero podría ser correcto para algunos casos.

La desventaja de este enfoque es evidente si la aplicación crece, lo que requiere que se escale. Si toda la aplicación puede escalar, no es realmente un problema. Sin embargo, en la mayoría de los casos, solo algunas partes de la aplicación son los puntos de ahogamiento que requieren escalado, mientras que otros componentes se usan menos.

Por ejemplo, en una aplicación típica de comercio electrónico, es probable que tenga que escalar el subsistema de información del producto, ya que muchos más clientes examinan productos que comprarlos. Más clientes usan su cesta que usar la canalización de pago. Menos clientes agregan comentarios o ven su historial de compras. Y es posible que solo tenga unos pocos empleados que necesiten administrar el contenido y las campañas de marketing. Si

escala el diseño monolítico, todo el código de estas diferentes tareas se despliega varias veces y se escala al mismo nivel.

Hay varias maneras de escalar una duplicación horizontal de la aplicación, dividir diferentes áreas de la aplicación y crear particiones de datos o conceptos empresariales similares. Pero, además del problema de escalar todos los componentes, los cambios en un único componente requieren una nueva prueba completa de toda la aplicación y una reimplementación completa de todas las instancias.

Sin embargo, el enfoque monolítico es común, ya que el desarrollo de la aplicación es inicialmente más fácil que para los enfoques de microservicios. Por lo tanto, muchas organizaciones desarrollan el uso de este enfoque arquitectónico. Aunque algunas organizaciones han tenido buenos resultados, otros están alcanzando límites. Muchas organizaciones diseñaron sus aplicaciones con este modelo porque las herramientas y la infraestructura hicieron que fuera demasiado difícil compilar arquitecturas orientadas a servicios (SOA) hace años, y no vieron la necesidad hasta que creció la aplicación.

Desde la perspectiva de la infraestructura, cada servidor puede ejecutar muchas aplicaciones dentro del mismo host y tener una proporción aceptable de eficiencia en el uso de recursos, como se muestra en la figura 4-2.

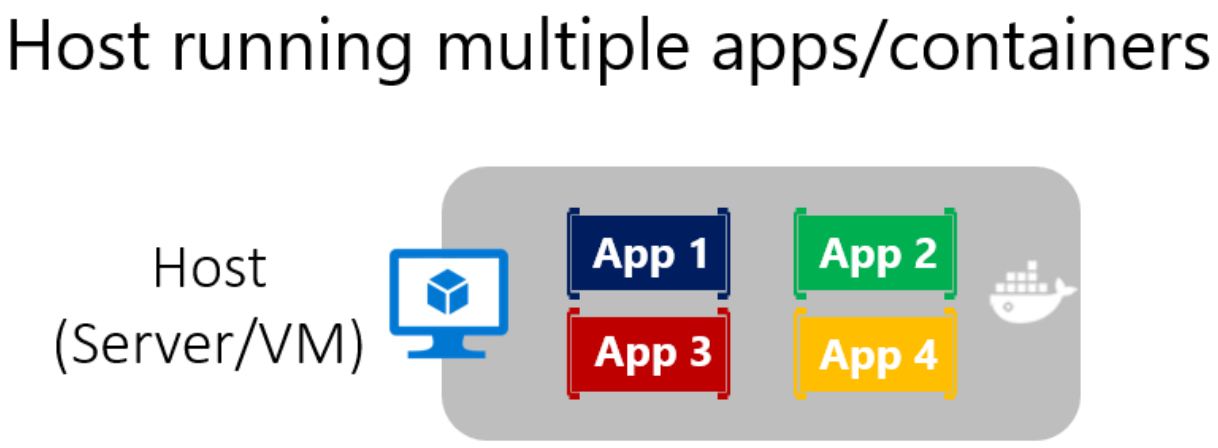


Figura 4-2. Enfoque monolítico: host que ejecuta varias aplicaciones, cada aplicación que se ejecuta como un contenedor

Las aplicaciones monolíticas de Microsoft Azure se pueden implementar mediante máquinas virtuales dedicadas para cada instancia. Además, mediante conjuntos de escalado de máquinas virtuales de Azure, puede escalar fácilmente las máquinas virtuales. Azure App Service también puede ejecutar aplicaciones monolíticas y escalar fácilmente instancias sin necesidad

de administrar las máquinas virtuales. Desde 2016, Azure App Services también puede ejecutar instancias únicas de contenedores de Docker, lo que simplifica la implementación.

Como entorno de control de calidad o un entorno de producción limitado, puede implementar varias máquinas virtuales host de Docker y equilibrarlas con el equilibrador de Azure, como se muestra en la figura 4-3. Esto le permite administrar el escalado con un enfoque general, ya que toda la aplicación reside dentro de un único contenedor.

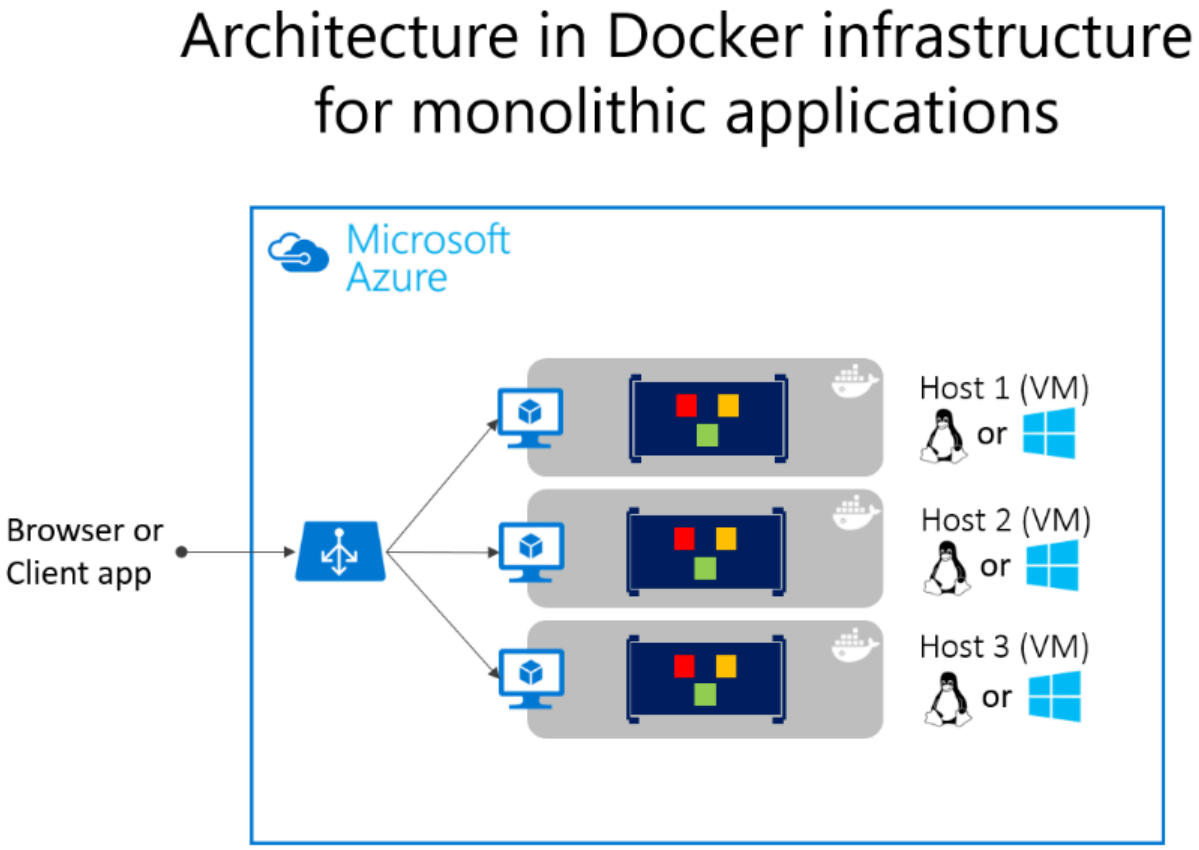


Figura 4-3. Ejemplo de varios hosts que escalan verticalmente una sola aplicación contenedora

La implementación en los distintos hosts se puede administrar con técnicas de implementación tradicionales. Los hosts de Docker se pueden administrar con comandos como `docker run` o `docker-compose`, realizados manualmente, o mediante la automatización, como las canalizaciones de entrega continua (CD).

Implementación de una aplicación monolítica como contenedor

Existen ventajas para usar contenedores para administrar implementaciones de aplicaciones monolíticas. El escalado de instancias de contenedor es mucho más rápido y sencillo que la implementación de máquinas virtuales adicionales. Incluso si usa conjuntos de escalado de máquinas virtuales, las máquinas virtuales tardan tiempo en iniciarse. Cuando se implementa

como instancias de aplicación tradicionales en lugar de contenedores, la configuración de la aplicación se administra como parte de la máquina virtual, lo que no es ideal.

La implementación de actualizaciones como imágenes de Docker es mucho más rápida y eficiente en cuanto a la red. Las imágenes de Docker normalmente se inician en segundos, lo que acelera los despliegues. Eliminar una instancia de imagen de Docker es tan fácil como emitir un comando `docker stop` y normalmente se completa en menos de un segundo.

Dado que los contenedores son inmutables por diseño, nunca debe preocuparse por las máquinas virtuales dañadas. Por el contrario, es posible que los scripts de actualización de una máquina virtual se olviden de tener en cuenta alguna configuración específica o archivo dejado en el disco.

Aunque las aplicaciones monolíticas pueden beneficiarse de Docker, solo mencionamos las ventajas. Las ventajas adicionales de administrar contenedores proceden de la implementación con orquestadores de contenedores, que administran las distintas instancias y el ciclo de vida de cada instancia de contenedor. Dividir la aplicación monolítica en subsistemas que se pueden escalar, desarrollar e implementar individualmente es el punto de entrada en el dominio de los microservicios.

Publicación de una aplicación basada en un solo contenedor en Azure App Service

Tanto si quiere obtener la validación de un contenedor implementado en Azure como si una aplicación es simplemente una aplicación de contenedor único, Azure App Service proporciona una excelente manera de proporcionar servicios escalables basados en contenedores únicos. El uso de Azure App Service es sencillo. Proporciona una excelente integración con Git para facilitar la toma del código, compilarlo en Visual Studio e implementarlo directamente en Azure.

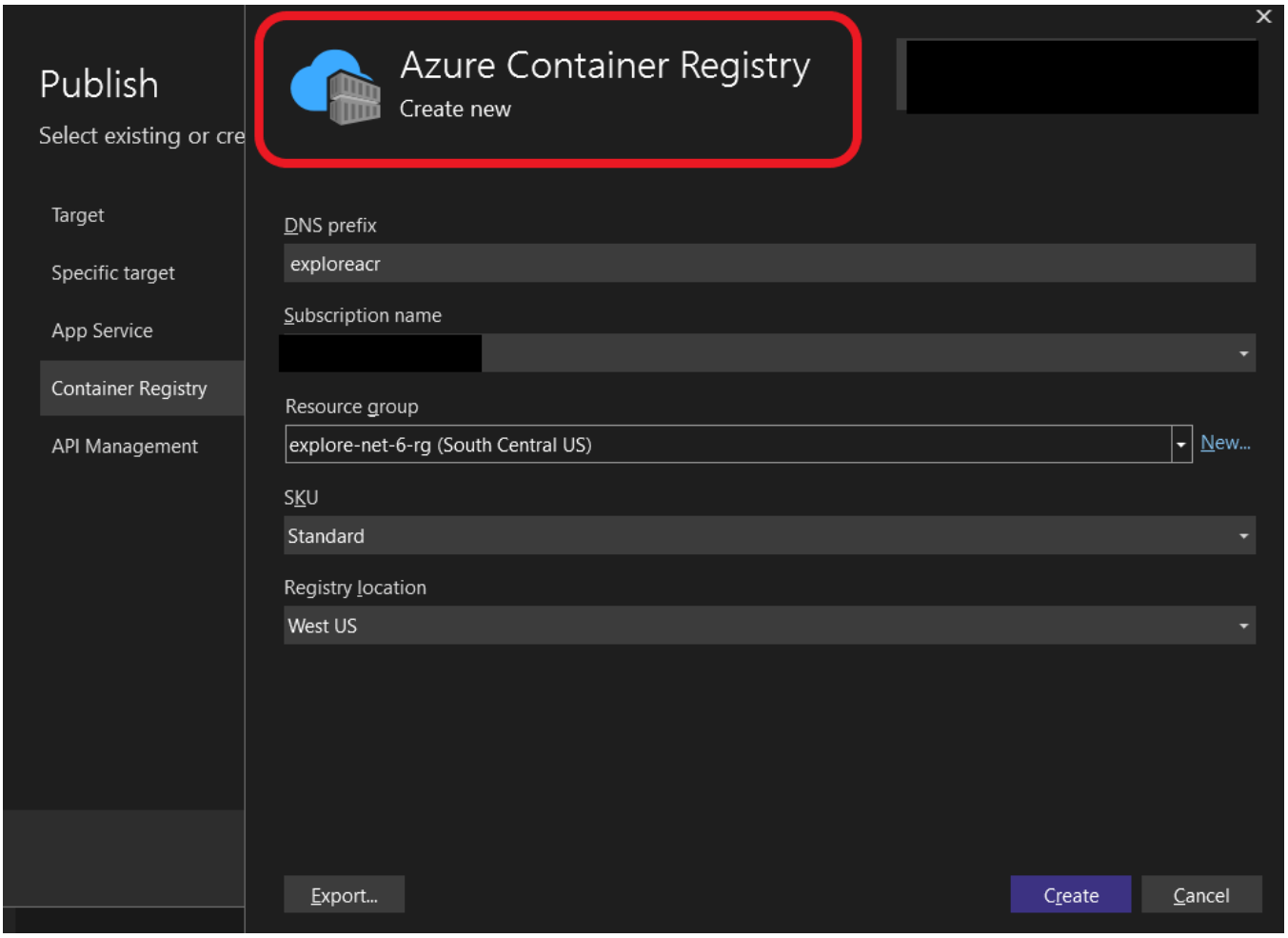


Figura 4-4. Publicación de una aplicación de contenedor único en Azure App Service desde Visual Studio 2022

Sin Docker, si necesitaba otras funcionalidades, marcos o dependencias que no se admiten en Azure App Service, tenía que esperar hasta que el equipo de Azure actualizara esas dependencias en App Service. O bien, tenía que cambiar a otros servicios, como Azure Cloud Services o máquinas virtuales, donde tenía más control y podía instalar un componente o marco necesarios para la aplicación.

La compatibilidad con contenedores en Visual Studio 2017 y versiones posteriores le ofrece la posibilidad de incluir lo que desee en el entorno de la aplicación, como se muestra en la figura 4-4. Como lo está ejecutando en un contenedor, si agrega una dependencia a la aplicación, puede incluir la dependencia en el Dockerfile o en la imagen de Docker.

Como también se muestra en la figura 4-4, el flujo de publicación inserta una imagen a través de un registro de contenedor. Puede ser Azure Container Registry (un registro cercano a las implementaciones en Azure y protegidas por grupos y cuentas de Azure Active Directory) o cualquier otro registro de Docker, como Docker Hub o un registro local.

Siguiente anterior

Administración de estado y datos en aplicaciones de Docker

Artículo • 10/05/2023

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en **Documentación de .NET** o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En la mayoría de los casos, un contenedor se puede considerar como una instancia de un proceso. Un proceso no mantiene un estado persistente. Si bien un contenedor puede escribir en su almacenamiento local, suponer que una instancia permanecerá indefinidamente sería como suponer que una sola ubicación en memoria será duradera. Debe suponer que las imágenes de contenedor, como los procesos, tienen varias instancias o finalmente se terminarán. Si se administran con un orquestador de contenedores, debe suponer que podrían desplazarse de un nodo o máquina virtual a otro.

Las soluciones siguientes se usan para administrar datos en aplicaciones de Docker:

Desde el host de Docker, como volúmenes de Docker:

- Los **volúmenes** se almacenan en un área del sistema de archivos de host administrado por Docker.
- Los **montajes de enlace** pueden asignar cualquier carpeta en el sistema de archivos de host, por lo que el acceso no se puede controlar desde el proceso de

Docker y puede suponer un riesgo de seguridad ya que un contenedor podría acceder a carpetas del sistema operativo confidenciales.

- Los **montajes tmpfs** son como carpetas virtuales que solo existen en la memoria del host y nunca se escriben en el sistema de archivos.

Desde el almacenamiento remoto:

- Azure Storage, que proporciona almacenamiento con distribución geográfica y representa una buena solución de persistencia a largo plazo para los contenedores.

- Bases de datos relacionales remotas como Azure SQL Database, bases de datos NoSQL como Azure Cosmos DB o servicios de caché como Redis.

Desde el contenedor de Docker:

- **Superposición del sistema de archivos**. Esta característica de Docker implementa una tarea de copia en escritura que almacena información actualizada en el sistema de archivos raíz del contenedor. Esa información se coloca "encima" de la imagen original en la que se basa el contenedor. Si se elimina el contenedor del sistema, estos cambios se pierden. Por tanto, si bien es posible guardar el estado de un contenedor dentro de su almacenamiento local, diseñar un sistema sobre esta base entraría en conflicto con la idea del diseño del contenedor, que de manera predeterminada es sin estado.

Sin embargo, el uso de los volúmenes de Docker es ahora la mejor manera de controlar datos locales en Docker. Si necesita obtener más información sobre el almacenamiento en contenedores, consulte Docker storage drivers (Controladores de almacenamiento de Docker) y About storage drivers (Sobre los controladores de almacenamiento).

Los siguientes puntos proporcionan más información sobre estas opciones:

Los **volúmenes** son directorios asignados desde el sistema operativo del host a directorios en contenedores. Cuando el código en el contenedor tiene acceso al directorio, ese acceso es realmente a un directorio en el sistema operativo del host. Este directorio no está asociado a la duración del contenedor y Docker lo administra y aísla de la funcionalidad básica de la máquina host. Por tanto, los volúmenes de datos están diseñados para conservar los datos independientemente de la vida del contenedor. Si elimina un contenedor o una imagen del host de Docker, los datos que se conservan en el volumen de datos no se eliminan.

Los volúmenes pueden tener nombre o ser anónimos (predeterminado). Los volúmenes con nombre son la evolución de los **Contenedores de volúmenes de datos** y facilitan el

uso compartido de datos entre contenedores. Los volúmenes también admiten controladores de volumen, que le permiten almacenar datos en hosts remotos, entre otras opciones.

Los **montajes de enlace** están disponibles desde hace mucho tiempo y permiten la asignación de cualquier carpeta a un punto de montaje en un contenedor. Los montajes de enlace tienen más limitaciones que los volúmenes y algunos problemas de seguridad importantes, por lo que los volúmenes son la opción recomendada.

Los **montajes tmpfs** son básicamente carpetas virtuales que solo existen en la memoria del host y nunca se escriben en el sistema de archivos. Son rápidos y seguros, pero usan memoria y solo están diseñados para datos temporales y no persistentes.

Tal como se muestra en la figura 4-5, los volúmenes de Docker normales pueden almacenarse fuera de los propios contenedores, pero dentro de los límites físicos del servidor de host o de la máquina virtual. Pero los contenedores de Docker no pueden acceder a un volumen desde un servidor de host o máquina virtual a otro. En otras palabras, con estos volúmenes, no es posible administrar los datos que se comparten entre contenedores que se ejecutan en otros hosts de Docker, aunque se podría lograr con un controlador de volumen que sea compatible con los hosts remotos.

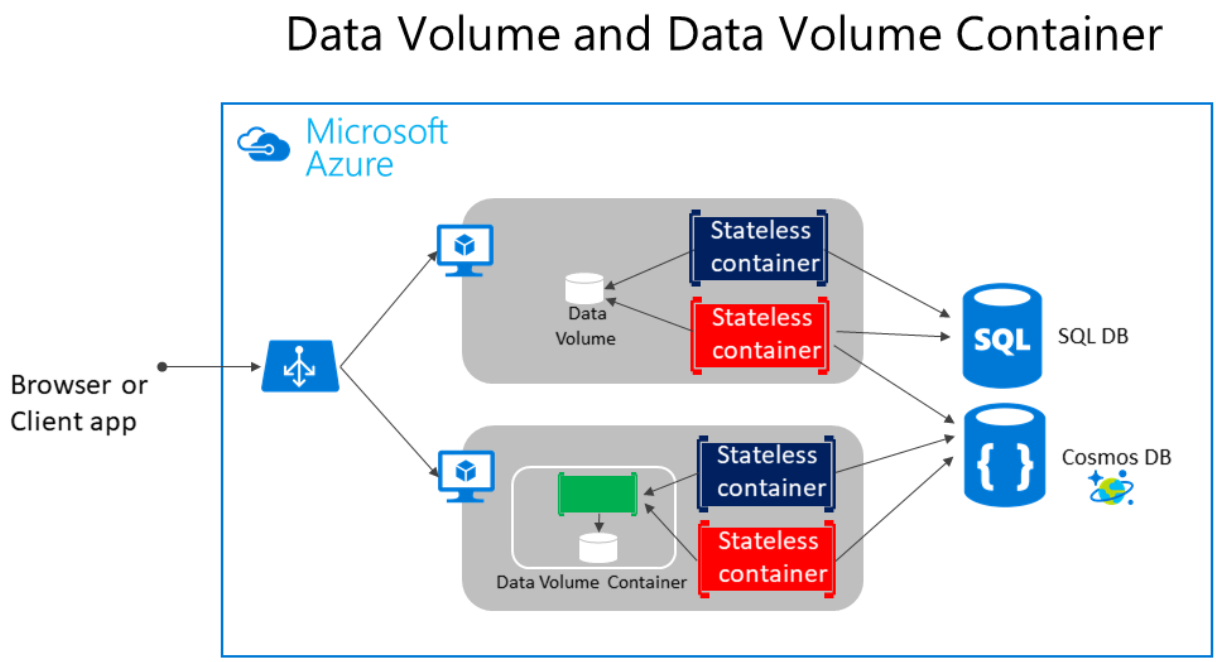


Figura 4-5. Volúmenes y orígenes de datos externos para aplicaciones basadas en contenedor

Los volúmenes se pueden compartir entre contenedores, pero solo en el mismo host, a menos que use un controlador remoto compatible con hosts remotos. Además, cuando un orquestador administra los contenedores de Docker, estos podrían "moverse" entre hosts, según las optimizaciones que el clúster realice. Por tanto, no se recomienda usar

volúmenes de datos para los datos empresariales. Pero son un buen mecanismo para trabajar con archivos de seguimiento, archivos temporales o similares que no afectarán a la coherencia de los datos empresariales.

Las herramientas de **orígenes de datos remotos y caché** como Azure SQL Database, Azure Cosmos DB o una caché remota como Redis pueden usarse en aplicaciones en contenedores del mismo modo que se usan al desarrollar sin contenedores. Se trata de una manera comprobada para almacenar datos de aplicaciones empresariales.

Azure Storage. Normalmente, los datos empresariales deben colocarse en recursos o bases de datos externos, como Azure Storage. Azure Storage, en concreto, proporciona los siguientes servicios en la nube:

- Blob Storage almacena datos de objetos no estructurados. Un blob puede ser cualquier tipo de texto o datos binarios, como documentos o archivos multimedia (archivos de imagen, audio y vídeo). El almacenamiento de blobs a veces se conoce también como almacenamiento de objetos.
- El almacenamiento de archivos ofrece almacenamiento compartido para aplicaciones heredadas mediante el protocolo SMB estándar. Las máquinas virtuales de Azure y los servicios en la nube pueden compartir datos de archivos en los componentes de la aplicación a través de recursos compartidos montados. Las aplicaciones locales pueden tener acceso a datos de archivos en un recurso compartido a través de la API de REST de servicio de archivos.
- El almacenamiento de tablas almacena conjuntos de datos estructurados. El almacenamiento de tabla es un almacén de datos del atributo de clave NoSQL, lo que permite desarrollar y acceder rápidamente a grandes cantidades de datos.

Bases de datos relacionales y bases de datos NoSQL. Hay muchas opciones para bases de datos externas, desde bases de datos relacionales como SQL Server, PostgreSQL u Oracle hasta bases de datos NoSQL como Azure Cosmos DB, MongoDB, etc. Estas bases de datos no se van a explicar en esta guía porque pertenecen a un tema completamente diferente.

Anterior Siguiente

Arquitectura orientada a servicios

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

La arquitectura orientada a servicios (SOA) era un término sobreutilizado y ha significado diferentes cosas para diferentes personas. Pero como denominador común, SOA significa que la aplicación se estructura descomponiendo en varios servicios (normalmente como servicios HTTP) que se pueden clasificar como tipos diferentes, como subsistemas o niveles.

Esos servicios ahora se pueden implementar como contenedores de Docker, lo que resuelve los problemas de implementación, ya que todas las dependencias se incluyen en la imagen de contenedor. Sin embargo, cuando necesite escalar aplicaciones SOA, es posible que enfrente desafíos de escalabilidad y disponibilidad si está implementando en hosts de Docker individuales. Aquí es donde el software de agrupación en clústeres de Docker o un orquestador pueden ayudarle, como se explica en secciones posteriores en las que se describen los enfoques de implementación para los microservicios.

Los contenedores de Docker son útiles (pero no necesarios) para las arquitecturas tradicionales orientadas a servicios y las arquitecturas de microservicios más avanzadas.

Los microservicios derivan de SOA, pero SOA es diferente de la arquitectura de microservicios. Las características como los agentes centrales de gran tamaño, los orquestadores centrales en el nivel de organización y Enterprise Service Bus (ESB) son típicos en SOA. Pero en la mayoría de los casos son antipatrones en la comunidad de microservicios. De hecho, algunas personas argumentan que "La arquitectura de microservicios es SOA correcta".

Esta guía se centra en los microservicios, ya que un enfoque de SOA es menos prescriptivo que los requisitos y las técnicas que se usan en una arquitectura de microservicios. Si sabe cómo compilar una aplicación basada en microservicios, también sabe cómo crear una aplicación orientada a servicios más sencilla.

Siguiente anterior

Arquitectura de microservicios

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Como el nombre implica, una arquitectura de microservicios es un enfoque para crear una aplicación de servidor como un conjunto de servicios pequeños. Esto significa que una arquitectura de microservicios está orientada principalmente al back-end, aunque el enfoque también se usa para el front-end. Cada servicio se ejecuta en su propio proceso y se comunica con otros procesos mediante protocolos como HTTP/HTTPS, WebSockets o AMQP. Cada microservicio implementa una funcionalidad específica de dominio o negocio de un extremo a otro dentro de un determinado límite de contexto, y cada uno debe desarrollarse de forma autónoma e implementarse de forma independiente. Por último, cada microservicio debe poseer su modelo de datos de dominio relacionado y lógica de dominio (soberanía y administración descentralizada de datos) y podría basarse en diferentes tecnologías de almacenamiento de datos (SQL, NoSQL) y diferentes lenguajes de programación.

¿Qué tamaño debe tener un microservicio? Al desarrollar un microservicio, el tamaño no debe ser el punto importante. En su lugar, el punto importante debe ser crear servicios acoplados flexiblemente para que tenga autonomía de desarrollo, implementación y escalado para cada servicio. Por supuesto, al identificar y diseñar microservicios, debe intentar hacerlos lo más pequeños posible siempre que no tenga demasiadas dependencias directas con otros microservicios. Más importante que el tamaño del microservicio es la cohesión interna que debe tener y su independencia de otros servicios.

¿Por qué una arquitectura de microservicios? En resumen, proporciona agilidad a largo plazo. Los microservicios permiten una mejor capacidad de mantenimiento en sistemas complejos, grandes y altamente escalables, ya que permiten crear aplicaciones basadas en muchos servicios que se pueden implementar de forma independiente que cada uno tiene ciclos de vida pormenorizados y autónomos.

Como ventaja adicional, los microservicios se pueden escalar horizontalmente de forma independiente. En lugar de disponer de una sola aplicación monolítica que debe escalar horizontalmente como una unidad, puede escalar horizontalmente microservicios concretos. De este modo, puede escalar solo el área funcional que necesita más potencia de procesamiento o ancho de banda de red para admitir la demanda, en lugar de escalar horizontalmente otras áreas de la aplicación que no es necesario escalar. Así, puede ahorrar en costes porque necesita menos hardware.

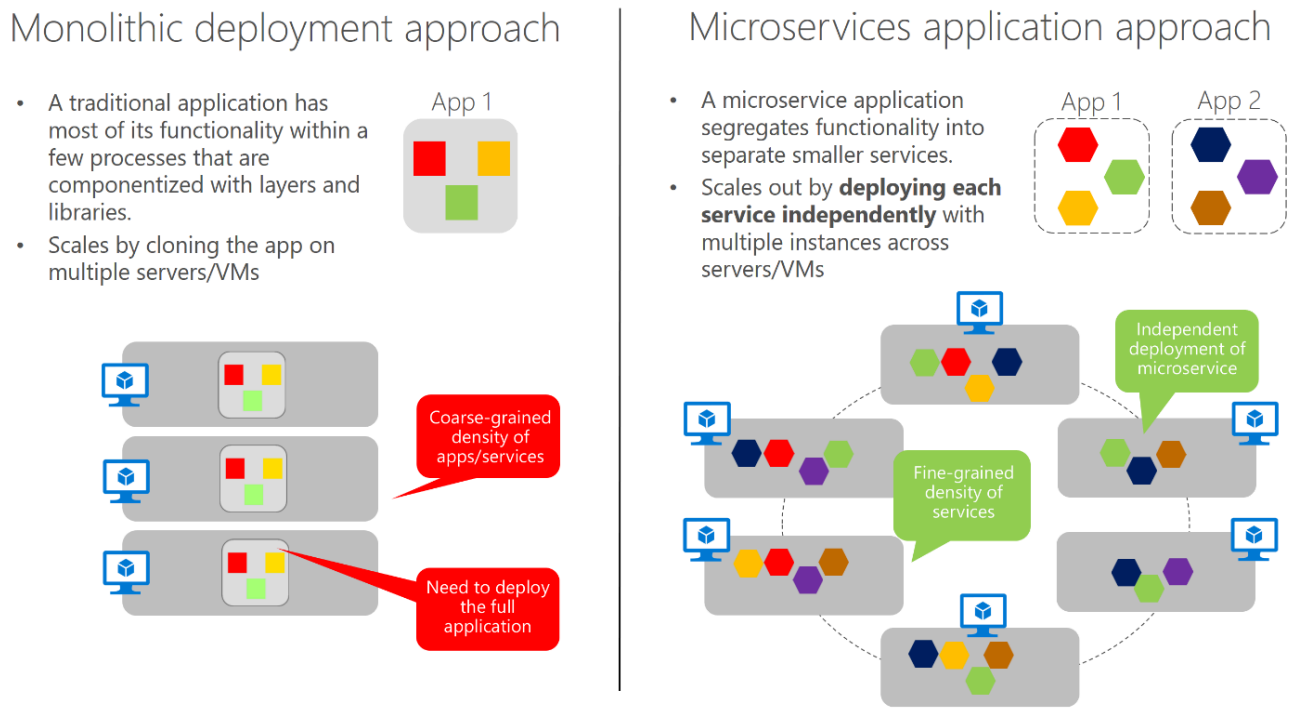


Figura 4-6. Implementación monolítica frente al enfoque de microservicios

Como se muestra en la figura 4-6, en el enfoque monolítico tradicional, la aplicación se escala mediante la clonación de toda la aplicación en varios servidores o máquinas virtuales. En el enfoque de microservicios, la funcionalidad se separa en servicios más pequeños, por lo que cada servicio puede escalar de forma independiente. El enfoque de microservicios permite cambios ágiles y iteración rápida de cada microservicio, ya que puede cambiar áreas específicas y pequeñas de aplicaciones complejas, grandes y escalables.

Diseñar la arquitectura de aplicaciones específicas basadas en microservicios habilita una integración continua y prácticas de entrega continua. También acelera la entrega de nuevas funciones en la aplicación. La composición específica de las aplicaciones también le permite ejecutar y probar microservicios de forma aislada y desarrollarlos de forma autónoma al

tiempo que mantiene contratos claros entre ellas. Siempre y cuando no cambie las interfaces o los contratos, puede cambiar la implementación interna de cualquier microservicio o agregar nuevas funciones sin que ello interrumpa otros microservicios.

A continuación se muestran aspectos importantes para permitir el éxito en la producción con un sistema basado en microservicios:

- Monitoreo y verificaciones de salud de los servicios y la infraestructura.
- Infraestructura escalable para los servicios (es decir, en la nube y orquestadores).
- Diseño e implementación de seguridad en varios niveles: autenticación, autorización, administración de secretos, comunicación segura, etc.
- Entrega rápida de aplicaciones, normalmente con diferentes equipos centrados en diferentes microservicios.
- Infraestructura y prácticas de DevOps y CI/CD.

De estos, solo se tratan o introducen los tres primeros en esta guía. Los dos últimos puntos, que están relacionados con el ciclo de vida de la aplicación, se tratan en el e-book adicional sobre el ciclo de vida de las aplicaciones Docker en contenedores, utilizando la plataforma y herramientas de Microsoft.

Recursos adicionales

- **Mark Russinovich. Microservicios: una revolución de aplicaciones con tecnología de la nube**
  https://azure.microsoft.com/blog/microservices-an-application-revolution-powered-by-the-cloud/
- **Martin Fowler. Microservicios**
  https://www.martinfowler.com/articles/microservices.html
- **Martin Fowler. Requisitos previos de microservicios**
  https://martinfowler.com/bliki/MicroservicePrerequisites.html
- **Jimmy Nilsson. Computación en la nube por bloques**
  https://www.infoq.com/articles/CCC-Jimmy-Nilsson
- **Cesar de la Torre. Ciclo de vida de aplicaciones de Docker en contenedores con la plataforma y las herramientas de Microsoft** (libro electrónico descargable)
  https://aka.ms/dockerlifecycleebook

Soberanía de datos por microservicio

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Una regla importante para la arquitectura de microservicios es que cada microservicio debe poseer sus datos de dominio y lógica. Al igual que una aplicación completa posee su lógica y sus datos, por lo que cada microservicio tiene su lógica y sus datos en un ciclo de vida autónomo, con implementación independiente por microservicio.

Esto significa que el modelo conceptual del dominio variará entre subsistemas o microservicios. Considere las aplicaciones empresariales, donde las aplicaciones de administración de relaciones con el cliente (CRM), los subsistemas de compra transaccional y los subsistemas de soporte al cliente utilizan atributos y datos únicos de entidades de cliente, y donde cada uno emplea un contexto delimitado (BC) diferente.

Este principio es similar en el diseño controlado por dominio (DDD), donde cada contexto enlazado o subsistema o servicio autónomo debe poseer su modelo de dominio (datos más lógica y comportamiento). Cada contexto enlazado de DDD se correlaciona con un microservicio empresarial (uno o varios servicios). Este punto sobre el patrón de Contexto Delimitado se expande en la sección siguiente.

Por otro lado, el enfoque tradicional (datos monolíticos) que se usa en muchas aplicaciones es tener una base de datos centralizada única o solo algunas bases de datos. Suele ser una base de datos SQL normalizada que se usa para toda la aplicación y todos sus subsistemas internos, como se muestra en la figura 4-7.

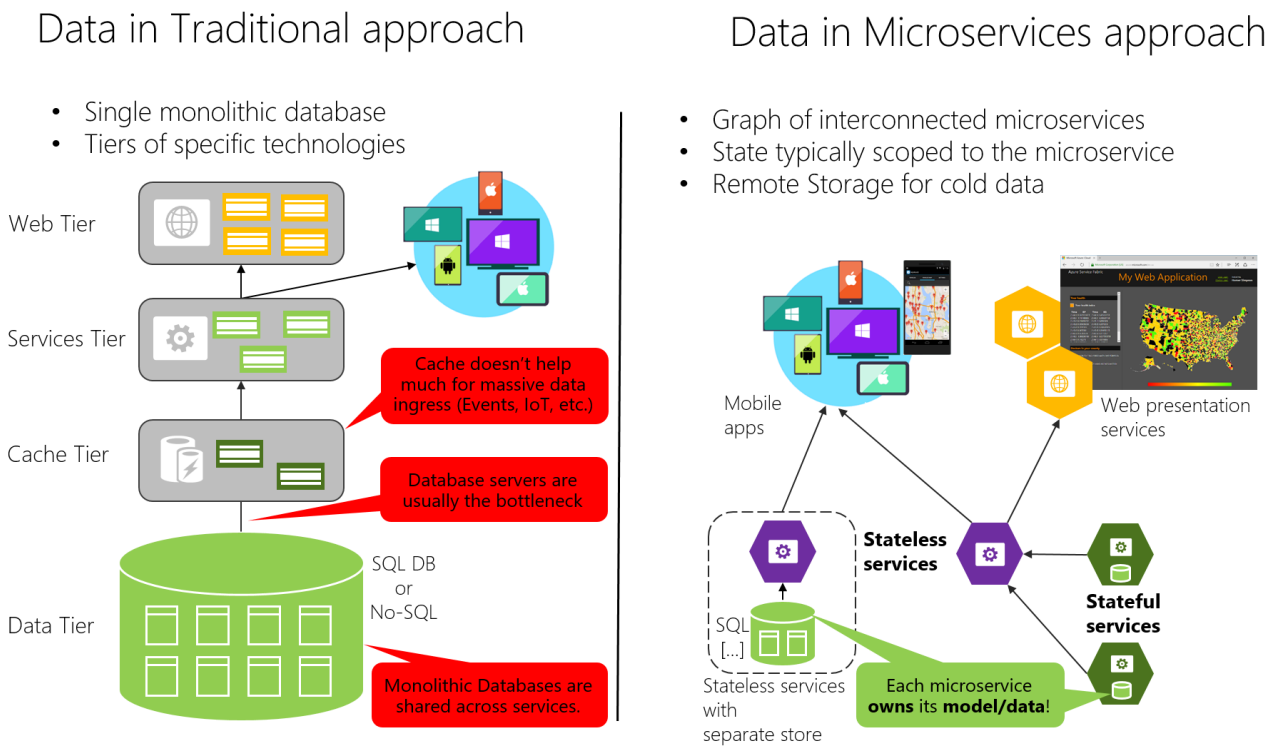


Figura 4-7. Comparación de soberanía de datos: base de datos monolítica frente a microservicios

En el enfoque tradicional, hay una base de datos única compartida en todos los servicios, normalmente en una arquitectura en capas. En el enfoque de microservicios, cada microservicio posee su modelo o datos. El enfoque centralizado de la base de datos se ve inicialmente más sencillo y parece permitir la reutilización de entidades en distintos subsistemas para que todo sea coherente. Pero la realidad es que termina con tablas enormes que sirven a muchos subsistemas diferentes y que incluyen atributos y columnas que no son necesarios en la mayoría de los casos. Es como intentar usar el mismo mapa físico para hacer senderismo en una ruta corta, realizar un viaje en coche de un día y aprender geografía.

Una aplicación monolítica con normalmente una base de datos relacional única tiene dos ventajas importantes: transacciones ACID y el lenguaje SQL, que funcionan en todas las tablas y datos relacionados con la aplicación. Este enfoque proporciona una manera de escribir fácilmente una consulta que combina datos de varias tablas.

Sin embargo, el acceso a datos se vuelve mucho más complicado al pasar a una arquitectura de microservicios. Incluso cuando se usan transacciones ACID dentro de un microservicio o contexto enlazado, es fundamental tener en cuenta que los datos que pertenecen a cada microservicio son privados para ese microservicio y solo se debe tener acceso de forma sincrónica a través de sus puntos de conexión de API (REST, gRPC, SOAP, etc.) o de forma asincrónica a través de mensajería (AMQP o similar).

Encapsular los datos garantiza que los microservicios están acoplados de forma flexible y pueden evolucionar independientemente entre sí. Si varios servicios tuvieran acceso a los

mismos datos, las actualizaciones de esquema requerirían actualizaciones coordinadas para todos los servicios. Esto interrumpiría la autonomía del ciclo de vida del microservicio. Pero las estructuras de datos distribuidas significan que no se puede realizar una sola transacción ACID entre microservicios. Esto a su vez significa que debe usar la consistencia eventual cuando un proceso de negocio abarca varios microservicios. Esto es mucho más difícil de implementar que las combinaciones SQL simples, ya que no se pueden crear restricciones de integridad ni usar transacciones distribuidas entre bases de datos independientes, como explicaremos más adelante. Del mismo modo, muchas otras características de base de datos relacionales no están disponibles en varios microservicios.

A menudo, los microservicios diferentes usan diferentes *tipos* de bases de datos. Las aplicaciones modernas almacenan y procesan diversos tipos de datos y una base de datos relacional no siempre es la mejor opción. En algunos casos de uso, una base de datos NoSQL, como Azure CosmosDB o MongoDB, podría tener un modelo de datos más conveniente y ofrecer un mejor rendimiento y escalabilidad que una base de datos SQL como SQL Server o Azure SQL Database. En otros casos, una base de datos relacional sigue siendo el mejor enfoque. Por lo tanto, las aplicaciones basadas en microservicios suelen utilizar una mezcla de bases de datos SQL y NoSQL, que a veces se denomina enfoque de persistencia políglota.

Una arquitectura con particiones de persistencia políglota para el almacenamiento de datos tiene muchas ventajas. Estos incluyen servicios acoplados de forma flexible y un mejor rendimiento, escalabilidad, costos y capacidad de administración. Sin embargo, puede presentar algunos desafíos de administración de datos distribuidos, como se explica en "Identificación de límites del modelo de dominio" más adelante en este capítulo.

Relación entre microservicios y el patrón de contexto enlazado

El concepto de microservicio se deriva del patrón de Contexto Delimitado (BC) en diseño dirigido por el dominio (DDD). DDD trabaja con modelos grandes al dividirlos en varios contextos enlazados y ser explícito sobre sus límites. Cada BC debe tener su propio modelo y base de datos; del mismo modo, cada microservicio posee sus datos relacionados. Además, cada BC suele tener su propio lenguaje omnipresente para ayudar a la comunicación entre desarrolladores de software y expertos en dominios.

Esos términos (principalmente entidades de dominio) en el lenguaje omnipresente pueden tener nombres diferentes en contextos enlazados diferentes, incluso cuando diferentes entidades de dominio comparten la misma identidad (es decir, el identificador único que se usa para leer la entidad del almacenamiento). Por ejemplo, en un contexto enlazado de perfil de usuario, la entidad de dominio User puede compartir identidad con la entidad de dominio Buyer en el contexto enlazado Ordering.

Por lo tanto, un microservicio es como un contexto enlazado, pero también especifica que es un servicio distribuido. Se crea como un proceso independiente para cada contexto enlazado y debe usar los protocolos distribuidos indicados anteriormente, como HTTP/HTTPS, WebSockets o AMQP. Sin embargo, el patrón de contexto enlazado no especifica si el contexto enlazado es un servicio distribuido o si es simplemente un límite lógico (como un subsistema genérico) dentro de una aplicación de implementación monolítica.

Es importante resaltar que definir un servicio para cada contexto delimitado es un buen punto de partida. Pero no tienes que limitar tu diseño a eso. A veces, debe diseñar un microservicio de contexto enlazado o empresarial compuesto por varios servicios físicos. Pero en última instancia, ambos patrones -Bounded Contexto y microservicio- están estrechamente relacionados.

DDD se beneficia de los microservicios obteniendo límites reales en forma de microservicios distribuidos. Pero el concepto de no compartir el modelo entre microservicios es algo que también se busca en un Contexto Delimitado.

Recursos adicionales

- **Chris Richardson. Patrón: base de datos por servicio**
  https://microservices.io/patterns/data/database-per-service.html
- **Martin Fowler. Contexto Delimitado**
  https://martinfowler.com/bliki/BoundedContext.html
- **Martin Fowler. PolyglotPersistence**
  https://martinfowler.com/bliki/PolyglotPersistence.html
- **Alberto Brandolini. Diseño Estratégico Guiado por Dominios con Mapeo de Contexto**
  https://www.infoq.com/articles/ddd-contextmapping

Siguiente anterior

Arquitectura lógica frente a arquitectura física

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En este punto resulta útil detener y analizar la distinción entre la arquitectura lógica y la arquitectura física, y cómo se aplica al diseño de aplicaciones basadas en microservicios.

Para empezar, la creación de microservicios no requiere el uso de ninguna tecnología específica. Por ejemplo, los contenedores de Docker no son obligatorios para crear una arquitectura basada en microservicios. Esos microservicios también se pueden ejecutar como procesos simples. Los microservicios son una arquitectura lógica.

Además, incluso cuando un microservicio se puede implementar físicamente como un único servicio, proceso o contenedor (por motivos de simplicidad, es el enfoque adoptado en la versión inicial de eShopOnContainers), esta paridad entre el microservicio empresarial y el servicio físico o contenedor no es necesariamente necesariamente en todos los casos cuando se crea una aplicación grande y compleja compuesta por muchas docenas o incluso cientos de servicios.

Aquí es donde hay una diferencia entre la arquitectura lógica de una aplicación y la arquitectura física. La arquitectura lógica y los límites lógicos de un sistema no se corresponden necesariamente de manera directa con la arquitectura física o de implementación. Puede ocurrir, pero a menudo no lo hace.

Aunque es posible que haya identificado determinados microservicios empresariales o contextos enlazados, no significa que la mejor manera de implementarlos siempre es crear un único servicio (como una API web de ASP.NET) o un único contenedor de Docker para cada microservicio empresarial. Tener una regla que indica que cada microservicio empresarial debe implementarse mediante un único servicio o contenedor es demasiado rígido.

Por lo tanto, un microservicio empresarial o contexto enlazado es una arquitectura lógica que podría coincidir (o no) con la arquitectura física. El punto importante es que un microservicio empresarial o contexto enlazado debe ser autónomo al permitir que el código y el estado se controlen de forma independiente, se implementen y escalen.

Como se muestra en la figura 4-8, el microservicio empresarial del catálogo podría estar compuesto de varios servicios o procesos. Pueden ser varios ASP.NET servicios de API web o cualquier otro tipo de servicios mediante HTTP o cualquier otro protocolo. Lo más importante es que los servicios puedan compartir los mismos datos, siempre que estos servicios sean cohesivos con respecto al mismo dominio empresarial.

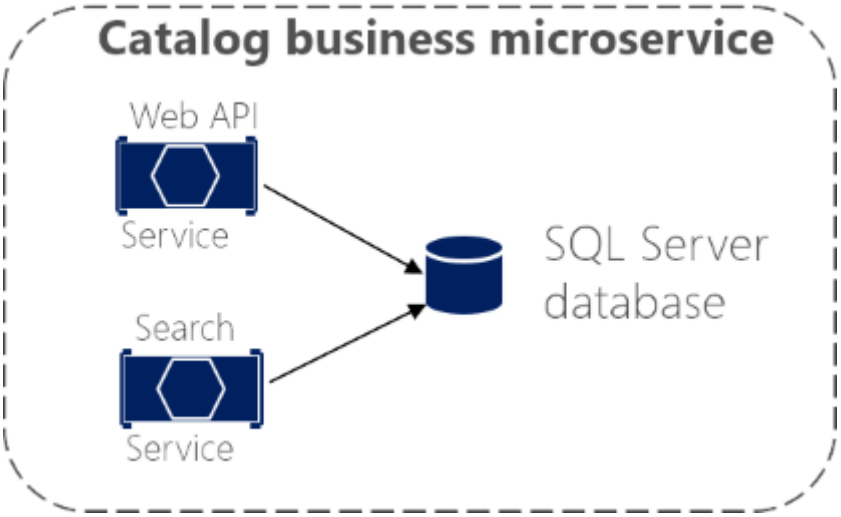


Figura 4-8. Microservicio empresarial con varios servicios físicos

Los servicios del ejemplo comparten el mismo modelo de datos porque el servicio API web tiene como destino los mismos datos que el servicio search. Por lo tanto, en la implementación física del microservicio empresarial, vas a dividir esa funcionalidad para que puedas escalar hacia arriba o hacia abajo cada uno de esos servicios internos según sea necesario. Quizás el servicio API web normalmente necesita más instancias que el servicio de búsqueda, o viceversa.

En resumen, la arquitectura lógica de los microservicios no siempre tiene que coincidir con la arquitectura de implementación física. En esta guía, siempre que mencionamos un microservicio, significamos un microservicio empresarial o lógico que podría asignarse a uno o varios servicios (físicos). En la mayoría de los casos, este será un único servicio, pero puede ser más.

Desafíos y soluciones para la administración de datos distribuidos

19/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Desafío 1: Cómo definir los límites de cada microservicio

Definir límites de microservicios probablemente es el primer desafío que cualquier persona encuentra. Cada microservicio debe ser una parte de la aplicación y cada microservicio debe ser autónomo con todas las ventajas y desafíos que transmite. ¿Pero cómo se identifican esos límites?

En primer lugar, debe centrarse en los modelos de dominio lógicos de la aplicación y en los datos relacionados. Intente identificar islas desacopladas de datos y contextos diferentes dentro de la misma aplicación. Cada contexto podría tener un lenguaje de negocio diferente (términos empresariales diferentes). Los contextos deben definirse y administrarse de forma independiente. Los términos y entidades que se usan en esos contextos diferentes pueden parecer similares, pero es posible que descubra que, en un contexto determinado, se usa un concepto empresarial con uno para un propósito diferente en otro contexto e incluso podría tener un nombre diferente. Por ejemplo, un usuario puede denominarse usuario en el contexto

de identidad o pertenencia, como cliente en un contexto de CRM, como comprador en un contexto de ordenación, etc.

La manera de identificar los límites entre varios contextos de aplicación con un dominio diferente para cada contexto es exactamente cómo puede identificar los límites de cada microservicio empresarial y su modelo de dominio y datos relacionados. Siempre intenta minimizar el acoplamiento entre esos microservicios. En esta guía se detalla más detalladamente este diseño de modelo de dominio y identificación en la sección Identificación de los límites del modelo de dominio para cada microservicio más adelante.

Desafío 2: Creación de consultas que recuperan datos de varios microservicios

Un segundo desafío es cómo implementar consultas que recuperan datos de varios microservicios, al tiempo que evitan la comunicación chatty con los microservicios desde aplicaciones cliente remotas. Un ejemplo podría ser una sola pantalla de una aplicación móvil que necesita mostrar la información del usuario que pertenece a los microservicios de cesta, catálogo e identidad de usuario. Otro ejemplo sería un informe complejo que implicaba muchas tablas ubicadas en varios microservicios. La solución adecuada depende de la complejidad de las consultas. Pero en cualquier caso, necesitará una manera de agregar información si desea mejorar la eficacia en las comunicaciones del sistema. Las soluciones más populares son las siguientes.

Puerta de enlace de API. Para la agregación de datos sencilla de varios microservicios que poseen bases de datos diferentes, el enfoque recomendado es un microservicio de agregación al que se hace referencia como puerta de enlace de API. Sin embargo, debe tener cuidado de implementar este patrón, ya que puede ser un punto de ahogo en el sistema, y puede infringir el principio de autonomía de microservicios. Para mitigar esta posibilidad, puede tener varias puertas de enlace de API específicas definidas, cada una de ellas centrada en un "segmento" vertical o un área de negocio del sistema. El patrón de puerta de enlace de API se explica con más detalle en la sección de puerta de enlace de API más adelante.

Federación de GraphQL Una opción que se debe tener en cuenta si los microservicios ya usan GraphQL es La federación de GraphQL. La federación permite definir "subgráficos" de otros servicios y redactarlos en un "supergráfico" agregado que actúa como un esquema independiente.

CQRS con tablas de consulta o lectura. Otra solución para agregar datos de varios microservicios es el patrón Materialized View. En este enfoque, se genera de antemano (preparar datos desnormalizados antes de que se produzcan las consultas reales), una tabla de

solo lectura con los datos que pertenecen a varios microservicios. La tabla tiene un formato adecuado para las necesidades de la aplicación cliente.

Considere algo parecido a la pantalla de una aplicación móvil. Si tiene una base de datos única, puede extraer los datos de esa pantalla mediante una consulta SQL que realice una combinación compleja que implique varias tablas. Sin embargo, cuando tiene varias bases de datos y cada base de datos es propiedad de un microservicio diferente, no puede consultar esas bases de datos y crear una combinación SQL. La consulta compleja se convierte en un desafío. Puede abordar el requisito mediante un enfoque de CQRS: se crea una tabla desnormalizada en una base de datos diferente que se usa solo para las consultas. La tabla se puede diseñar específicamente para los datos que necesita para la consulta compleja, con una relación uno a uno entre los campos necesarios para la pantalla de la aplicación y las columnas de la tabla de consulta. También podría servir para la elaboración de informes.

Este enfoque no solo resuelve el problema original (cómo consultar y combinar entre microservicios), sino que también mejora considerablemente el rendimiento en comparación con una combinación compleja, ya que ya tiene los datos que la aplicación necesita en la tabla de consulta. Por supuesto, la utilización de CQRS (Segregación de responsabilidades de comandos y consultas) con tablas de consulta o lectura implica un mayor trabajo de desarrollo y debe adoptarse coherencia final. Sin embargo, los requisitos de rendimiento y alta escalabilidad en escenarios de colaboración (o escenarios competitivos , según el punto de vista) son donde debe aplicar CQRS con varias bases de datos.

"Datos en frío" en bases de datos centrales. En el caso de informes complejos y consultas que podrían no requerir datos en tiempo real, un enfoque común consiste en exportar los "datos activos" (datos transaccionales de los microservicios) como "datos inactivos" en bases de datos grandes que solo se usan para los informes. Ese sistema central de bases de datos puede ser un sistema basado en macrodatos, como Hadoop; un almacenamiento de datos como uno basado en Azure SQL Data Warehouse; o incluso una base de datos SQL única que se usa solo para los informes (si el tamaño no es un problema).

Tenga en cuenta que esta base de datos centralizada solo se usaría para consultas e informes que no necesitan datos en tiempo real. Las actualizaciones y las transacciones originales, como origen confiable, deben estar en los datos de microservicios. La forma en que sincronizaría los datos sería mediante la comunicación controlada por eventos (que se trata en las secciones siguientes) o mediante otras herramientas de importación y exportación de la infraestructura de base de datos. Si usa la comunicación controlada por eventos, ese proceso de integración sería similar a la forma en que propaga los datos tal como se describió anteriormente para las tablas de consulta CQRS.

Sin embargo, si el diseño de la aplicación implica agregar constantemente información de varios microservicios para consultas complejas, podría ser un síntoma de un diseño incorrecto -

a microservicio debe estar lo más aislado posible de otros microservicios. (Esto excluye informes o análisis que siempre deben usar bases de datos centrales de datos en frío). Tener este problema a menudo podría ser una razón para combinar microservicios. Debe equilibrar la autonomía de la evolución y la implementación de cada microservicio con dependencias fuertes, cohesión y agregación de datos.

Desafío 3: Cómo lograr la coherencia en varios microservicios

Como se indicó anteriormente, los datos propiedad de cada microservicio son privados para ese microservicio y solo se puede acceder a ellos mediante su API de microservicios. Por lo tanto, un desafío presentado es cómo implementar procesos empresariales de un extremo a otro, a la vez que se mantiene la coherencia entre varios microservicios.

Para analizar este problema, echemos un vistazo a un ejemplo de la aplicación de referencia eShopOnContainers. El microservicio Catalog mantiene información sobre todos los productos, incluido el precio del producto. El microservicio Basket administra datos temporales sobre los productos que los usuarios agregan a sus cestas de compras, lo que incluye el precio de los artículos en el momento en que se agregaron a la cesta. Cuando el precio de un producto se actualiza en el catálogo, ese precio también debe actualizarse en las cestas activas que contienen ese mismo producto, además del sistema probablemente debería advertir al usuario que dice que el precio de un artículo determinado ha cambiado desde que lo agregó a su cesta.

En una versión hipotética monolítica de esta aplicación, cuando el precio cambia en la tabla de productos, el subsistema de catálogo podría simplemente usar una transacción ACID para actualizar el precio actual en la tabla Basket.

Sin embargo, en una aplicación basada en microservicios, las tablas Product y Basket son propiedad de sus respectivos microservicios. Ningún microservicio nunca debe incluir tablas o almacenamiento propiedad de otro microservicio en sus propias transacciones, ni siquiera en consultas directas, como se muestra en la figura 4-9.

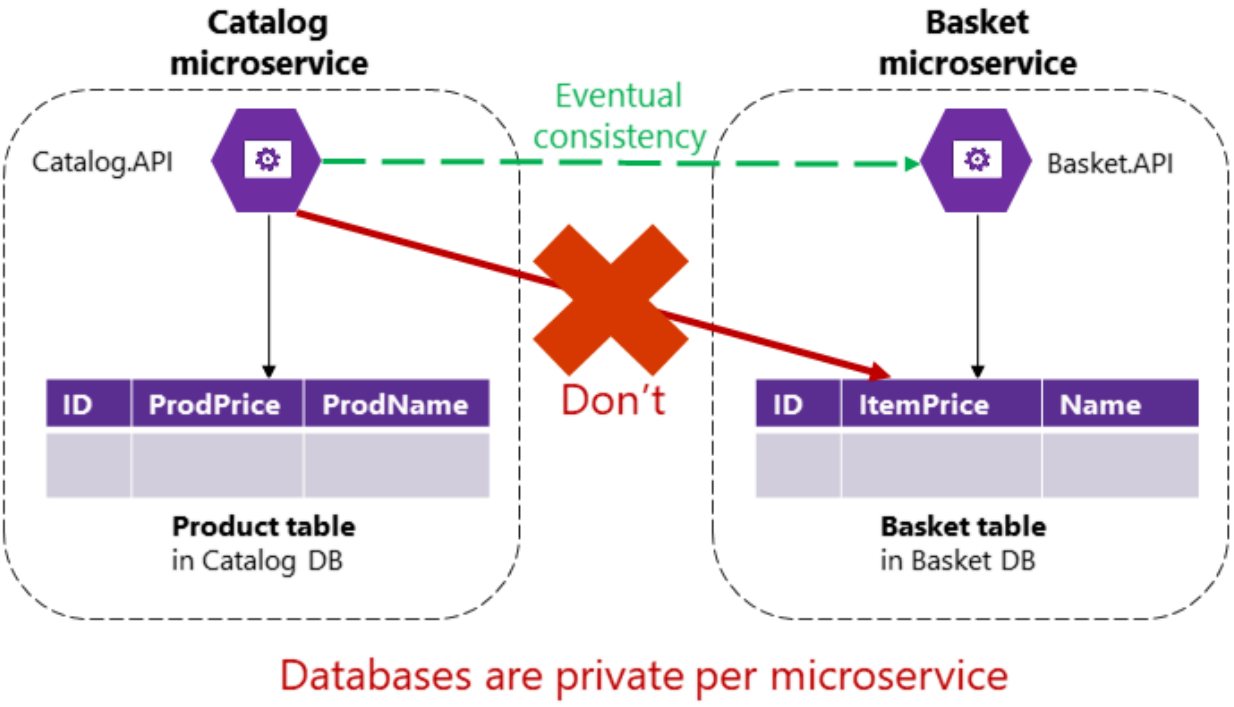


Figura 4-9. Un microservicio no puede acceder directamente a una tabla en otro microservicio

El microservicio Catalog no debe actualizar directamente la tabla Basket, ya que la tabla Basket es propiedad del microservicio Basket. Para realizar una actualización del microservicio Basket, el microservicio Catálogo debe usar la coherencia eventual, probablemente basada en la comunicación asincrónica, como mediante eventos de integración (comunicación basada en mensajes y eventos). Así es como la aplicación de referencia eShopOnContainers realiza este tipo de coherencia entre microservicios.

Como indica el teorema CAP, debe elegir entre disponibilidad y coherencia ACID. La mayoría de los escenarios basados en microservicios exigen disponibilidad y alta escalabilidad, en lugar de una coherencia fuerte. Las aplicaciones de misión crítica deben permanecer en funcionamiento, y los desarrolladores pueden sortear la consistencia fuerte utilizando técnicas para trabajar con consistencia débil o eventual. Este es el enfoque adoptado por la mayoría de las arquitecturas basadas en microservicios.

Además, las transacciones de confirmación de dos fases o de estilo ACID no solo se aplican a los principios de microservicios; La mayoría de las bases de datos NoSQL (como Azure Cosmos DB, MongoDB, etc.) no admiten transacciones de confirmación en dos fases, típicas en escenarios de bases de datos distribuidas. Sin embargo, mantener la coherencia de los datos entre servicios y bases de datos es esencial. Este desafío también está relacionado con la cuestión de cómo propagar los cambios en varios microservicios cuando determinados datos deben ser redundantes, por ejemplo, cuando necesita tener el nombre o la descripción del producto en el microservicio catalog y el microservicio Basket.

Una buena solución para este problema es usar la coherencia final entre los microservicios articulados a través de la comunicación controlada por eventos y un sistema de publicación y suscripción. Estos temas se tratan en la sección Comunicación asincrónica controlada por eventos más adelante en esta guía.

Desafío n.º 4: Cómo diseñar la comunicación entre límites de microservicios

La comunicación entre límites de microservicios es un desafío real. En este contexto, la comunicación no hace referencia al protocolo que debe usar (HTTP y REST, AMQP, mensajería, etc.). En su lugar, aborda qué estilo de comunicación debe usar y, especialmente, cómo de acoplados deben estar los microservicios. Según el nivel de acoplamiento, cuando se produzca un error, el impacto de ese error en el sistema variará significativamente.

En un sistema distribuido como una aplicación basada en microservicios, con tantos artefactos que se mueven y con servicios distribuidos entre muchos servidores o hosts, los componentes finalmente producirán un error. Se producirán errores parciales e incluso interrupciones mayores, por lo que debe diseñar los microservicios y la comunicación entre ellos teniendo en cuenta los riesgos comunes en este tipo de sistema distribuido.

Un enfoque popular es implementar microservicios basados en HTTP (REST), debido a su simplicidad. Un enfoque basado en HTTP es perfectamente aceptable; el problema aquí está relacionado con cómo se usa. Si usa solicitudes y respuestas HTTP solo para interactuar con los microservicios desde aplicaciones cliente o desde puertas de enlace de API, es correcto. Pero si crea cadenas largas de llamadas HTTP sincrónicas a través de microservicios, comunicándose a través de sus límites como si los microservicios fueran objetos en una aplicación monolítica, la aplicación finalmente se encontrará con problemas.

Por ejemplo, imagine que la aplicación cliente realiza una llamada API HTTP a un microservicio individual, como el microservicio Ordering. Si el microservicio Ordering a su vez llama a microservicios adicionales mediante HTTP dentro del mismo ciclo de solicitud/respuesta, va a crear una cadena de llamadas HTTP. Podría sonar razonable inicialmente. Sin embargo, hay puntos importantes que se deben tener en cuenta al seguir este camino.

- Bloqueo y bajo rendimiento. Debido a la naturaleza sincrónica de HTTP, la solicitud original no obtiene una respuesta hasta que finalizan todas las llamadas HTTP internas. Imagine si el número de estas llamadas aumenta significativamente y al mismo tiempo se bloquea una de las llamadas HTTP intermedias a un microservicio. El resultado es que el rendimiento se ve afectado y la escalabilidad general se verá afectada exponencialmente a medida que aumentan las solicitudes HTTP adicionales.

- Acoplamiento de microservicios con HTTP. Los microservicios empresariales no deben acoplarse con otros microservicios empresariales. Idealmente, no deben "saber" la existencia de otros microservicios. Si la aplicación se basa en microservicios de acoplamiento como en el ejemplo, lograr la autonomía por microservicio será casi imposible.

- Error en un microservicio. Si implementó una cadena de microservicios vinculados por llamadas HTTP, cuando se produce un error en cualquiera de los microservicios (y finalmente se producirá un error) se producirá un error en toda la cadena de microservicios. Un sistema basado en microservicios debe diseñarse para seguir funcionando así como sea posible durante los errores parciales. Incluso si implementa lógica de cliente que usa reintentos con mecanismos de retroceso exponencial o cortacircuitos, cuanto más complejas sean las cadenas de llamadas HTTP, más compleja es implementar una estrategia de gestión de errores basada en HTTP.

De hecho, si los microservicios internos se comunican mediante la creación de cadenas de solicitudes HTTP como se describe, se podría afirmar que tiene una aplicación monolítica, pero una basada en HTTP entre procesos en lugar de mecanismos de comunicación intraproceso.

Por lo tanto, para aplicar la autonomía de los microservicios y tener una mejor resistencia, debe minimizar el uso de cadenas de comunicación de solicitud/respuesta entre microservicios. Se recomienda usar solo la interacción asincrónica para la comunicación entre microservicios, ya sea mediante la comunicación asincrónica basada en mensajes y eventos, o mediante sondeo HTTP (asincrónico) independientemente del ciclo de solicitud/respuesta HTTP original.

El uso de la comunicación asincrónica se explica con detalles adicionales más adelante en esta guía en las secciones Integración asincrónica de microservicios exige la autonomía del microservicio y la comunicación asincrónica basada en mensajes.

Recursos adicionales

- **Teorema CAP**
  https://en.wikipedia.org/wiki/CAP_theorem

- **Coherencia final**
  https://en.wikipedia.org/wiki/Eventual_consistency

- **Información básica sobre la consistencia de datos**
  https://learn.microsoft.com/previous-versions/msp-n-p/dn589800(v=pandp.10)

- **Martin Fowler. CQRS (Command and Query Responsibility Segregation) [CQRS (Segregación de responsabilidades de consultas y comandos)]**
  https://martinfowler.com/bliki/CQRS.html

- **Vista materializada**
  https://learn.microsoft.com/azure/architecture/patterns/materialized-view
- **Charles Row. ACID frente a BASE: el cambio de pH del procesamiento de transacciones de base de datos**
  https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/
- **Transacción Compensatoria**
  https://learn.microsoft.com/azure/architecture/patterns/compensating-transaction
- **Udi Dahan. Composición orientada a servicios**
  https://udidahan.com/2014/07/30/service-oriented-composition-with-video/

Siguiente anterior

Identificación de los límites del modelo de dominio para cada microservicio

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El objetivo al identificar los límites y el tamaño del modelo para cada microservicio no es alcanzar la separación más granular posible, aunque deberían tender a los microservicios pequeños si es posible. sino que debería ser llegar a la separación más significativa basada en el conocimiento del dominio. El énfasis no está en el tamaño, sino en las funcionalidades empresariales. Además, si hay una cohesión clara necesaria para un área determinada de la aplicación en función de un gran número de dependencias, que también indica la necesidad de un único microservicio. La cohesión es una manera de identificar cómo separar o agrupar microservicios. Finalmente, mientras adquiere más conocimiento sobre el dominio, debe adaptar el tamaño del microservicio de forma iterativa. La búsqueda del tamaño correcto no es un proceso único.

Sam Newman, un reconocido promotor de microservicios y autor del libro Building Microservices, resalta que debes diseñar tus microservicios en función del patrón de Contexto Limitado (BC) (parte del diseño dirigido por el dominio), tal como se explicó previamente. A veces, un BC podría estar compuesto de varios servicios físicos, pero no viceversa.

Un modelo de dominio con entidades de dominio específicas se aplica dentro de un microservicio o BC concreto. Un BC delimita la aplicabilidad de un modelo de dominio y

proporciona a los miembros del equipo de desarrolladores una comprensión clara y compartida de lo que debe ser cohesivo y lo que se puede desarrollar de forma independiente. Estos son los mismos objetivos para los microservicios.

Otra herramienta que informa a su elección de diseño es la ley de Conway, que indica que una aplicación reflejará los límites sociales de la organización que lo produjo. Pero a veces ocurre lo contrario: la organización de la empresa -the está formada por el software. Es posible que tenga que replantear la ley de Conway y establecer los límites de modo que reflejen cómo desea organizar la empresa, enfocándose en la consultoría de procesos empresariales.

Para identificar contextos delimitados, puede usar un patrón DDD denominado patrón de mapeo de contexto. Con la asignación de contexto, se identifican los distintos contextos de la aplicación y sus límites. Es habitual tener un contexto y un límite diferentes para cada subsistema pequeño, por ejemplo. El mapa de contexto es una manera de definir y hacer explícitos esos límites entre dominios. Un BC es autónomo e incluye los detalles de un único dominio -details como las entidades de dominio, y define los contratos de integración con otros BCs. Esto es similar a la definición de un microservicio: es autónomo, implementa cierta funcionalidad de dominio y debe proporcionar interfaces. Por este motivo, la asignación de contexto y el patrón de contexto enlazado son buenos enfoques para identificar los límites del modelo de dominio de los microservicios.

Al diseñar una aplicación grande, verá cómo se puede fragmentar su modelo de dominio: un experto en dominios del dominio del catálogo asignará nombres a entidades de forma diferente en los dominios de catálogo e inventario que un experto en dominios de dominio de envío, por ejemplo. O bien, la entidad de dominio de usuario puede ser diferente en el tamaño y el número de atributos al tratar con un experto de CRM que quiere almacenar cada detalle sobre el cliente que para un experto en dominio de pedidos que solo necesita datos parciales sobre el cliente. Es muy difícil desambiguar todos los términos de dominio en todos los dominios relacionados con una aplicación grande. Pero lo más importante es que no debe intentar unificar los términos. En su lugar, acepte las diferencias y la riqueza proporcionadas por cada dominio. Si intenta crear una base de datos unificada para toda la aplicación, los intentos de unificar el vocabulario serán incómodos y no sonarán adecuados para ninguno de los diversos expertos en los diferentes dominios. Por lo tanto, los BCs (implementados como microservicios) le ayudarán a aclarar dónde puede usar ciertos términos de dominio y dónde tendrá que dividir el sistema y crear BCs adicionales con diferentes dominios.

Sabrá que tiene los límites y tamaños adecuados de cada modelo de dominio y BC si tiene pocas relaciones sólidas entre los modelos de dominio y normalmente no necesita combinar información de varios modelos de dominio al realizar operaciones de aplicación típicas.

Quizás la mejor respuesta a la pregunta sobre el tamaño de un modelo de dominio para cada microservicio es la siguiente: debe tener un BC autónomo, lo más aislado posible, que le

permite trabajar sin tener que cambiar constantemente a otros contextos (otros modelos de microservicios). En la figura 4-10, puede ver cómo varios microservicios (varios BCs) tienen su propio modelo y cómo se pueden definir sus entidades, en función de los requisitos específicos para cada uno de los dominios identificados de la aplicación.

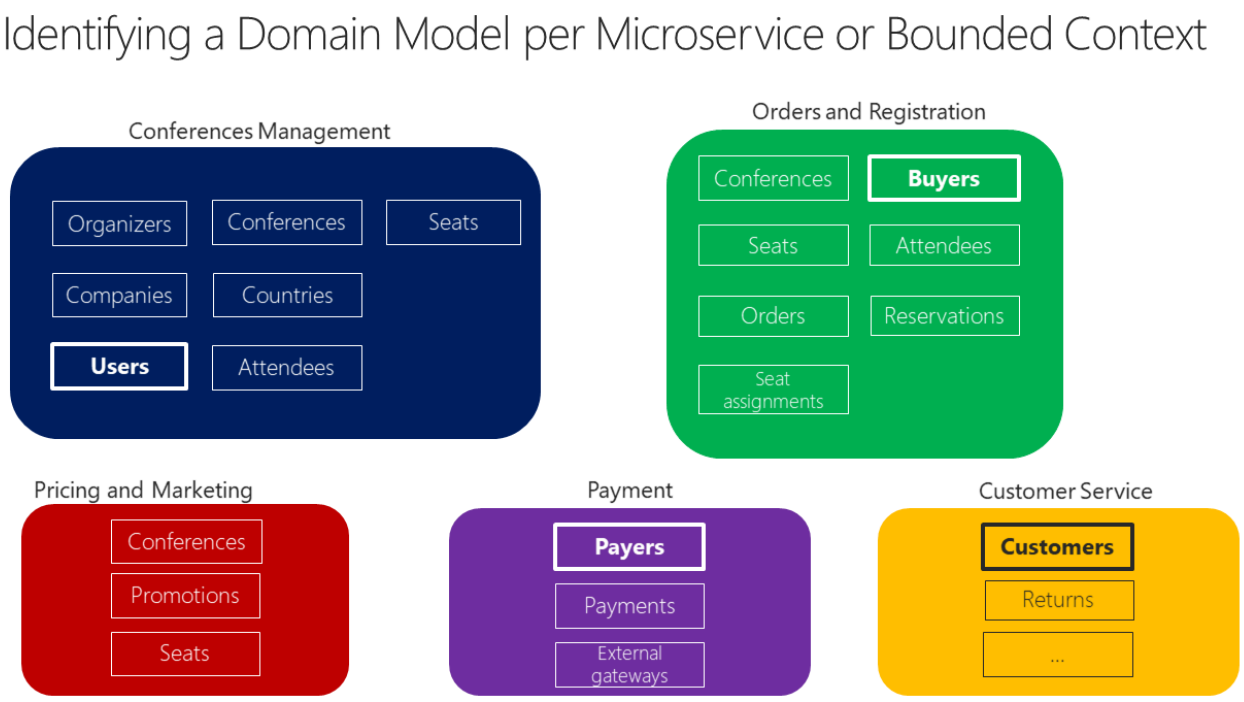


Figura 4-10. Identificación de entidades y límites del modelo de microservicios

En la figura 4-10 se muestra un escenario de ejemplo relacionado con un sistema de administración de conferencias en línea. La misma entidad aparece como "Usuarios", "Compradores", "Pagadores" y "Clientes" en función del contexto delimitado. Ha identificado varios BC que se podrían implementar como microservicios, de acuerdo con los dominios que los expertos en dominios definieron para su caso. Como puede ver, hay entidades que están presentes en un solo modelo de microservicios, como Pagos en el microservicio de pago. Serán fáciles de implementar.

Sin embargo, también puede tener entidades que tengan una forma diferente, pero que compartan la misma identidad en varios modelos de dominio de varios microservicios. Por ejemplo, la entidad User se identifica en el microservicio Administración de conferencias. Ese mismo usuario, con la misma identidad, es el denominado Buyers en el microservicio Ordering, o el llamado Payer en el microservicio Payment, e incluso el denominado Customer en el microservicio Customer Service. Esto se debe a que, dependiendo del lenguaje omnipresente que usa cada experto en dominio, un usuario podría tener una perspectiva diferente incluso con atributos diferentes. La entidad de usuario en el modelo de microservicio llamada Gestión de Conferencias puede tener la mayoría de sus atributos de datos personales. Sin embargo, ese mismo usuario en forma de Payer en el microservicio Pago o en forma de Cliente en el microservicio Servicio al cliente podría no necesitar la misma lista de atributos.

En la figura 4-11 se muestra un enfoque similar.

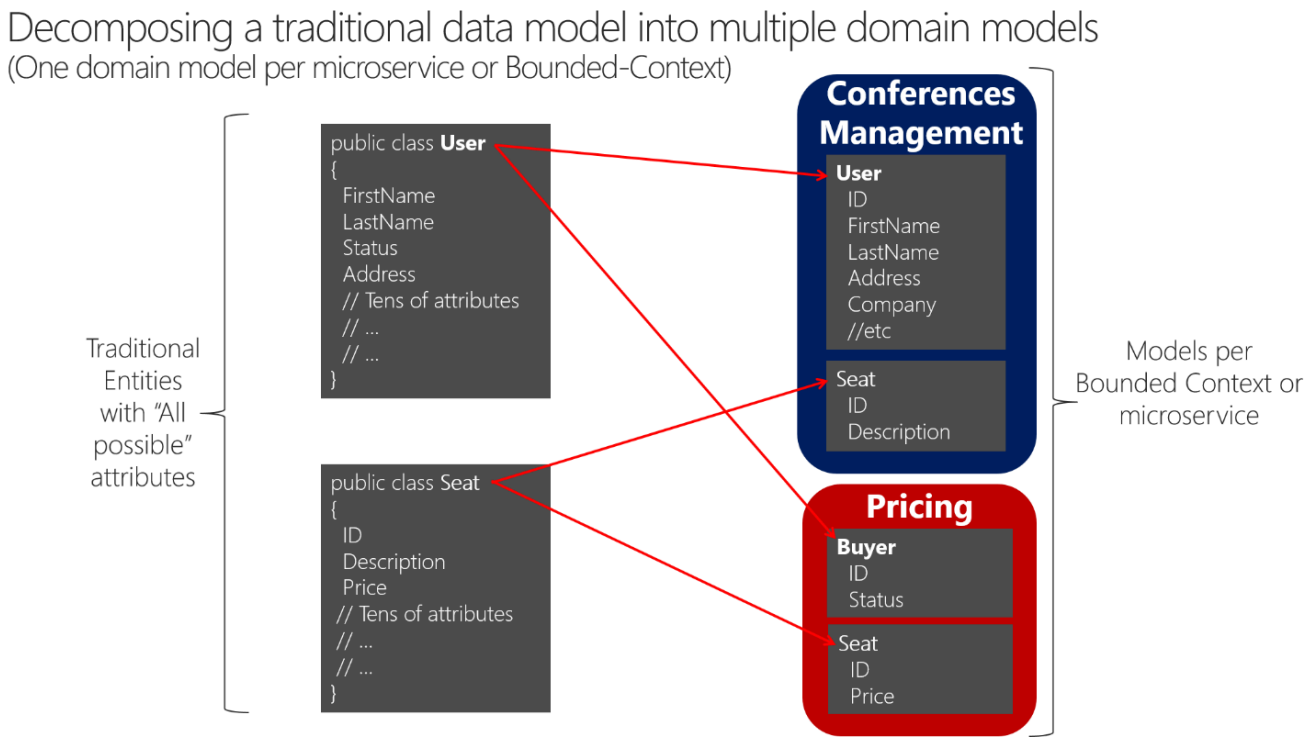


Figura 4-11. Descomposición de modelos de datos tradicionales en varios modelos de dominio

Al descomponer un modelo de datos tradicional entre contextos enlazados, puede tener entidades diferentes que compartan la misma identidad (un comprador también es un usuario) con atributos diferentes en cada contexto enlazado. Puede ver cómo el usuario está presente en el modelo de microservicio de administración de conferencias como la entidad User y también está presente en la forma de la entidad Buyer en el microservicio de precios, con atributos o detalles alternativos sobre el usuario cuando es realmente un comprador. Es posible que cada microservicio o Contexto Limitado (BC) no necesite todos los datos relacionados con una entidad de Usuario, simplemente parte de ella, dependiendo del problema que se va a resolver o del contexto. Por ejemplo, en el modelo de microservicios de precios, no necesita la dirección ni el nombre del usuario, solo el ID (como identificación) y el estado, lo cual tendrá un impacto en los descuentos al fijar los precios de los asientos por comprador.

La entidad Seat tiene el mismo nombre pero atributos diferentes en cada modelo de dominio. pero Seat comparte la identidad según el mismo identificador, como sucede con User y Buyer.

Básicamente, existe un concepto compartido de un usuario que existe en varios servicios (dominios), que comparten la identidad de ese usuario. Pero en cada modelo de dominio puede haber detalles adicionales o diferentes sobre la entidad de usuario. Por lo tanto, debe haber una manera de asignar una entidad de usuario de un dominio (microservicio) a otro.

Hay varias ventajas para no compartir la misma entidad de usuario con el mismo número de atributos entre dominios. Una ventaja es reducir la duplicación, de modo que los modelos de

microservicios no tengan datos que no necesiten. Otra ventaja es tener un microservicio principal que posee un determinado tipo de datos por entidad para que las actualizaciones y las consultas de ese tipo de datos solo estén controladas por ese microservicio.

Siguiente anterior

El patrón de puerta de enlace de API frente a la comunicación directa de cliente a microservicio

19/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En una arquitectura de microservicios, cada microservicio expone un conjunto de puntos de conexión específicos (normalmente). Este hecho puede afectar a la comunicación de cliente a microservicio, como se explica en esta sección.

Comunicación directa de cliente a microservicio

Un posible enfoque es usar una arquitectura de comunicación de cliente a microservicio directa. En este enfoque, una aplicación cliente puede realizar solicitudes directamente a algunos de los microservicios, como se muestra en la figura 4-12.

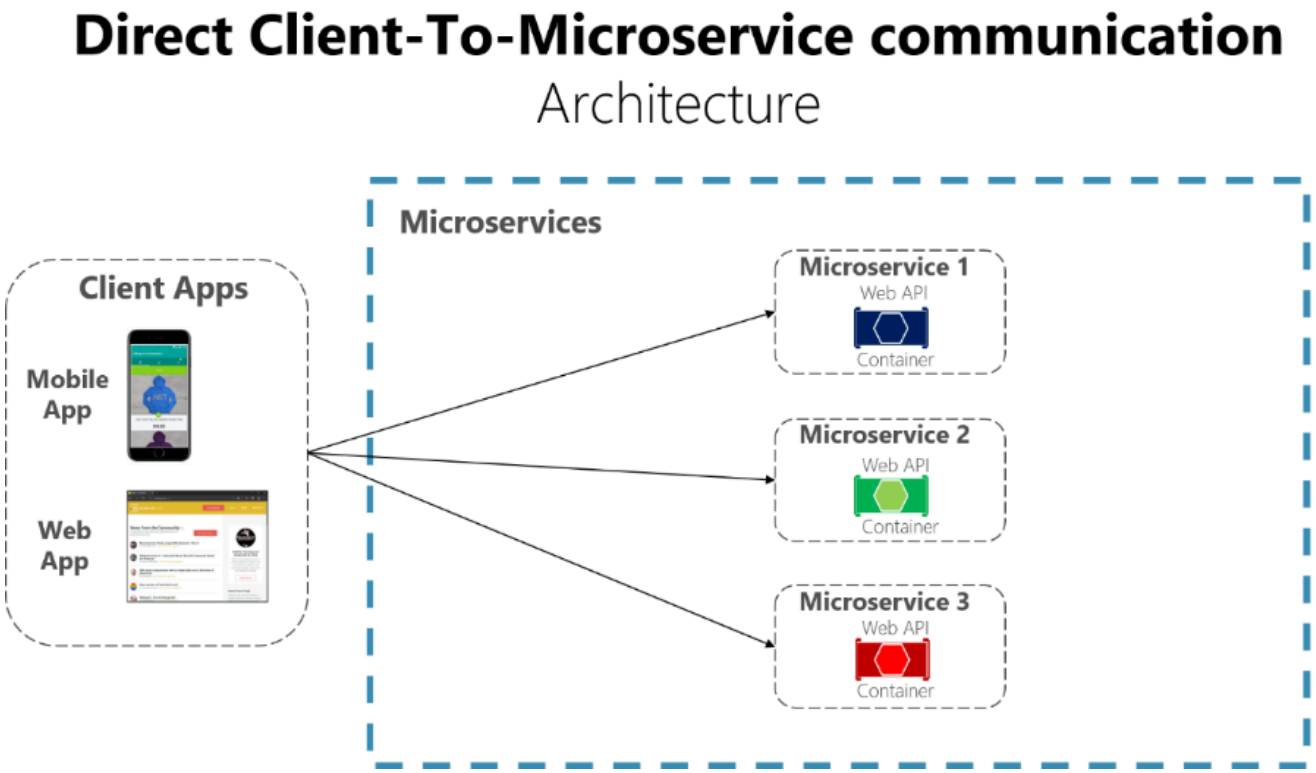


Figura 4-12. Uso de una arquitectura de comunicación de cliente a microservicio directa

En este enfoque, cada microservicio tiene un punto de conexión público, a veces con un puerto TCP diferente para cada microservicio. Un ejemplo de una dirección URL para un servicio determinado podría ser la siguiente dirección URL en Azure:

`http://eshoponcontainers.westus.cloudapp.azure.com:88/`

En un entorno de producción basado en un clúster, esa dirección URL se asignaría al equilibrador de carga usado en el clúster, que a su vez distribuye las solicitudes entre los microservicios. En entornos de producción, podría tener un controlador de entrega de aplicaciones (ADC) como Azure Application Gateway entre los microservicios e Internet. Esta capa actúa como un nivel transparente que no solo realiza el equilibrio de carga, sino que protege los servicios ofreciendo terminación SSL. Este enfoque mejora la carga de los servidores al transferir la terminación SSL intensiva de CPU y otras funciones de enrutamiento al Azure Application Gateway. En cualquier caso, un equilibrador de carga y ADC son transparentes desde un punto de vista de arquitectura de aplicación lógica.

Una arquitectura de comunicación de cliente a microservicio directa podría ser suficiente para una pequeña aplicación basada en microservicios, especialmente si la aplicación cliente es una aplicación web del lado servidor como una aplicación ASP.NET MVC. Sin embargo, al construir aplicaciones grandes y complejas basadas en microservicios (por ejemplo, al gestionar docenas de tipos de microservicios), y especialmente cuando las aplicaciones cliente son aplicaciones móviles remotas o aplicaciones web SPA, este enfoque enfrenta ciertos problemas.

Tenga en cuenta las siguientes preguntas al desarrollar una aplicación grande basada en microservicios:

- *¿Cómo pueden las aplicaciones cliente minimizar el número de solicitudes al back-end y reducir la comunicación chatty a varios microservicios?*

La interacción con varios microservicios para crear una sola pantalla de interfaz de usuario aumenta el número de recorridos de ida y vuelta a través de Internet. Este enfoque aumenta la latencia y la complejidad en la interfaz de usuario. Idealmente, las respuestas deben agregarse eficazmente en el lado servidor. Este enfoque reduce la latencia, ya que varios fragmentos de datos vuelven en paralelo y algunas interfaces de usuario pueden mostrar los datos tan pronto como esté listo.

- *¿Cómo puede controlar los problemas transversales, como la autorización, las transformaciones de datos y el envío dinámico de solicitudes?*

La implementación de problemas de seguridad y transversales, como la seguridad y la autorización en cada microservicio, pueden requerir un esfuerzo de desarrollo significativo. Un posible enfoque consiste en tener esos servicios dentro del host de Docker o en el clúster interno para restringir el acceso directo a ellos desde el exterior e implementar esos problemas transversales en un lugar centralizado, como una puerta de enlace de API.

- *¿Cómo se pueden comunicar las aplicaciones cliente con servicios que usan protocolos que no son compatibles con Internet?*

Los protocolos usados en el lado servidor (como AMQP o protocolos binarios) no se admiten en las aplicaciones cliente. Por lo tanto, las solicitudes deben realizarse a través de protocolos como HTTP/HTTPS y traducirse a los demás protocolos después. Un enfoque *man-in-the-middle* puede ayudar en esta situación.

- *¿Cómo puede dar forma a una fachada especialmente hecha para aplicaciones móviles?*

Es posible que la API de varios microservicios no esté bien diseñada para las necesidades de diferentes aplicaciones cliente. Por ejemplo, las necesidades de una aplicación móvil pueden ser diferentes de las necesidades de una aplicación web. En el caso de las aplicaciones móviles, es posible que tenga que optimizar aún más para que las respuestas de datos puedan ser más eficaces. Puede realizar esta funcionalidad agregando datos de varios microservicios y devolviendo un único conjunto de datos y, a veces, eliminando los datos de la respuesta que no necesita la aplicación móvil. Y, por supuesto, podría comprimir esos datos. De nuevo, una fachada o una API entre la aplicación móvil y los microservicios pueden ser convenientes para este escenario.

Por qué considerar las puertas de enlace de API en lugar de la comunicación directa de cliente a microservicio

En una arquitectura de microservicios, las aplicaciones cliente suelen necesitar consumir funcionalidad de más de un microservicio. Si ese consumo se realiza directamente, el cliente debe controlar varias llamadas a los puntos de conexión de microservicios. ¿Qué ocurre cuando la aplicación evoluciona y se introducen nuevos microservicios o se actualizan los microservicios existentes? Si la aplicación tiene muchos microservicios, controlar tantos puntos de conexión de las aplicaciones cliente pueden ser una pesadilla. Dado que la aplicación cliente se acoplaría a esos puntos de conexión internos, la evolución de los microservicios en el futuro puede provocar un gran impacto para las aplicaciones cliente.

Por lo tanto, tener un nivel intermedio o nivel de direccionamiento indirecto (puerta de enlace) puede ser cómodo para las aplicaciones basadas en microservicios. Si no tiene puertas de enlace de API, las aplicaciones cliente deben enviar solicitudes directamente a los microservicios y que generen problemas, como los siguientes:

- **Acoplamiento**: sin el patrón de puerta de enlace de API, las aplicaciones cliente se acoplan a los microservicios internos. Las aplicaciones cliente deben saber cómo se descompone varias áreas de la aplicación en microservicios. Al evolucionar y refactorizar los microservicios internos, esas acciones afectan al mantenimiento porque provocan cambios importantes en las aplicaciones cliente debido a la referencia directa a los microservicios internos de las aplicaciones cliente. Las aplicaciones cliente deben actualizarse con frecuencia, lo que dificulta la evolución de la solución.

- **Demasiados recorridos de ida y vuelta**: una sola página o pantalla en la aplicación cliente puede requerir varias llamadas a varios servicios. Este enfoque puede dar lugar a varios recorridos de ida y vuelta de red entre el cliente y el servidor, lo que agrega una latencia significativa. La agregación controlada en un nivel intermedio podría mejorar el rendimiento y la experiencia del usuario para la aplicación cliente.

- **Problemas de seguridad**: sin una puerta de enlace, todos los microservicios deben exponerse al "mundo externo", haciendo que la superficie expuesta a ataques sea mayor que si oculta microservicios internos que las aplicaciones cliente no usan directamente. Cuanto menor sea la superficie expuesta a ataques, más segura puede ser la aplicación.

- **Problemas transversales**: cada microservicio publicado públicamente debe controlar cuestiones como la autorización y SSL. En muchas situaciones, esas preocupaciones se podrían controlar en un solo nivel para que los microservicios internos se simplificaran.

¿Cuál es el patrón de puerta de enlace de API?

Al diseñar y construir aplicaciones grandes o complejas basadas en microservicios con múltiples aplicaciones cliente, se puede considerar un buen enfoque como el API Gateway. Este patrón es un servicio que proporciona un punto de entrada único para determinados grupos de microservicios. Es similar al patrón de fachada del diseño orientado a objetos, pero en este caso, forma parte de un sistema distribuido. En ocasiones, el patrón de puerta de enlace de API también se conoce como "back-end para front-end" (BFF) porque en la compilación se tienen en cuenta las necesidades de la aplicación cliente.

Por lo tanto, la puerta de enlace de API se encuentra entre las aplicaciones cliente y los microservicios. Funciona como proxy inverso, enrutando las solicitudes de los clientes a los servicios. También puede proporcionar otras características transversales, como la autenticación, la terminación SSL y la memoria caché.

En la figura 4-13 se muestra cómo una puerta de enlace de API personalizada puede caber en una arquitectura simplificada basada en microservicios con solo unos pocos microservicios.

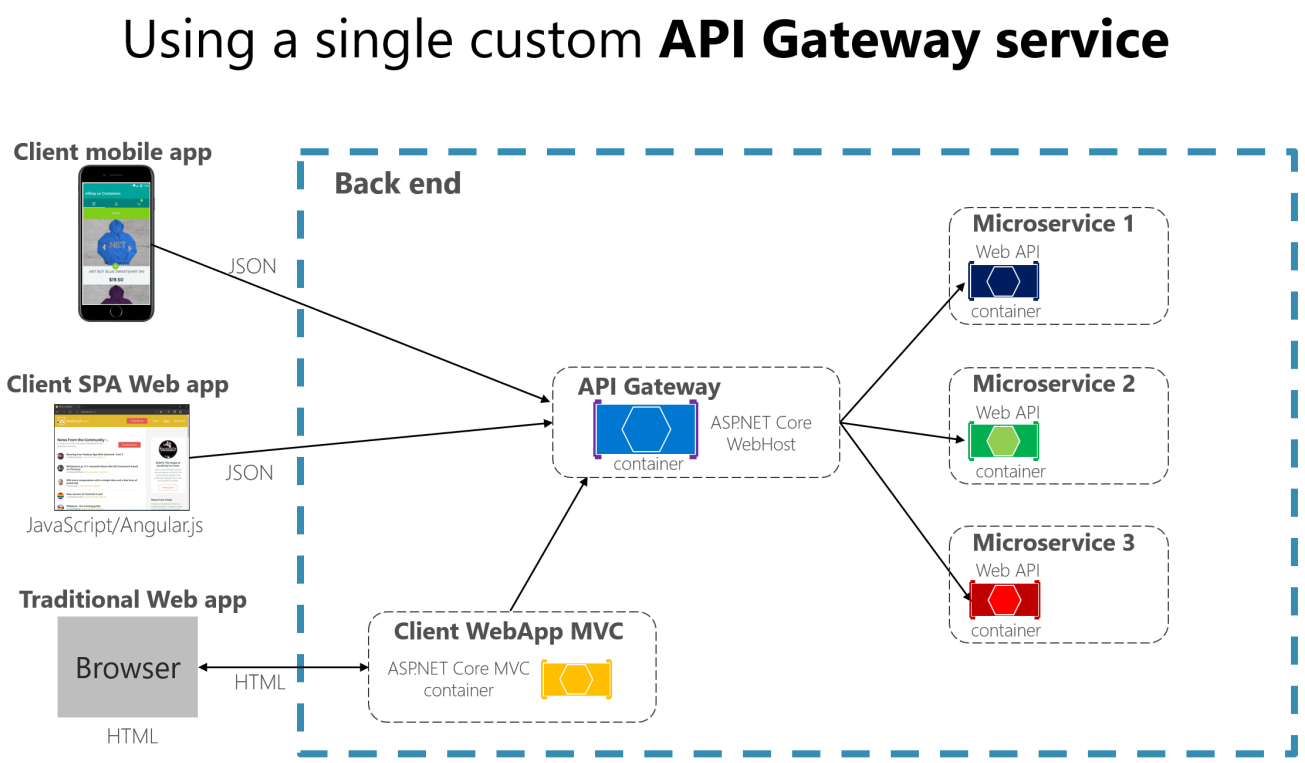


Figura 4-13. Uso de una puerta de enlace de API implementada como un servicio personalizado

Las aplicaciones se conectan a un único punto de conexión, la puerta de enlace de API, que está configurada para reenviar solicitudes a microservicios individuales. En este ejemplo, la puerta de enlace de API se implementaría como un servicio WebHost de ASP.NET core personalizado que se ejecuta como un contenedor.

Es importante resaltar que, en ese diagrama, usaría un único servicio de puerta de enlace de API personalizada que sirve a varias y diversas aplicaciones cliente. Ese hecho puede ser un riesgo importante porque el servicio API Gateway va a crecer y evolucionar en función de muchos requisitos diferentes de las aplicaciones cliente. Finalmente, se sobredimensionará debido a esas necesidades diferentes y, de hecho, podría ser similar a una aplicación monolítica o servicio monolítico. Por eso es muy recomendable dividir la puerta de enlace de API en varios servicios o varias puertas de enlace de API más pequeñas, una por tipo de factor de forma de aplicación cliente, por ejemplo.

Debe tener cuidado al implementar el patrón de puerta de enlace de API. Normalmente, no es una buena idea tener una sola puerta de enlace de API que agregue todos los microservicios internos de tu aplicación. Si lo hace, actúa como agregador monolítico o orquestador y infringe la autonomía del microservicio mediante el acoplamiento de todos los microservicios.

Por lo tanto, las puertas de enlace de API se deben separar en función de los límites empresariales y las aplicaciones cliente y no actuar como un único agregador para todos los microservicios internos.

Al dividir el nivel de API Gateway en varias puertas de enlace de API, si su aplicación tiene varias aplicaciones cliente, puede servir como un punto clave para identificar los diferentes tipos de puertas de enlace de API, permitiéndole tener diferentes fachadas según las necesidades de cada aplicación cliente. Este caso es un patrón denominado "back-end para front-end" (BFF), donde cada puerta de enlace de API puede proporcionar una API distinta adaptada a cada tipo de aplicación cliente, posiblemente basada incluso en el factor de forma de cliente, mediante la implementación de código adaptador específico que llame, de forma subyacente, a varios servicios internos, como se muestra en la imagen siguiente:

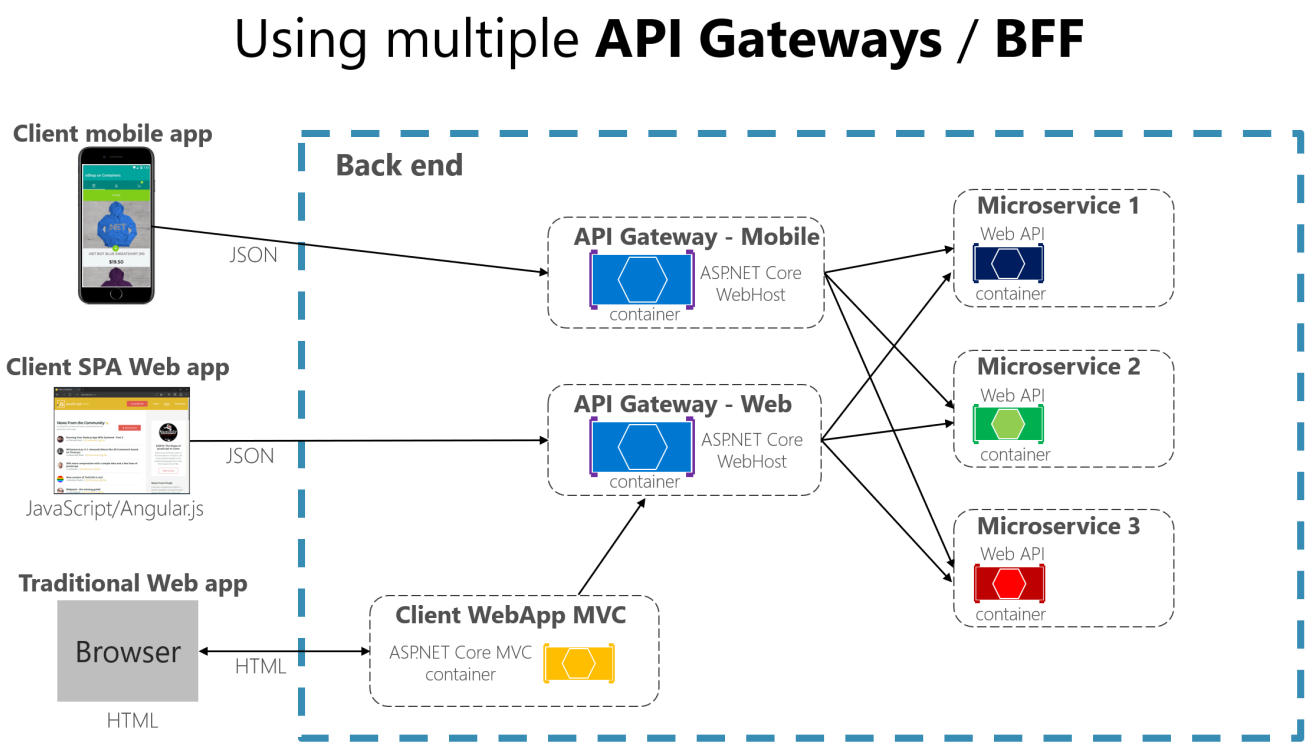


Figura 4-13.1. Uso de varias puertas de enlace de API personalizadas

En la figura 4-13.1 se muestran las puertas de enlace de API segregadas por tipo de cliente; uno para clientes móviles y otro para clientes web. Una aplicación web tradicional se conecta a un microservicio MVC que usa la puerta de enlace de API web. En el ejemplo se muestra una arquitectura simplificada con varias puertas de enlace de API específicas. En este caso, los límites identificados para cada puerta de enlace de API se basan exclusivamente en el patrón "Back-end para front-end" (BFF), por lo que se basa solo en la API necesaria por aplicación cliente. Sin embargo, en aplicaciones más grandes, también debe ir más allá y crear otras puertas de enlace de API basadas en los límites del negocio como un segundo eje de diseño.

Características principales del patrón de puerta de enlace de API

Una puerta de enlace de API puede ofrecer varias características. En función del producto, puede ofrecer características más enriquecidas o más sencillas, pero las características más importantes y fundamentales de cualquier puerta de enlace de API son los siguientes patrones de diseño:

Proxy inverso o enrutamiento de puerta de enlace. La puerta de enlace de API ofrece un proxy inverso para redirigir o enrutar solicitudes (enrutamiento de nivel 7, normalmente solicitudes HTTP) a los puntos de conexión de los microservicios internos. La puerta de enlace proporciona un único punto de conexión o una dirección URL para las aplicaciones cliente y, a continuación, asigna internamente las solicitudes a un grupo de microservicios internos. Esta característica de enrutamiento ayuda a desconectar las aplicaciones cliente de los microservicios, pero también resulta práctica al modernizar una API monolítica colocando la puerta de enlace de API entre la API monolítica y las aplicaciones cliente; de este modo, se pueden agregar nuevas API como nuevos microservicios mientras se sigue usando la API monolítica heredada hasta que se divida en muchos microservicios en el futuro. Debido a la puerta de enlace de API, las aplicaciones cliente no observarán si las API que se usan se implementan como microservicios internos o una API monolítica y, lo que es más importante, al evolucionar y refactorizar la API monolítica en microservicios, gracias al enrutamiento de la puerta de enlace de API, las aplicaciones cliente no se verán afectadas con ningún cambio de URI.

Para obtener más información, consulte Patrón Gateway Routing.

Agregación de solicitudes. Como parte del patrón de puerta de enlace, puede agregar varias solicitudes de cliente (normalmente solicitudes HTTP) destinadas a varios microservicios internos en una sola solicitud de cliente. Este patrón es especialmente conveniente cuando una página o pantalla de cliente necesita información de varios microservicios. Con este enfoque, la

aplicación cliente envía una única solicitud a la puerta de enlace de API que envía varias solicitudes a los microservicios internos y, a continuación, agrega los resultados y devuelve todo a la aplicación cliente. La principal ventaja y el objetivo de este patrón de diseño es reducir el chateo entre las aplicaciones cliente y la API de backend, lo cual es especialmente importante para las aplicaciones remotas fuera del centro de datos donde residen los microservicios, como las aplicaciones móviles o las solicitudes provenientes de aplicaciones SPA que se originan en JavaScript en navegadores remotos del cliente. Para las aplicaciones web normales que realizan las solicitudes en el entorno de servidor (como una aplicación web MVC de ASP.NET Core), este patrón no es tan importante, ya que la latencia es mucho menor que para las aplicaciones cliente remotas.

En función del producto de puerta de enlace de API que use, es posible que pueda realizar esta agregación. Sin embargo, en muchos casos es más flexible crear microservicios de agregación en el ámbito de la puerta de enlace de API, por lo que se define la agregación en el código (es decir, código de C#):

Para obtener más información, consulte Patrón de agregación de Gateway.

Cuestiones transversales o descarga de puerta de enlace. En función de las características que ofrece cada producto de puerta de enlace de API, puede descargar la funcionalidad de microservicios individuales a la puerta de enlace, lo que simplifica la implementación de cada microservicio mediante la consolidación de preocupaciones transversales en un nivel. Este enfoque es especialmente conveniente para las características especializadas que pueden ser complejas de implementar correctamente en cada microservicio interno, como la siguiente funcionalidad:

- Autenticación y autorización
- Integración de la detección de servicios
- Almacenamiento en caché de respuesta
- Directivas de reintento, cortacircuitos, y Calidad de Servicio (QoS)
- Limitación de velocidad
- Equilibrio de carga
- Registro, seguimiento, correlación
- Encabezados, cadenas de consulta y transformación de notificaciones
- Adición a la lista de direcciones IP permitidas

Para obtener más información, consulte Patrón Gateway Offloading.

Uso de productos con características de API Gateway

Puede haber muchos más problemas transversales ofrecidos por los productos de puertas de enlace de API en función de cada implementación. Exploraremos aquí:

- Azure API Management
- Ocelote

Azure API Management

Azure API Management (como se muestra en la figura 4-14) no solo resuelve las necesidades de la puerta de enlace de API, sino que proporciona características como recopilar información de las API. Si usa una solución de API Management, una puerta de enlace de API solo es un componente dentro de esa solución completa de API Management.

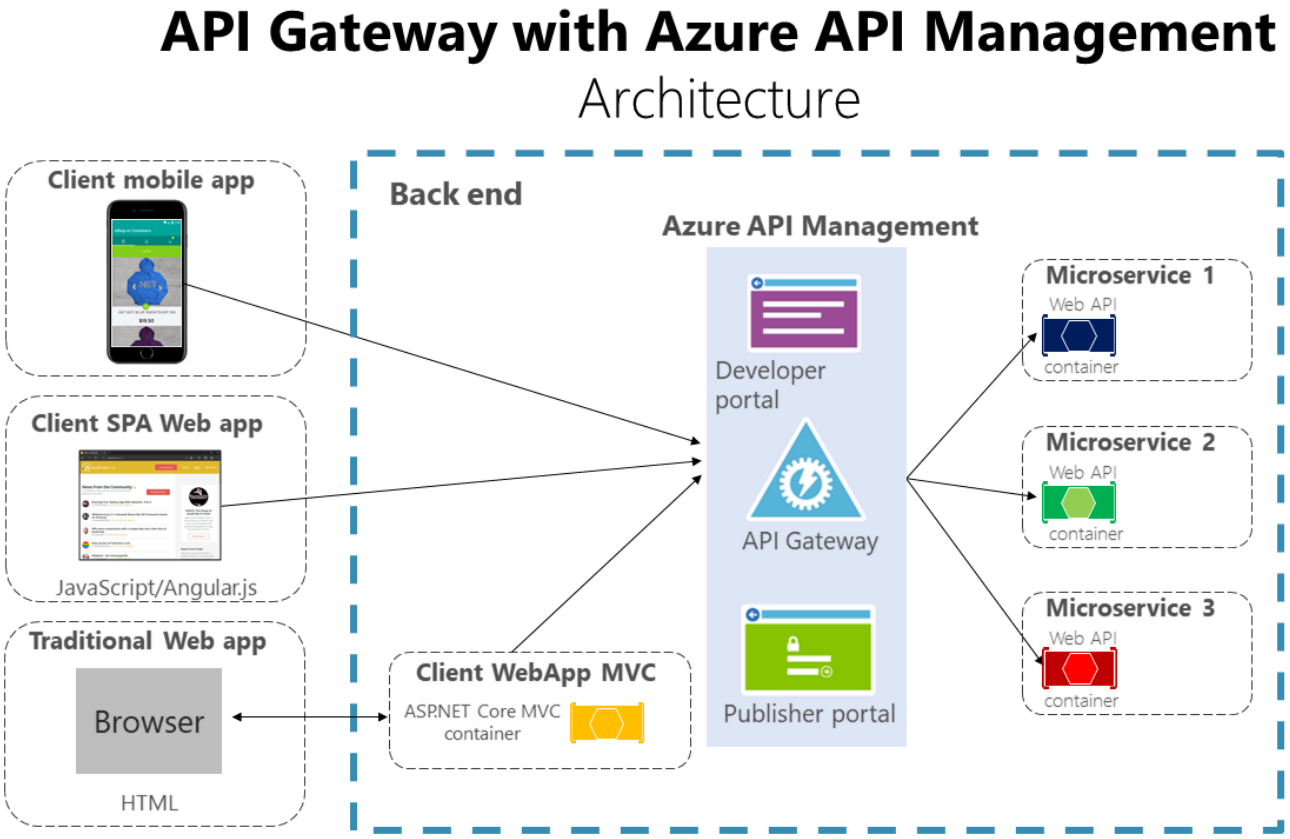


Figura 4-14. Uso de Azure API Management para la puerta de enlace de API

Azure API Management resuelve las necesidades de API Gateway y Management, como el registro, la seguridad, la medición, etc. En este caso, cuando se usa un producto como Azure API Management, el hecho de que puede tener una sola puerta de enlace de API no es tan arriesgada porque estos tipos de puertas de enlace de API son "más delgados", lo que significa que no implementa código de C# personalizado que podría evolucionar hacia un componente monolítico.

Los productos de API Gateway suelen actuar como un proxy inverso para la comunicación de entrada, donde también puede filtrar las API de los microservicios internos además de aplicar

autorización a las API publicadas en una sola capa.

La información proporcionada por un sistema de gestión de API le ayuda a comprender cómo se utilizan sus API y cómo están funcionando. Hacen esta actividad al permitirle ver informes de análisis casi en tiempo real e identificar tendencias que podrían afectar a su negocio. Además, puede tener registros sobre la actividad de solicitud y respuesta para un análisis más en línea y sin conexión.

Con Azure API Management, puede proteger las API mediante una clave, un token y el filtrado de IP. Estas características permiten aplicar cuotas y límites de velocidad flexibles y específicos, modificar la forma y el comportamiento de las API mediante directivas y mejorar el rendimiento con el almacenamiento en caché de respuestas.

En esta guía y la aplicación de ejemplo de referencia (eShopOnContainers), la arquitectura se limita a una arquitectura en contenedores más sencilla y personalizada para centrarse en contenedores sin formato sin usar productos PaaS como Azure API Management. Pero para aplicaciones de microservicio de gran tamaño que se implementan en Microsoft Azure, le animamos a evaluar Azure API Management como base para las puertas de enlace de API en producción.

Ocelote

Ocelot es una puerta de enlace de API ligera, recomendada para enfoques más sencillos. Ocelot es una puerta de enlace de API basada en .NET Core de código abierto especialmente creada para arquitecturas de microservicios que necesitan puntos unificados de entrada en sus sistemas. Es ligero, rápido y escalable y proporciona enrutamiento y autenticación entre muchas otras características.

La razón principal para elegir Ocelot para la aplicación de referencia eShopOnContainers 2.0 es porque Ocelot es una puerta de enlace de API ligera de .NET Core que puede implementar en el mismo entorno de implementación de aplicaciones donde va a implementar los microservicios o contenedores, como un host de Docker, Kubernetes, etc. Y como se basa en .NET Core, es multiplataforma que le permite implementar en Linux o Windows.

Los diagramas anteriores que muestran puertas de enlace de API personalizadas que se ejecutan en contenedores son precisamente cómo puede ejecutar Ocelot en una aplicación basada en contenedores y microservicios.

Además, hay otros muchos productos en el mercado que ofrecen características de puertas de enlace de API, como Apigee, Kong, MuleSoft o WSO2, y otros productos, como Linkerd y Istio, para características de controlador de ingreso de malla de servicio.

Después de las secciones de explicación de patrones y arquitectura iniciales, las secciones siguientes explican cómo implementar puertas de enlace de API con Ocelot.

Desventajas del patrón de puerta de enlace de API

- El inconveniente más importante es que, al implementar una puerta de enlace de API, se acopla ese nivel con los microservicios internos. El acoplamiento como este podría suponer graves dificultades para la aplicación. Clemens Vaster, arquitecto del equipo de Azure Service Bus, hace referencia a esta posible dificultad como "la nueva ESB" en la sesión "Mensajería y microservicios" en GOTO 2016.

- El uso de una puerta de enlace de API de microservicios crea un posible punto de error adicional.

- Una pasarela de API puede provocar un incremento en el tiempo de respuesta debido a la llamada de red adicional. Pero esta llamada adicional suele tener menor impacto que una interfaz de cliente que realiza demasiadas llamadas a los microservicios internos.

- Si no se escala horizontalmente de manera correcta, la puerta de enlace de API puede dar lugar a un cuello de botella.

- Una puerta de enlace de API requiere un costo de desarrollo adicional y un mantenimiento futuro si incluye lógica personalizada y agregación de datos. Los desarrolladores deben actualizar la puerta de enlace de API para exponer los puntos de conexión de cada microservicio. Además, los cambios de implementación en los microservicios internos pueden provocar cambios de código en el nivel de puerta de enlace de API. Sin embargo, si la puerta de enlace de API solo aplica seguridad, registro y control de versiones (como cuando se usa Azure API Management), es posible que este costo de desarrollo adicional no se aplique.

- Si la puerta de enlace de API ha sido desarrollada por un único equipo, puede haber un cuello de botella de desarrollo. Este aspecto es otro motivo por el que un mejor enfoque es tener varias puertas de enlace de API específicas definidas que responden a diferentes necesidades de cliente. También puede separar la puerta de enlace de API internamente en varias áreas o capas que pertenecen a los distintos equipos que trabajan en los microservicios internos.

Recursos adicionales

- **Chris Kemp. Pattern: API Gateway / Backend for Front-End (Patrón: puerta de enlace de API o back-end para front-end)**
  https://microservices.io/patterns/apigateway.html

- **Patrón de puerta de enlace de API**
  https://learn.microsoft.com/azure/architecture/microservices/gateway
- **Patrón de agregación y composición**
  https://microservices.io/patterns/data/api-composition.html
- **Azure API Management**
  https://azure.microsoft.com/services/api-management/
- **Udi Dahan. Composición orientada a servicios**
  https://udidahan.com/2014/07/30/service-oriented-composition-with-video/
- **Clemens Vasters. Mensajería y microservicios en GOTO 2016 (vídeo)**
  https://www.youtube.com/watch?v=rXi5CLjIQ9k
- **Puerta de enlace API en pocas palabras** (serie de tutoriales de puerta de enlace de API de ASP.NET Core)
  https://www.pogsdotnet.com/2018/08/api-gateway-in-nutshell.html

Siguiente anterior

Comunicación en una arquitectura de microservicios

19/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En una aplicación monolítica que se ejecuta en un único proceso, los componentes se invocan entre sí mediante llamadas de función o método de nivel de lenguaje. Estos se pueden acoplar fuertemente si va a crear objetos con código (por ejemplo, `new ClassName()`), o se pueden invocar de forma desacoplada si usa la inserción de dependencias haciendo referencia a abstracciones en lugar de instancias de objeto concretas. En cualquier caso, los objetos se ejecutan dentro del mismo proceso. El mayor desafío al cambiar de una aplicación monolítica a una aplicación basada en microservicios radica en cambiar el mecanismo de comunicación. Una conversión directa de llamadas de método en curso a llamadas RPC a servicios dará lugar a una comunicación extensa y no eficaz con un mal rendimiento en entornos distribuidos. Los desafíos de diseñar el sistema distribuido correctamente son lo suficientemente conocidos que incluso hay un canon conocido como fallacies de computación distribuida que enumera las suposiciones que los desarrolladores suelen hacer al pasar de diseños monolíticos a distribuidos.

No hay una solución, pero varias. Una solución implica aislar los microservicios empresariales tanto como sea posible. Luego se usa la comunicación asincrónica entre los microservicios internos y se sustituye la comunicación específica típica de la comunicación en proceso entre objetos por la comunicación general. Para ello, puede agrupar llamadas y devolver datos que agregan los resultados de varias llamadas internas al cliente.

Una aplicación basada en microservicios es un sistema distribuido que se ejecuta en varios procesos o servicios, normalmente incluso en varios servidores o hosts. Cada instancia de servicio suele ser un proceso. Por lo tanto, los servicios deben interactuar mediante un protocolo de comunicación entre procesos, como HTTP, AMQP o un protocolo binario como TCP, en función de la naturaleza de cada servicio.

La comunidad de microservicios promueve la filosofía de "puntos de conexión inteligentes y tuberías tontas ". Este eslogan fomenta un diseño lo más desacoplado posible entre microservicios y lo más cohesivo posible dentro de un único microservicio. Como se explicó anteriormente, cada microservicio posee sus propios datos y su propia lógica de dominio. Pero los microservicios que componen una aplicación de extremo a extremo suelen estar simplemente coreografiados mediante comunicaciones REST, en lugar de protocolos complejos como WS-* y de comunicaciones flexibles impulsadas por eventos en lugar de orquestadores centralizados de procesos de negocio.

Los dos protocolos usados habitualmente son solicitud/respuesta HTTP con API de recursos (al consultar la mayoría de todos) y mensajería asincrónica ligera al comunicar actualizaciones entre varios microservicios. Estos se explican con más detalle en las secciones siguientes.

Tipos de comunicación

El cliente y los servicios pueden comunicarse a través de muchos tipos diferentes de comunicación, cada uno de los cuales tiene como destino un escenario y objetivos diferentes. Inicialmente, esos tipos de comunicaciones se pueden clasificar en dos ejes.

El primer eje define si el protocolo es sincrónico o asincrónico:

- Protocolo sincrónico. HTTP es un protocolo sincrónico. El cliente envía una solicitud y espera una respuesta del servicio. Eso es independiente de la ejecución de código de cliente, que puede ser sincrónica (el subproceso está bloqueado) o asincrónica (el subproceso no está bloqueado y al final la respuesta llega a una devolución de llamada). El punto importante aquí es que el protocolo (HTTP/HTTPS) es sincrónico y el código de cliente solo puede continuar su tarea cuando recibe la respuesta del servidor HTTP.

- Protocolo asincrónico. Otros protocolos como AMQP (un protocolo compatible con muchos sistemas operativos y entornos en la nube) usan mensajes asincrónicos. Normalmente, el código de cliente o el remitente del mensaje no esperan una respuesta. Simplemente envía el mensaje como al enviar un mensaje a una cola de RabbitMQ o a cualquier otro agente de mensajes.

El segundo eje define si la comunicación tiene un solo receptor o varios receptores:

- Receptor único. Cada solicitud debe procesarse exactamente por un receptor o servicio. Un ejemplo de este tipo de comunicación es el patrón Command.

- Varios receptores. Cada solicitud puede ser procesada por entre cero y varios receptores. Este tipo de comunicación debe ser asincrónico. Un ejemplo es el mecanismo de publicación y suscripción que se usa en patrones como la arquitectura controlada por eventos. Esto se basa en una interfaz de bus de eventos o un agente de mensajes al propagar actualizaciones de datos entre varios microservicios a través de eventos; normalmente se implementa a través de un bus de servicio o un artefacto similar, como Azure Service Bus, mediante temas y suscripciones.

Una aplicación basada en microservicios suele usar una combinación de estos estilos de comunicación. El tipo más común es la comunicación de un solo receptor con un protocolo sincrónico como HTTP/HTTPS al invocar un servicio HTTP de API web normal. Los microservicios también suelen usar protocolos de mensajería para la comunicación asincrónica entre microservicios.

Estos ejes son buenos para conocerlos, para que tenga claridad sobre los posibles mecanismos de comunicación, pero no son las preocupaciones importantes al construir microservicios. Ni la naturaleza asincrónica de la ejecución de subprocesos de cliente ni la naturaleza asincrónica del protocolo seleccionado son los puntos importantes al integrar microservicios. Lo que *es importante es* poder integrar los microservicios de forma asincrónica a la vez que se mantiene la independencia de los microservicios, como se explica en la sección siguiente.

La integración asincrónica de microservicios exige la autonomía del microservicio

Como se mencionó, el punto importante al crear una aplicación basada en microservicios es la forma en que se integran los microservicios. Lo ideal es intentar minimizar la comunicación entre los microservicios internos. Cuantos menos comunicaciones haya entre microservicios, mejor. En muchos casos, deberá integrar los microservicios de alguna forma. Cuando tenga que hacerlo, la regla crítica aquí es que la comunicación entre los microservicios debe ser asincrónica. Esto no significa que tenga que usar un protocolo específico (por ejemplo, mensajería asincrónica frente a HTTP sincrónico). Solo significa que la comunicación entre microservicios solo se debe realizar mediante la propagación de datos de forma asincrónica, pero intente no depender de otros microservicios internos como parte de la operación de solicitud/respuesta HTTP del servicio inicial.

Si es posible, nunca dependa de la comunicación sincrónica (solicitud/respuesta) entre varios microservicios, ni siquiera para las consultas. El objetivo de cada microservicio es ser autónomo y estar disponible para el consumidor del cliente, incluso si los demás servicios que forman

parte de la aplicación de un extremo a otro están inactivos o están en mal estado. Si cree que necesita realizar una llamada de un microservicio a otros microservicios (como realizar una solicitud HTTP para una consulta de datos) para poder proporcionar una respuesta a una aplicación cliente, tiene una arquitectura que no será resistente cuando se produzca un error en algunos microservicios.

Además, tener dependencias HTTP entre microservicios, como al crear ciclos de solicitud y respuesta largos con cadenas de solicitudes HTTP, como se muestra en la primera parte de la figura 4-15, no solo hace que los microservicios no sean autónomos, sino que también su rendimiento se vea afectado en cuanto uno de los servicios de esa cadena no funciona bien.

Cuanto más agregue dependencias sincrónicas entre microservicios, como las solicitudes de consulta, peor será el tiempo de respuesta general para las aplicaciones cliente.

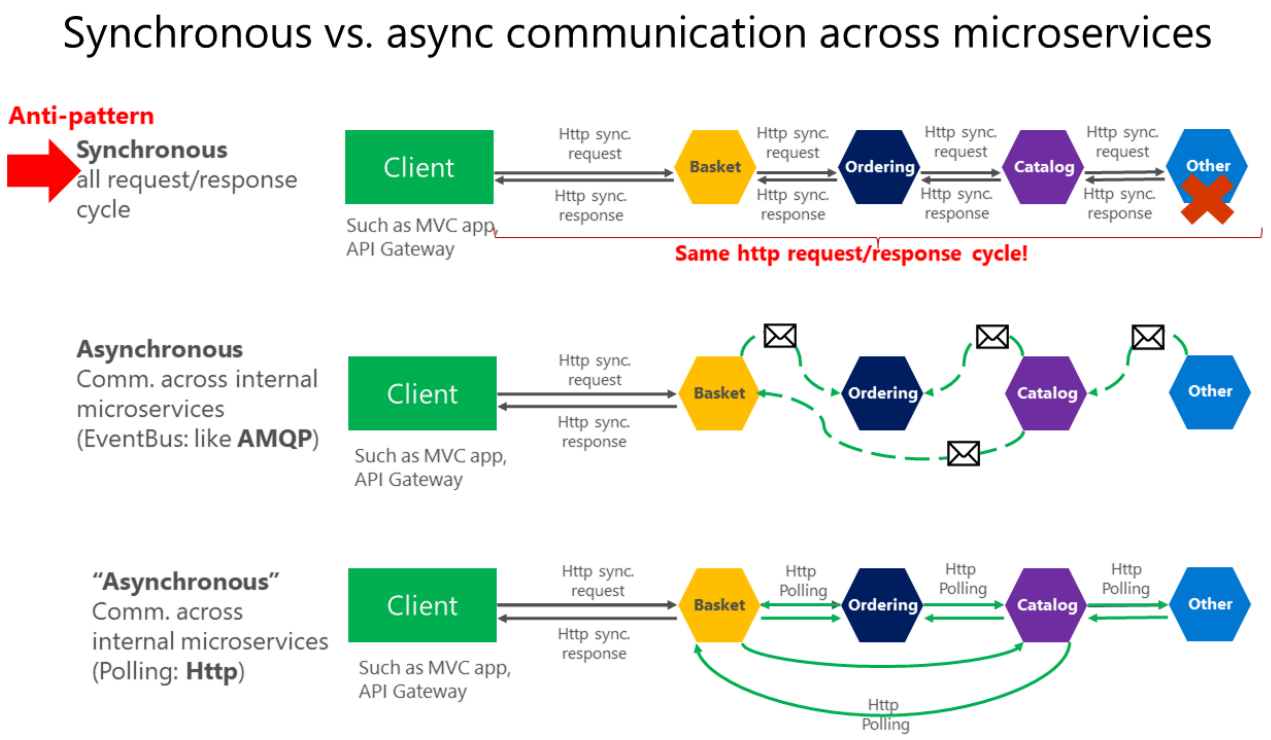


Figura 4-15. Antipatrones y patrones en la comunicación entre microservicios

Como se muestra en el diagrama anterior, en la comunicación sincrónica se crea una "cadena" de solicitudes entre microservicios al atender la solicitud de cliente. Esto es un antipatrón. En los microservicios de comunicación asincrónica, use mensajes asincrónicos o sondeo http para comunicarse con otros microservicios, pero la solicitud de cliente se atiende inmediatamente.

Si el microservicio necesita generar una acción adicional en otro microservicio, si es posible, no realice esa acción de forma sincrónica y como parte de la operación original de solicitud y respuesta del microservicio. En su lugar, házlo de forma asincrónica (mediante mensajería asincrónica o eventos de integración, colas, etc.). Pero, tanto como sea posible, no invoque la acción de forma sincrónica como parte de la operación original de solicitud y respuesta sincrónica.

Y por último (y aquí es donde surgen la mayoría de los problemas al desarrollar microservicios), si tu microservicio inicial necesita datos que originalmente pertenecen a otros microservicios, no dependas de realizar solicitudes sincrónicas para esos datos. En su lugar, replique o propague esos datos (solo los atributos que necesita) en la base de datos del servicio inicial mediante la coherencia final (normalmente mediante eventos de integración, como se explica en las secciones próximas).

Como se indicó anteriormente en la sección Identificación de los límites del modelo de dominio para cada microservicio , duplicar algunos datos en varios microservicios no es un diseño incorrecto; por el contrario, al hacerlo, puede traducir los datos en el idioma o términos específicos de ese dominio adicional o contexto enlazado. Por ejemplo, en la aplicación eShopOnContainers tiene un microservicio denominado `identity-api` que se encarga de la mayoría de los datos del usuario con una entidad denominada `User`. Sin embargo, cuando necesite almacenar datos sobre el usuario dentro del `Ordering` microservicio, lo almacenará como una entidad diferente denominada `Buyer`. La `Buyer` entidad comparte la misma identidad con la entidad original `User` , pero puede tener solo los pocos atributos necesarios para el `Ordering` dominio y no todo el perfil de usuario.

Puede usar cualquier protocolo para comunicar y propagar datos de forma asincrónica entre microservicios con el fin de tener coherencia final. Como se mencionó, podría usar eventos de integración mediante un bus de eventos o un agente de mensajes, o incluso podría usar HTTP sondeando los otros servicios en su lugar. No importa. La regla importante es no crear dependencias sincrónicas entre los microservicios.

En las secciones siguientes se explican los varios estilos de comunicación que puede considerar usar en una aplicación basada en microservicios.

Estilos de comunicación

Hay muchos protocolos y opciones que puede usar para la comunicación, en función del tipo de comunicación que quiera usar. Si usa un mecanismo de comunicación sincrónico basado en solicitudes o respuestas, los protocolos como los enfoques HTTP y REST son los más comunes, especialmente si está publicando los servicios fuera del host de Docker o el clúster de microservicios. Si está comunicándose entre los servicios de manera interna (dentro del entorno Docker o del clúster de microservicios), también podría querer utilizar mecanismos de comunicación en formato binario (como WCF mediante TCP y formato binario). Como alternativa, puede usar mecanismos de comunicación asincrónicos basados en mensajes, como AMQP.

También hay varios formatos de mensaje como JSON o XML, o incluso formatos binarios, que pueden ser más eficaces. Si el formato binario elegido no es un estándar, probablemente no es

una buena idea publicar públicamente los servicios con ese formato. Puede usar un formato no estándar para la comunicación interna entre los microservicios. Puede hacerlo al comunicarse entre microservicios dentro del host de Docker o el clúster de microservicios (por ejemplo, orquestadores de Docker) o para aplicaciones cliente propietarias que se comunican con los microservicios.

Comunicación de solicitud/respuesta con HTTP y REST

Cuando un cliente usa la comunicación de solicitud/respuesta, envía una solicitud a un servicio y, a continuación, el servicio procesa la solicitud y devuelve una respuesta. La comunicación de solicitud y respuesta es especialmente adecuada para consultar datos para una interfaz de usuario en tiempo real (una interfaz de usuario activa) desde aplicaciones cliente. Por lo tanto, en una arquitectura de microservicios probablemente usará este mecanismo de comunicación para la mayoría de las consultas, como se muestra en la figura 4-16.

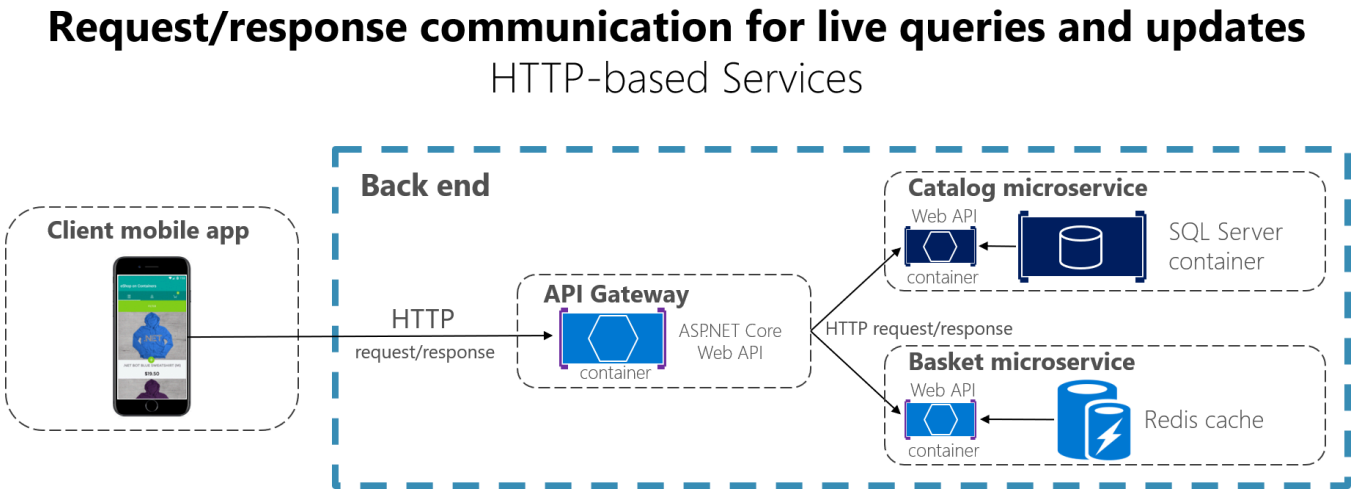


Figura 4-16. Uso de la comunicación de solicitud/respuesta HTTP (sincrónica o asincrónica)

Cuando un cliente usa la comunicación de solicitud/respuesta, se supone que la respuesta llegará en un breve tiempo, normalmente menos de un segundo o unos segundos como máximo. Para las respuestas retrasadas, debe implementar la comunicación asincrónica basada en patrones de mensajería y tecnologías de mensajería, que es un enfoque diferente que se explica en la sección siguiente.

Un estilo arquitectónico popular para la comunicación de solicitud/respuesta es REST. Este enfoque se basa en, y estrechamente acoplado, el protocolo HTTP, adoptando verbos HTTP como GET, POST y PUT. REST es el enfoque de comunicación arquitectónico más usado al crear servicios. Puede implementar servicios REST al desarrollar ASP.NET servicios de API web core.

Hay un valor adicional al usar servicios REST HTTP como lenguaje de definición de interfaz. Por ejemplo, si usa metadatos de Swagger para describir la API de servicio, puede usar

herramientas que generan código auxiliar de cliente que puede detectar y usar directamente los servicios.

Recursos adicionales

- **Martin Fowler. Modelo de Madurez de Richardson** Descripción del modelo REST. https://martinfowler.com/articles/richardsonMaturityModel.html
- **Swagger** El sitio oficial. https://swagger.io/

Notificaciones push y comunicación en tiempo real basada en HTTP

Otra posibilidad (normalmente con fines diferentes a REST) es una comunicación de uno a varios en tiempo real con marcos de nivel superior, como ASP.NET SignalR y protocolos como WebSockets.

Como se muestra en la figura 4-17, la comunicación HTTP en tiempo real significa que puede tener código de servidor que inserta contenido en clientes conectados a medida que los datos estén disponibles, en lugar de tener que esperar al servidor a que un cliente solicite nuevos datos.

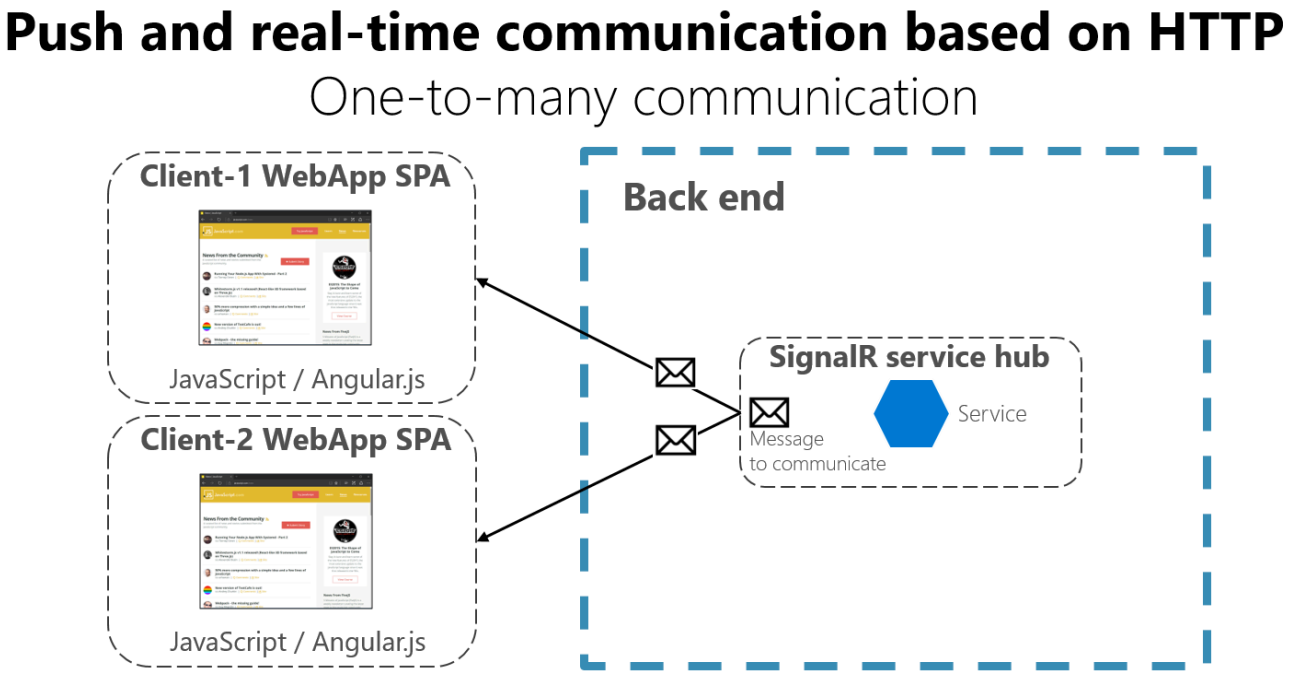


Figura 4-17. Comunicación de mensajes asincrónica de uno a varios en tiempo real

SignalR es una buena manera de lograr la comunicación en tiempo real para transmitir contenido a los clientes desde un servidor backend. Puesto que la comunicación está en

tiempo real, las aplicaciones cliente muestran los cambios casi al instante. Normalmente, esto se controla mediante un protocolo como WebSockets, mediante muchas conexiones de WebSockets (una por cliente). Un ejemplo típico es cuando un servicio comunica un cambio en la puntuación de un juego deportivo a muchas aplicaciones web cliente simultáneamente.

Siguiente anterior

Comunicación asincrónica basada en mensajes

Artículo • 10/05/2023

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en **Documentación de .NET** o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

La mensajería asincrónica y la comunicación controlada por eventos son fundamentales para propagar cambios entre varios microservicios y sus modelos de dominio relacionados. Como se mencionó anteriormente en la descripción de los microservicios y los contextos delimitados (BC), los modelos (de usuario, cliente, producto, cuenta, etc.) pueden tener diferentes significados para distintos microservicios o BC. Esto significa que, cuando se producen cambios, se necesita alguna manera de conciliarlos entre los diferentes modelos. Una solución es la coherencia final y la comunicación controlada por eventos basada en la mensajería asincrónica.

Cuando se usa la mensajería, los procesos se comunican mediante el intercambio de mensajes de forma asincrónica. Un cliente ejecuta una orden o una solicitud a un servicio mediante el envío de un mensaje. Si el servicio tiene que responder, envía un mensaje diferente al cliente. Como se trata de una comunicación basada en mensajes, el cliente asume que la respuesta no se recibirá inmediatamente y que es posible que no haya ninguna respuesta.

Un mensaje está compuesto por un encabezado (metadatos como información de identificación o seguridad) y un cuerpo. Normalmente, los mensajes se envían a través de protocolos asincrónicos como AMQP.

La infraestructura preferida para este tipo de comunicación en la comunidad de microservicios es un agente de mensajes ligero, que es diferente a los agentes grandes y orquestadores que se usan en SOA. En un agente de mensajes ligero, la infraestructura suele ser "simple" y solo actúa como un agente de mensajes, con implementaciones sencillas como RabbitMQ o un Service Bus escalable en la nube como Azure Service Bus. En este escenario, la mayoría de las ideas "inteligentes" siguen existiendo en los puntos de conexión que generan y consumen mensajes, es decir, en los microservicios.

Otra regla que debe intentar seguir, tanto como sea posible, es usar la mensajería asincrónica solo entre los servicios internos y la comunicación sincrónica (como HTTP) solo desde las aplicaciones cliente a los servicios front-end (puertas de enlace de API y el primer nivel de microservicios).

Hay dos tipos de comunicación de mensajería asincrónica: la comunicación basada en mensajes de receptor único y la comunicación basada en mensajes de varios receptores. En las siguientes secciones se proporcionan detalles sobre los dos tipos.

Comunicación basada en mensajes de receptor único

La comunicación asincrónica basada en mensajes con un receptor único significa que hay una comunicación punto a punto que entrega un mensaje a exactamente uno de los consumidores que está leyendo en el canal, y que el mensaje solo se procesa una vez. Pero hay situaciones especiales. Por ejemplo, en un sistema de nube que intenta recuperarse automáticamente de los errores, el mismo mensaje se podría enviar varias veces. Debido a problemas de red o de otro tipo, el cliente tiene que poder volver a intentar el envío de los mensajes y el servidor tiene que implementar una operación que sea idempotente para procesar un mensaje concreto una sola vez.

La comunicación basada en mensajes de receptor único es especialmente idónea para enviar comandos asincrónicos de un microservicio a otro, como se muestra en la figura 4-18.

Una vez que se inicia el envío mediante la comunicación basada en mensajes (ya sea a través de comandos o eventos), no se debe mezclar con la comunicación sincrónica de HTTP.

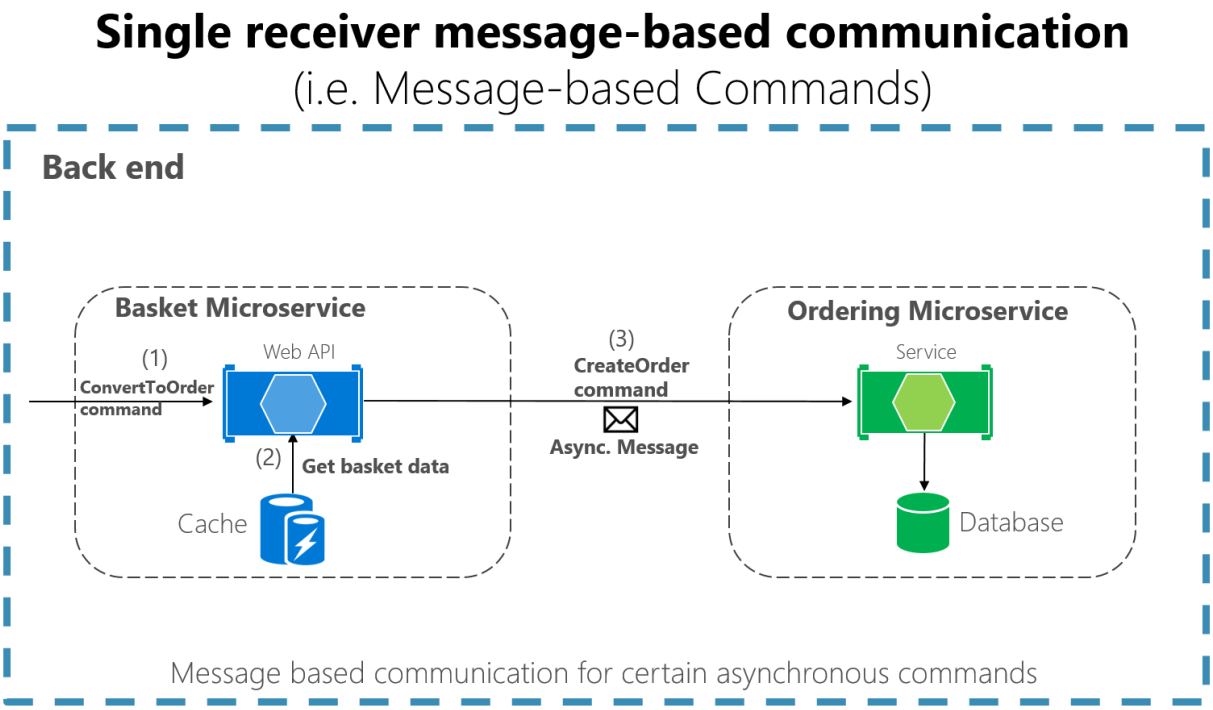


Figura 4-18. Un único microservicio en el que se recibe un mensaje asincrónico

Cuando los comandos proceden de aplicaciones cliente, se pueden implementar como comandos sincrónicos de HTTP. Use comandos basados en mensajes cuando necesite una mayor escalabilidad o cuando ya se encuentre en un proceso empresarial basado en mensajes.

Comunicación basada en mensajes de varios receptores

Como un enfoque más flexible, es posible que también le interese usar un mecanismo de publicación y suscripción para que la comunicación desde el remitente esté disponible para microservicios de suscriptor adicionales o para aplicaciones externas. Por tanto, le ayudará seguir el principio de abierto y cerrado en el servicio de envío. De este modo, se pueden agregar suscriptores adicionales en el futuro sin necesidad de modificar el servicio del remitente.

Cuando se usa una comunicación de publicación y suscripción, es posible que se use una interfaz de bus de eventos para publicar eventos en cualquier suscriptor.

Comunicación asincrónica controlada por eventos

Cuando se usa la comunicación asincrónica controlada por eventos, un microservicio publica un evento de integración cuando sucede algo dentro de su dominio y otro

microservicio debe ser consciente de ello, como por ejemplo un cambio de precio en un microservicio del catálogo de productos. Los microservicios adicionales se suscriben a los eventos para poder recibirlos de forma asincrónica. Cuando esto sucede, es posible que los receptores actualicen sus propias entidades de dominio, lo que puede provocar la publicación de más eventos de integración. Este sistema de publicación/suscripción se realiza mediante una implementación de un bus de eventos. El bus de eventos se puede diseñar como una abstracción o interfaz, con la API que se necesita para suscribirse o cancelar la suscripción a los eventos y para publicarlos. El bus de eventos también puede tener una o más implementaciones basadas en cualquier agente entre procesos y de mensajería, como una cola de mensajes o un Service Bus que admita la comunicación asincrónica y un modelo de publicación y suscripción.

Si un sistema usa la coherencia final controlada por eventos de integración, se recomienda que este enfoque sea transparente para el usuario final. El sistema no debe usar un enfoque que imite a los eventos de integración, como SignalR o sistemas de sondeo desde el cliente. El usuario final y el propietario de la empresa tienen que adoptar explícitamente la coherencia final en el sistema y saber que, en muchos casos, la empresa no tiene ningún problema con este enfoque, siempre que sea explícito. Este enfoque es importante porque los usuarios pueden esperar ver algunos resultados inmediatamente y es posible que esto no pase con la coherencia final.

Como se indicó anteriormente en la sección Desafíos y soluciones para la administración de datos distribuidos, se pueden usar eventos de integración para implementar tareas de negocio que abarquen varios microservicios. Por tanto, tendrá coherencia final entre dichos servicios. Una transacción con coherencia final se compone de una colección de acciones distribuidas. En cada acción, el microservicio relacionado actualiza una entidad de dominio y publica otro evento de integración que genera la siguiente acción dentro de la misma tarea empresarial descentralizada.

Un punto importante es que es posible que le interese comunicarse con varios microservicios que estén suscritos al mismo evento. Para ello, puede usar la mensajería de publicación y suscripción basada en la comunicación controlada por eventos, como se muestra en la figura 4-19. Este mecanismo de publicación y suscripción no es exclusivo de la arquitectura de microservicios. Es similar a la forma en que deben comunicarse los contextos delimitados en DDD o a la forma en que se propagan las actualizaciones desde la base de datos de escritura a la de lectura en el modelo de arquitectura Command and Query Responsibility Segregation (CQRS) (Segregación de responsabilidades de comandos y consultas). El objetivo es tener coherencia final entre varios orígenes de datos en el sistema distribuido.

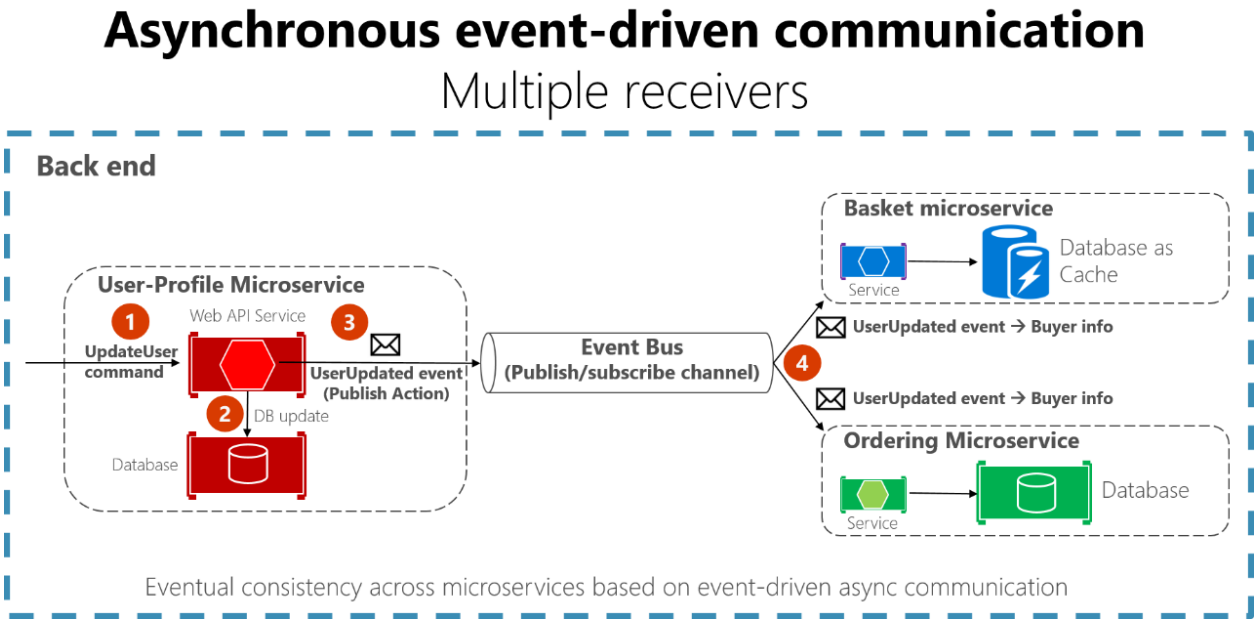


Figura 4-19. Comunicación asincrónica de mensajes controlada por eventos

En la comunicación controlada por eventos asincrónica, un microservicio publica los eventos en un bus de eventos y muchos microservicios pueden suscribirse a él para recibir una notificación y actuar en consecuencia. La implementación determinará qué protocolo se va a usar para las comunicaciones basadas en mensajes y controladas por eventos. AMQP puede ayudar a lograr una comunicación en cola confiable.

Al usar un bus de eventos, es posible que le interese usar una capa de abstracción (como una interfaz de bus de eventos) basada en una implementación relacionada en las clases con código que use la API de un agente de mensajes como RabbitMQ o un Service Bus como Azure Service Bus con Topics. Como alternativa, es posible que le interese usar un Service Bus de nivel superior como NServiceBus, MassTransit o Brighter para articular el bus de eventos y el sistema de publicación y suscripción.

Una nota sobre las tecnologías de mensajería destinadas a sistemas de producción

Las tecnologías de mensajería disponibles para implementar el bus de eventos abstractos se encuentran en distintos niveles. Por ejemplo, productos como RabbitMQ (un transporte de agente de mensajería) y Azure Service Bus se colocan en un nivel inferior a otros productos como NServiceBus, MassTransit o Brighter, que pueden trabajar sobre RabbitMQ y Azure Service Bus. La elección depende de la cantidad de características enriquecidas en el nivel de aplicación y de la escalabilidad de serie que necesite para la aplicación. Para implementar solamente un bus de eventos de prueba de concepto para el entorno de desarrollo, como se ha hecho en el ejemplo

eShopOnContainers, una implementación sencilla sobre RabbitMQ que se ejecute en un contenedor de Docker podría ser suficiente.

Pero para sistemas decisivos y de producción que necesiten una gran escalabilidad, es posible que quiera probar Azure Service Bus. Para las abstracciones generales y las características que facilitan el desarrollo de aplicaciones distribuidas, se recomienda evaluar otros Service Bus comerciales y de código abierto, como NServiceBus, MassTransit y Brighter. Por supuesto, puede crear sus propias características de Service Bus sobre tecnologías de nivel inferior como RabbitMQ y Docker. Pero ese trabajo podría ser muy costoso para una aplicación empresarial personalizada.

Publicación de forma resistente en el bus de eventos

Un desafío al implementar una arquitectura controlada por eventos entre varios microservicios es cómo actualizar de manera atómica el estado en el microservicio original mientras se publica de forma resistente su evento de integración relacionado en el bus de eventos, en cierta medida en función de las transacciones. Las siguientes son algunas maneras de lograr esta funcionalidad, aunque podría haber enfoques adicionales.

- Uso de una cola transaccional (basada en DTC) como MSMQ. (Pero es un método heredado).
- Uso de la minería del registro de transacciones.
- Uso del patrón de orígenes de eventos completo.
- Uso del patrón de bandeja de salida: una tabla de base de datos transaccional como una cola de mensajes que será la base para un componente de creador de eventos que creará el evento y lo publicará.

Para obtener una descripción más completa de las dificultades de este espacio, incluido el modo en que los mensajes con datos potencialmente incorrectos pueden terminar publicándose, consulte Plataforma de datos para cargas de trabajo críticas en Azure: se deben procesar todos los mensajes.

Temas adicionales que se deben tener en cuenta al usar la comunicación asincrónica son la idempotencia y la desduplicación de los mensajes. Estos temas se describen en la sección Implementación de la comunicación basada en eventos entre microservicios (eventos de integración) más adelante en esta guía.

Recursos adicionales

- **Mensajería controlada por eventos**
  https://patterns.arcitura.com/soa-patterns/design_patterns/event_driven_messaging
- **Canal de publicación y suscripción**
  https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
- **Udi Dahan. CQRS aclarado**
  https://udidahan.com/2009/12/09/clarified-cqrs/
- **Segregación de responsabilidades de consultas y comandos (CQRS)**
  https://learn.microsoft.com/azure/architecture/patterns/cqrs
- **Comunicación entre contextos delimitados**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591572(v=pandp.10)
- **Coherencia final**
  https://en.wikipedia.org/wiki/Eventual_consistency
- **Jimmy Bogard. Refactoring Towards Resilience: Evaluating Coupling (Refactorización hacia la resiliencia: evaluación del acoplamiento)**
  https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/

Anterior Siguiente

Creación, evolución y control de versiones de las API y contratos de microservicios

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Una API de microservicios es un contrato entre el servicio y sus clientes. Podrá evolucionar un microservicio de forma independiente solo si no interrumpe su contrato de API, por lo que el contrato es tan importante. Si cambia el contrato, afectará a las aplicaciones cliente o a la puerta de enlace de API.

La naturaleza de la definición de api depende del protocolo que use. Por ejemplo, si usa mensajería, como AMQP, la API consta de los tipos de mensaje. Si usa servicios HTTP y RESTful, la API consta de las direcciones URL y los formatos JSON de solicitud y respuesta.

Sin embargo, incluso si usted es cuidadoso con su contrato inicial, una API de servicio tendrá que cambiar con el tiempo. Cuando esto sucede, y especialmente si la API es una API pública consumida por varias aplicaciones cliente, normalmente no puede obligar a todos los clientes a actualizar a su nuevo contrato de API. Normalmente, es necesario implementar incrementalmente nuevas versiones de un servicio de forma que las versiones antiguas y nuevas de un contrato de servicio se ejecuten simultáneamente. Por lo tanto, es importante tener una estrategia para el control de versiones del servicio.

Cuando los cambios en la API son pequeños, como si agrega atributos o parámetros a la API, los clientes que usan una API anterior deben cambiar y trabajar con la nueva versión del

servicio. Es posible que pueda proporcionar valores predeterminados para los atributos que faltan que sean necesarios y que los clientes puedan omitir los atributos de respuesta adicionales.

Sin embargo, a veces es necesario realizar cambios importantes e incompatibles en una API de servicio. Dado que es posible que no pueda forzar que las aplicaciones o servicios cliente se actualicen inmediatamente a la nueva versión, un servicio debe admitir versiones anteriores de la API durante algún período. Si usa un mecanismo basado en HTTP como REST, un enfoque consiste en insertar el número de versión de la API en la dirección URL o en un encabezado HTTP. Después, puede decidir entre implementar ambas versiones del servicio simultáneamente dentro de la misma instancia de servicio o implementar instancias diferentes que cada una controla una versión de la API. Un buen enfoque para esta funcionalidad es el patrón de mediador (por ejemplo, biblioteca MediatR) para desacoplar las distintas versiones de implementación en controladores independientes.

Por último, si usa una arquitectura REST, Hypermedia es la mejor solución para el control de versiones de los servicios y permitir API en constante evolución.

Recursos adicionales

- **Scott Hanselman. Versionado de la API web RESTful de ASP.NET Core hecho fácil**
  https://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx
- **Control de versiones de una API web RESTful**
  https://learn.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api
- **Roy Fielding. Control de versiones, Hipermedia y REST**
  https://www.infoq.com/articles/roy-fielding-on-versioning

Siguiente anterior

Direccionabilidad de microservicios y registro de servicios

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Cada microservicio tiene un nombre único (URL) que se usa para resolver su ubicación. El microservicio debe ser direccionable, independientemente de dónde se ejecute. Si tiene que pensar en qué equipo ejecuta un microservicio determinado, las cosas pueden ir mal rápidamente. De la misma manera que DNS resuelve una dirección URL en un equipo determinado, el microservicio debe tener un nombre único para que se pueda detectar su ubicación actual. Los microservicios necesitan nombres direccionables que los hagan independientes de la infraestructura en la que se ejecutan. Este enfoque implica que hay una interacción entre cómo se implementa el servicio y cómo se detecta, ya que debe haber un registro de servicio. En la misma línea, cuando se produce un error en un equipo, el servicio del Registro debe ser capaz de indicar dónde se está ejecutando el servicio.

El patrón del Registro de servicio es una parte clave de la detección de servicios. El registro es una base de datos que contiene las ubicaciones de red de las instancias de servicio. Un Registro del servicio debe estar muy disponible y actualizado. Los clientes podrían almacenar en caché las ubicaciones de red obtenidas del registro de servicio. Sin embargo, esa información finalmente deja de estar actualizada y los clientes ya no pueden detectar instancias de servicio. Por lo tanto, un registro de servicio consta de un clúster de servidores que usan un protocolo de replicación para mantener la coherencia.

En algunos entornos de implementación de microservicios (denominados clústeres, que se tratará en una sección posterior), la detección de servicios está integrada. Por ejemplo, un entorno de Azure Kubernetes Service (AKS) puede controlar el registro y la cancelación de registro de instancias de servicio. También ejecuta un proxy en cada host de clúster que desempeña el rol del enrutador de detección del lado servidor.

Recursos adicionales

- **Chris Richardson. Patrón: Registro de servicios**
  https://microservices.io/patterns/service-registry.html
- **Auth0. Registro de servicios**
  https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/
- **Gabriel Schenker. Descubrimiento de servicios**
  https://lostechies.com/gabrielschenker/2016/01/27/service-discovery/

Siguiente anterior

Creación de una interfaz de usuario compuesta basada en microservicios

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

La arquitectura de microservicios a menudo comienza con el control de datos y la lógica del lado servidor, pero, en muchos casos, la interfaz de usuario todavía se controla como un monolito. Sin embargo, un enfoque más avanzado, denominado micro front-end, es diseñar también la interfaz de usuario de la aplicación en función de los microservicios. Esto significa tener una interfaz de usuario compuesta producida por los microservicios, en lugar de tener microservicios en el servidor y simplemente una aplicación cliente monolítica que consume los microservicios. Con este enfoque, los microservicios que construyes pueden estar completos tanto con la lógica como con la representación visual.

En la figura 4-20 se muestra el enfoque más sencillo de consumir solo microservicios de una aplicación cliente monolítica. Por supuesto, podría tener un servicio ASP.NET MVC entre la producción de HTML y JavaScript. La ilustración es una simplificación que resalta que tiene una sola interfaz de usuario cliente (monolítica) que consume los microservicios, que solo se centran en la lógica y los datos y no en la forma de la interfaz de usuario (HTML y JavaScript).

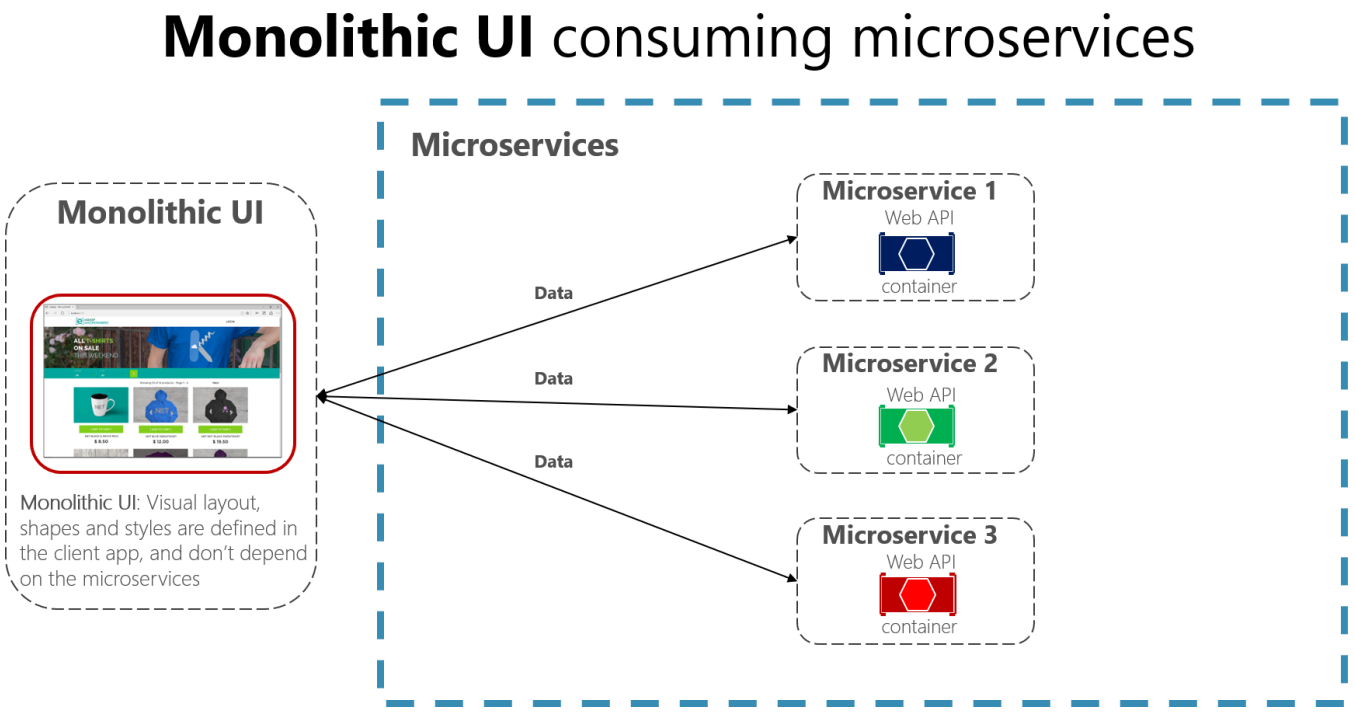


Figura 4-20. Una aplicación monolítica de interfaz de usuario que consume microservicios de fondo

Por el contrario, la interfaz de usuario compuesta se genera y se compone precisamente por los microservicios. Algunos de los microservicios impulsan la forma visual de áreas específicas de la interfaz de usuario. La diferencia clave es que tiene componentes de la interfaz de usuario del cliente (clases TypeScript, por ejemplo) basados en plantillas, y el modelo de vista que da forma a los datos para esas plantillas es proporcionado por cada microservicio.

En el momento de inicio de la aplicación cliente, cada uno de los componentes de la interfaz de usuario de cliente (clases TypeScript, por ejemplo) se registra con un microservicio de infraestructura capaz de proporcionar ViewModels para un escenario determinado. Si el microservicio cambia la forma, la interfaz de usuario también cambia.

En la figura 4-21 se muestra una versión de este enfoque de interfaz de usuario compuesta. Este enfoque se simplifica porque es posible que tenga otros microservicios que agregan partes pormenorizadas basadas en distintas técnicas. Depende de si va a crear un enfoque web tradicional (ASP.NET MVC) o un SPA (aplicación de página única).

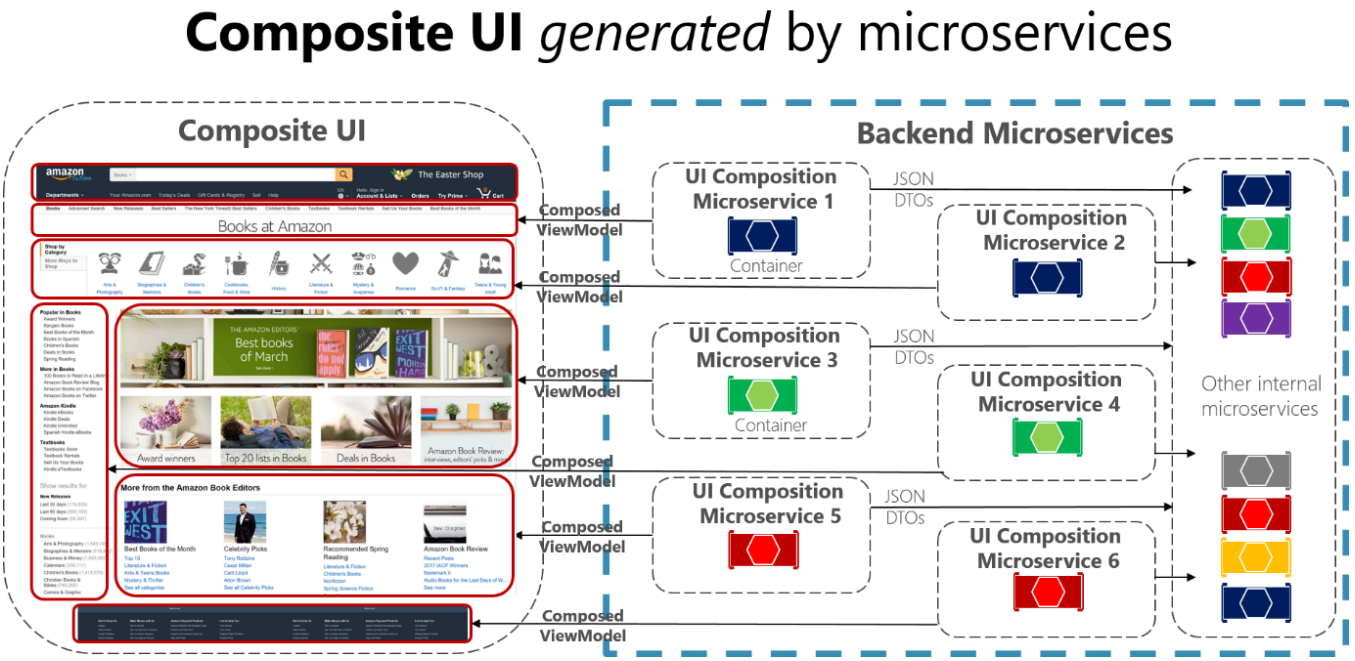


Figura 4-21. Ejemplo de una aplicación compuesta de UI modelada por microservicios de back-end

Cada uno de esos microservicios de composición de la interfaz de usuario sería similar a una puerta de enlace de API pequeña. Pero en este caso, cada uno es responsable de un área de interfaz de usuario pequeña.

Un enfoque de interfaz de usuario compuesto controlado por microservicios puede ser más difícil o menos, en función de las tecnologías de interfaz de usuario que use. Por ejemplo, no usará las mismas técnicas para crear una aplicación web tradicional que usará para desarrollar una SPA o para una aplicación móvil nativa, lo cual puede ser más desafiante con este enfoque.

La aplicación de ejemplo eShopOnContainers usa el enfoque de interfaz de usuario monolítica por varias razones. En primer lugar, es una introducción a los microservicios y contenedores. Una interfaz de usuario compuesta es más avanzada, pero también requiere una mayor complejidad al diseñar y desarrollar la interfaz de usuario. En segundo lugar, eShopOnContainers también proporciona una aplicación móvil nativa basada en Xamarin (no compatible a partir de mayo de 2024), lo que lo haría más complejo en el lado cliente de C#.

Sin embargo, le recomendamos que use las siguientes referencias para obtener más información sobre la interfaz de usuario compuesta basada en microservicios.

Recursos adicionales

- **Micro Frontends (blog de Martin Fowler)**https://martinfowler.com/articles/micro-frontends.html
- **Micro front-ends (sitio de Michael Geers)**https://micro-frontends.org/

- **UI compuesta mediante ASP.NET (Particular's Workshop)**https://github.com/Particular/Workshop/tree/master/demos/asp-net-core
- **Ruben Oostinga. Front-end monolítico en la arquitectura de microservicios** https://xebia.com/blog/the-monolithic-frontend-in-the-microservices-architecture/
- **Mauro Servienti. El secreto de una mejor composición de la interfaz de usuario** https://particular.net/blog/secret-of-better-ui-composition
- **Viktor Farcic. Incluir componentes Front-End Web en microservicios** https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/
- **Administración de front-end en la arquitectura de microservicios**https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

Siguiente anterior

Resistencia y alta disponibilidad en microservicios

29/05/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Enfrentarse a errores inesperados es uno de los problemas más difíciles de resolver, sobre todo en un sistema distribuido. Gran parte del código que escriben los desarrolladores implica controlar excepciones y esto también es donde se dedica más tiempo a las pruebas. El problema es más importante que escribir código para controlar los errores. ¿Qué ocurre cuando se produce un error en la máquina en la que se está ejecutando el microservicio? No solo necesita detectar este error de microservicio (un problema difícil por sí solo), sino que también necesita algo para reiniciar el microservicio.

Un microservicio debe ser resistente a errores y poder reiniciar a menudo en otra máquina para obtener disponibilidad. La resiliencia también depende del estado que se guardó en nombre del microservicio, desde donde el microservicio puede recuperar este estado y si el microservicio puede reiniciarse con éxito. Es decir, debe haber resistencia en la funcionalidad de proceso (el proceso puede reiniciarse en cualquier momento), así como resistencia en el estado o los datos (sin pérdida de datos y los datos permanecen coherentes).

Los problemas de resistencia se combinan durante otros escenarios, como cuando se producen errores durante una actualización de la aplicación. El microservicio, que trabaja con el sistema de implementación, debe determinar si puede continuar avanzando a la versión más reciente o revertir a una versión anterior para mantener un estado coherente. Preguntas como si hay

suficientes máquinas disponibles para seguir avanzando y cómo recuperar versiones anteriores del microservicio deben tenerse en cuenta. Este enfoque requiere que el microservicio emita información de estado para que la aplicación general y el orquestador puedan tomar estas decisiones.

Además, la resistencia está relacionada con el comportamiento de los sistemas basados en la nube. Como se mencionó, un sistema basado en la nube debe adoptar errores y debe intentar recuperarse automáticamente de ellos. Por ejemplo, en caso de errores de red o contenedor, las aplicaciones cliente o los servicios cliente deben tener una estrategia para reintentar el envío de mensajes o para reintentar solicitudes, ya que en muchos casos los errores en la nube son parciales. En la sección Implementación de aplicaciones resistentes de esta guía se explica cómo controlar errores parciales. Describe técnicas como reintentos con retroceso exponencial o el patrón Circuit Breaker en .NET mediante bibliotecas como Polly, que ofrece una gran variedad de directivas para controlar este tema.

Gestión de la salud y diagnóstico en microservicios

Puede parecer obvio y a menudo se pasa por alto, pero un microservicio debe notificar su estado y diagnóstico. De lo contrario, hay poca información desde la perspectiva de las operaciones. La correlación de eventos de diagnóstico en un conjunto de servicios independientes y el manejo de las desviaciones del reloj de las máquinas para comprender el orden de los eventos es un desafío. De la misma manera que interactúa con un microservicio a través de protocolos y formatos de datos acordados, es necesario normalizar cómo registrar eventos de salud y diagnóstico que, en última instancia, terminan en un almacén de eventos para consulta y visualización. En un enfoque de microservicios, es clave que los distintos equipos acepten un único formato de registro. Debe haber un enfoque coherente para ver los eventos de diagnóstico en la aplicación.

Exámenes de salud

El estado es diferente de los diagnósticos. El estado trata de cuando el microservicio informa sobre su estado actual para que se tomen las medidas oportunas. Un buen ejemplo es colaborar con los mecanismos de actualización e implementación para mantener la disponibilidad. Aunque un servicio podría estar en mal estado debido a un bloqueo de proceso o reinicio de la máquina, es posible que el servicio siga funcionando. Lo último que necesita es empeorarlo realizando una actualización. El mejor enfoque es realizar una investigación primero o permitir que el microservicio se recupere. Los eventos de mantenimiento de un microservicio permiten tomar decisiones fundamentadas y, de hecho, ayudan a crear servicios de recuperación automática.

En la sección Implementación de comprobaciones de estado en ASP.NET Servicios principales de esta guía, se explica cómo usar una nueva biblioteca de ASP.NET HealthChecks en los microservicios para que puedan notificar su estado a un servicio de supervisión para realizar las acciones adecuadas.

También tiene la opción de usar una excelente biblioteca de código abierto denominada AspNetCore.Diagnostics.HealthChecks, disponible en GitHub y como paquete NuGet. Esta biblioteca también realiza comprobaciones de estado, con un giro inesperado, gestiona dos tipos de comprobaciones:

- **Vivacidad**: Comprueba si el microservicio está vivo, es decir, si es capaz de aceptar solicitudes y responder.
- **Preparación**: comprueba si las dependencias del microservicio (base de datos, servicios de cola, etc.) están listas, por lo que el microservicio puede hacer lo que se supone que debe hacer.

Uso de flujos de eventos de diagnóstico y registros

Los registros proporcionan información sobre cómo se ejecuta una aplicación o servicio, incluidas las excepciones, las advertencias y los mensajes informativos simples. Normalmente, cada registro tiene un formato de texto con una línea por evento, aunque a menudo las excepciones también muestran el seguimiento de la pila en varias líneas.

En aplicaciones monolíticas basadas en servidor, puede escribir registros en un archivo en disco (un archivo de registro) y, a continuación, analizarlo con cualquier herramienta. Dado que la ejecución de la aplicación se limita a un servidor fijo o una máquina virtual, normalmente no es demasiado complejo analizar el flujo de eventos. Sin embargo, en una aplicación distribuida en la que se ejecutan varios servicios en muchos nodos de un clúster de orquestador, la posibilidad de correlacionar eventos distribuidos es un desafío.

Una aplicación basada en microservicios no debe intentar almacenar el flujo de salida de eventos o archivos de registro por sí mismo, ni siquiera intentar administrar el enrutamiento de los eventos a un lugar central. Debe ser transparente, lo que significa que cada proceso solo debe escribir su flujo de eventos en una salida estándar que será recogida por la infraestructura del entorno de ejecución. Un ejemplo de estos enrutadores de flujo de eventos es Microsoft.Diagnostic.EventFlow, que recopila flujos de eventos de varios orígenes y los publica en sistemas de salida. Esto puede incluir una salida estándar sencilla para un entorno de desarrollo o sistemas en la nube como Azure Monitor y Azure Diagnostics. También hay buenas plataformas y herramientas de análisis de registros de terceros que pueden buscar, alertar, informar y supervisar registros, incluso en tiempo real, como Splunk o Middleware.

Orquestadores que administran la información de salud y diagnóstico

Al crear una aplicación basada en microservicios, debe tratar la complejidad. Por supuesto, un único microservicio es sencillo de tratar, pero docenas o cientos de tipos y miles de instancias de microservicios es un problema complejo. No solo se trata de crear la arquitectura de microservicios; también necesita alta disponibilidad, direccionabilidad, resistencia, mantenimiento y diagnósticos si tiene previsto tener un sistema estable y cohesivo.

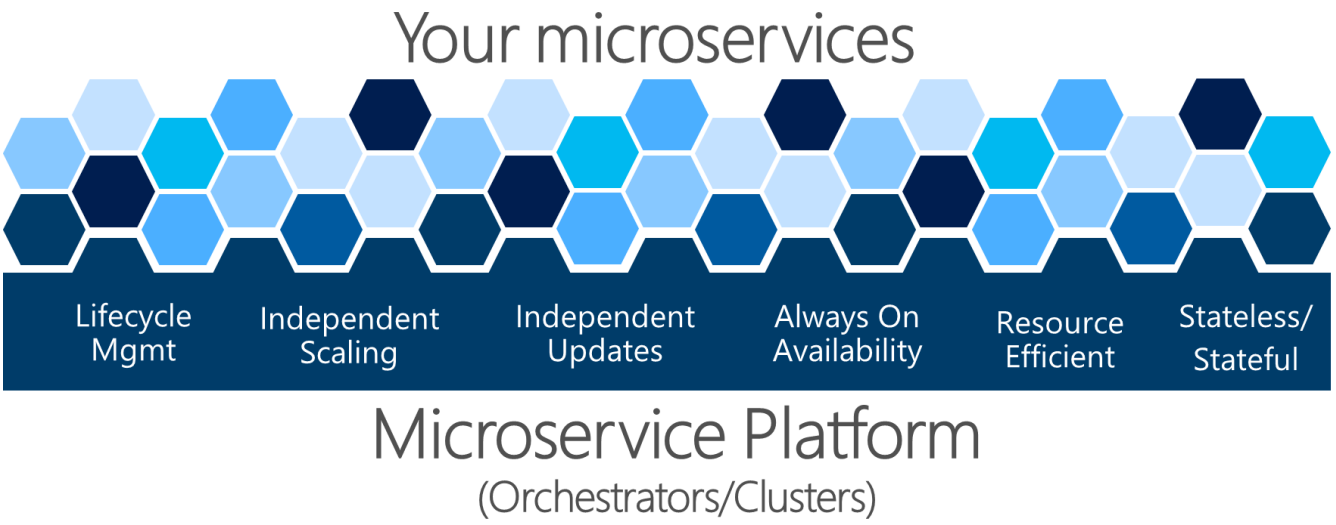


Figura 4-22. Una plataforma de microservicios es fundamental para la administración del estado de una aplicación

Los problemas complejos que se muestran en la figura 4-22 son difíciles de resolver por usted mismo. Los equipos de desarrollo deben centrarse en resolver problemas empresariales y crear aplicaciones personalizadas con enfoques basados en microservicios. No deben centrarse en resolver problemas complejos de infraestructura; Si lo hicieran, el costo de cualquier aplicación basada en microservicios sería enorme. Por lo tanto, hay plataformas orientadas a microservicios, denominadas orquestadores o clústeres de microservicios, que intentan resolver los problemas difíciles de crear y ejecutar un servicio y usar recursos de infraestructura de forma eficaz. Este enfoque reduce las complejidades de la creación de aplicaciones que usan un enfoque de microservicios.

Los diferentes orquestadores pueden parecer similares, pero los diagnósticos y las comprobaciones de estado que ofrece cada uno de ellos difieren en las características y el estado de madurez, a veces en función de la plataforma del sistema operativo, como se explica en la sección siguiente.

Recursos adicionales

- **La aplicación Twelve-Factor. XI. Registros: tratar los registros como flujos de eventos**
  https://12factor.net/logs
- Repositorio de GitHub de la biblioteca de diagnóstico EventFlow de Microsoft.
  https://github.com/Azure/diagnostics-eventflow
- **¿Qué es Azure Diagnostics?**
  https://learn.microsoft.com/azure/azure-diagnostics
- **Conexión de equipos Windows al servicio Azure Monitor**
  https://learn.microsoft.com/azure/azure-monitor/platform/agent-windows
- **Registrar lo que quiere decir: Uso del bloque de registro semántico**
  https://learn.microsoft.com/previous-versions/msp-n-p/dn440729(v=pandp.60)
- **Splunk** Sitio oficial.
  https://www.splunk.com/
- **Middleware** Sitio oficial.
  https://middleware.io/
- **Clase EventSource** API para el seguimiento de eventos para Windows (ETW)
  https://learn.microsoft.com/dotnet/api/system.diagnostics.tracing.eventsource

Siguiente anterior

Orquestar microservicios y aplicaciones de varios contenedores para una alta escalabilidad y disponibilidad

Artículo • 10/05/2023

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en **Documentación de .NET** o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

La utilización de orquestadores para aplicaciones listas para producción es fundamental si la aplicación se basa en microservicios o simplemente está dividida entre varios contenedores. Como se mencionó anteriormente, en un enfoque basado en microservicios, cada microservicio posee su modelo y sus datos para que sea autónomo desde un punto de vista del desarrollo y la implementación. Pero incluso si tiene una aplicación más tradicional que se compone de varios servicios (por ejemplo, SOA), también tendrá varios contenedores o servicios que conforman una sola aplicación de negocio que deban implementarse como un sistema distribuido. Estos tipos de sistemas son difíciles de administrar y escalar horizontalmente; por lo tanto, un orquestador es indispensable si se quiere tener una aplicación de varios contenedores, escalable y lista para la producción.

La figura 4-23 ilustra la implementación en un clúster de una aplicación formada por varios microservicios (contenedores).

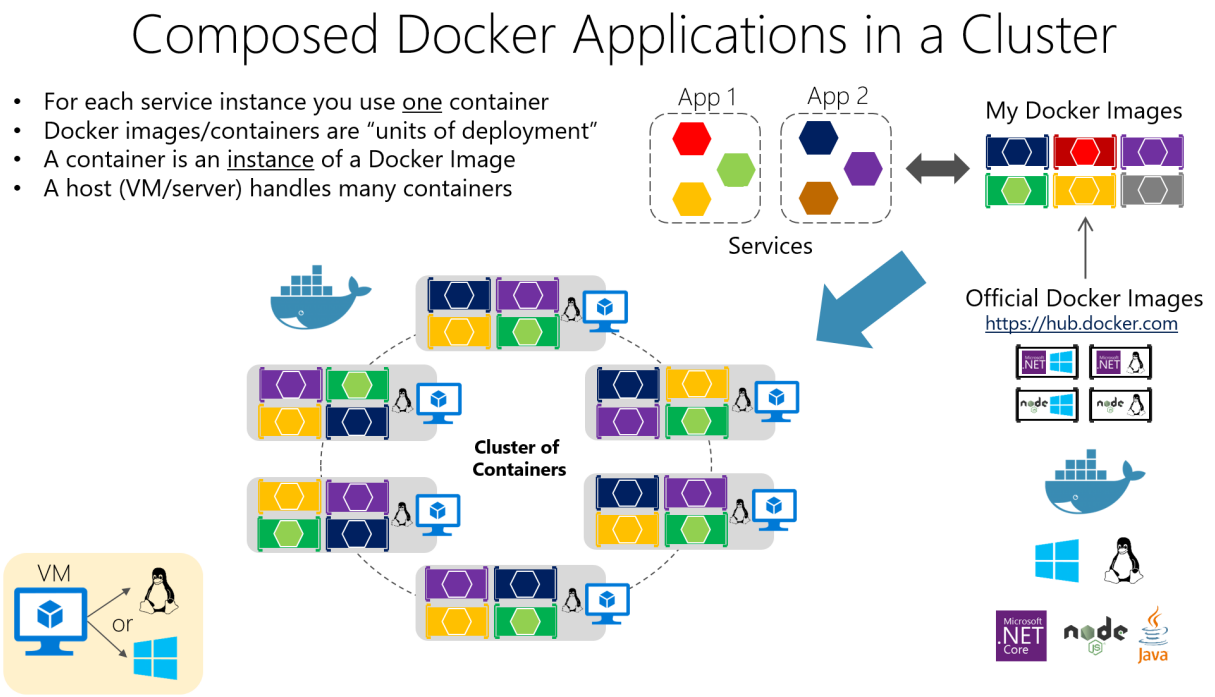


Figura 4-23. Un clúster de contenedores

use un contenedor para cada instancia de servicio. Los contenedores de Docker son "unidades de implementación" y un contenedor es una instancia de Docker. Un host controla muchos contenedores. Parece un enfoque lógico. Pero, ¿cómo se está administrando el equilibrio de carga de control, el enrutamiento y la orquestación de estas aplicaciones compuestas?

El motor de Docker estándar en hosts de Docker único satisface las necesidades de administración de instancias de imagen únicas en un host, pero se queda corto a la hora de administrar varios contenedores implementados en varios hosts para aplicaciones distribuidas más complejas. En la mayoría de los casos, se necesita una plataforma de administración que inicie automáticamente los contenedores, los escale horizontalmente con varias instancias por imagen, y los suspenda o los cierre cuando sea necesario, así como que, a ser posible, controle su acceso a recursos como la red y el almacenamiento de datos.

Para ir más allá de la administración de contenedores individuales o aplicaciones compuestas simples y pasar a aplicaciones empresariales más grandes con microservicios, debe cambiar a orquestación y plataformas de agrupación en clústeres.

Desde el punto de vista de la arquitectura y el desarrollo, si está compilando grandes aplicaciones empresariales basadas en microservicios, es importante familiarizarse con las siguientes plataformas y productos que admiten escenarios avanzados:

Clústeres y orquestadores. Cuando se necesita escalar horizontalmente las aplicaciones a varios hosts de Docker, como con una aplicación grande basada en microservicios, es

fundamental poder administrar todos los hosts como un solo clúster mediante la abstracción de la complejidad de la plataforma subyacente. Eso es lo que proporcionan los clústeres de contenedor y los orquestadores. Kubernetes es un ejemplo de orquestador, y está disponible en Azure a través de Azure Kubernetes Service.

Programadores.*Programar* hace referencia a la capacidad de un administrador para iniciar contenedores de un clúster, y de proporcionar también una interfaz de usuario. Un programador de clúster tiene varias responsabilidades: usar eficazmente los recursos del clúster, establecer las restricciones definidas por el usuario, equilibrar eficazmente la carga de los contenedores entre los distintos nodos o hosts, ser resistente a los errores y proporcionar un alto grado de disponibilidad.

Los conceptos de un clúster y un programador están estrechamente relacionados, por lo que los productos proporcionados por diferentes proveedores suelen ofrecer ambos conjuntos de funciones. En la lista siguiente se muestran las plataformas y las opciones de software más importantes disponibles para clústeres y programadores. Estos orquestadores generalmente se ofrecen en nubes públicas como Azure.

Plataformas de software para agrupación en clústeres de contenedores, orquestación y programación

Plataforma	Descripción
Kubernetes	*Kubernetes* es un producto de código abierto cuya funcionalidad abarca desde la infraestructura de clúster y la programación de contenedores a las capacidades de orquestación. Permite automatizar la implementación, la escala y las operaciones de contenedores de aplicaciones en varios clústeres de hosts. *Kubernetes* proporciona una infraestructura centrada en el contenedor que agrupa los contenedores de la aplicación en unidades lógicas para facilitar la administración y detección. *Kubernetes* está más desarrollado en Linux que en Windows.
Azure Kubernetes Service (AKS)	AKS es un servicio de orquestación de contenedores de Kubernetes administrado en Azure que simplifica la administración, la implementación y las operaciones del clúster de Kubernetes.

Plataforma	Descripción
Azure Container Apps	Azure Container Apps es un servicio de contenedor sin servidor administrado para la creación y la implementación de aplicaciones modernas a gran escala.

Uso de orquestadores basados en contenedor en Microsoft Azure

Existen varios proveedores de nube que ofrecen compatibilidad con contenedores de Docker más compatibilidad con la orquestación y los clústeres de Docker, como Microsoft Azure, Amazon EC2 Container Service y Google Container Engine. Microsoft Azure proporciona compatibilidad con orquestador y clúster de Docker a través de Azure Kubernetes Service (AKS).

Uso de Azure Kubernetes Service

Un clúster de Kubernetes agrupa varios hosts de Docker y los expone como un único host virtual de Docker, lo que permite implementar varios contenedores en el clúster y escalar horizontalmente con cualquier número de instancias de contenedor. El clúster controlará toda la mecánica de administración compleja, como la escalabilidad, el estado, etc.

AKS proporciona una manera de simplificar la creación, la configuración y la administración de un clúster de máquinas virtuales en Azure que están preconfiguradas para ejecutar aplicaciones en contenedores. Al utilizar una configuración optimizada de herramientas de orquestación y programación de código abierto populares, AKS le permite usar sus habilidades existentes o aprovechar un gran corpus creciente de conocimientos de la comunidad para implementar y administrar aplicaciones basadas en contenedor en Microsoft Azure.

Azure Kubernetes Service optimiza la configuración de tecnologías y herramientas populares de código abierto de agrupación en clústeres de Docker específicamente para Azure. Se trata de una solución abierta que ofrece la portabilidad de los contenedores y la configuración de la aplicación. Seleccione el tamaño, el número de hosts y las herramientas de orquestador, y AKS se encarga de todo lo demás.

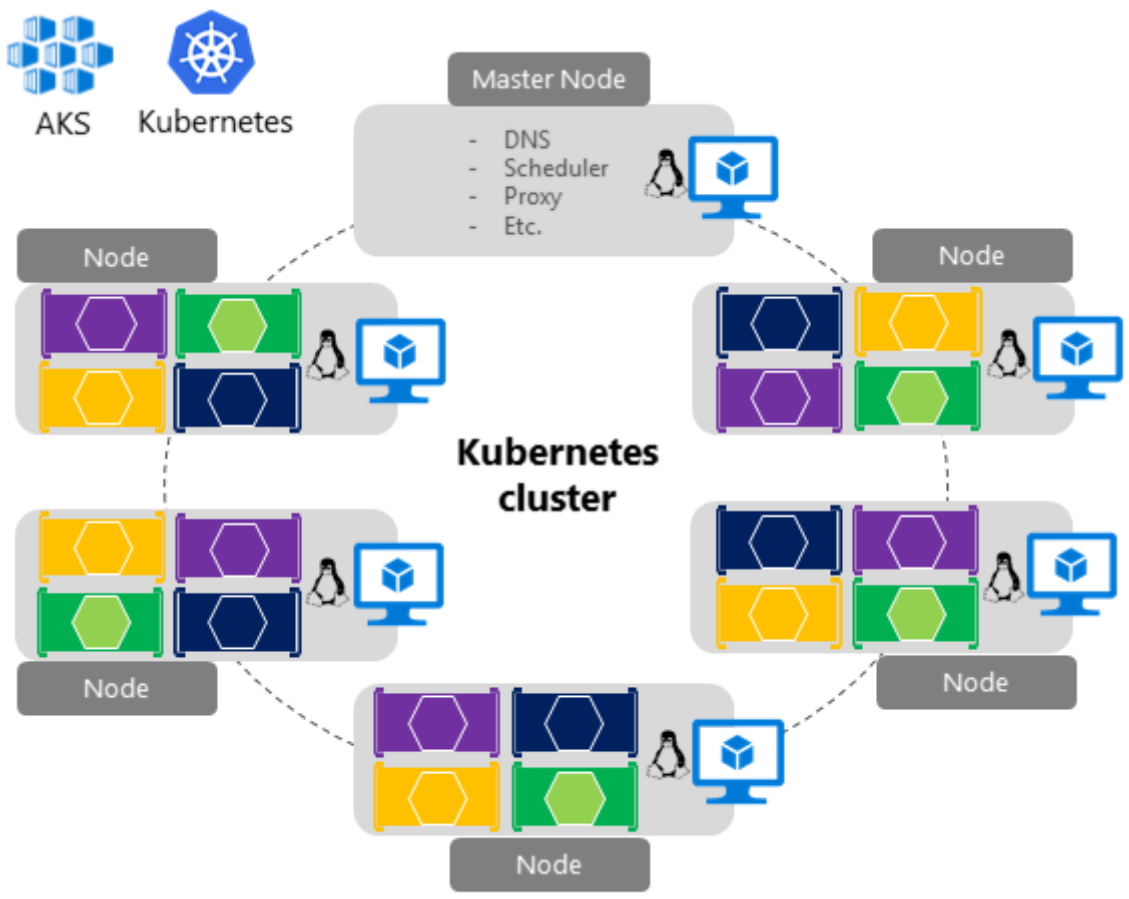


Figura 4-24. Topología y estructura simplificada del clúster de Kubernetes

En la figura 4-24, puede ver la estructura de un clúster de Kubernetes, donde un nodo maestro (VM) controla la mayor parte de la coordinación del clúster. Se pueden implementar contenedores en el resto de los nodos, que se administran como un único grupo desde un punto de vista de la aplicación. También puede realizar un escalado a miles o incluso a decenas de miles de contenedores.

Entorno de desarrollo para Kubernetes

En el entorno de desarrollo, Docker anunció en julio de 2018 que Kubernetes también podía ejecutarse en un único equipo de desarrollo (Windows 10 o macOS) mediante la instalación de Docker Desktop. Puede implementar posteriormente en la nube (AKS) para obtener más pruebas de integración, como se muestra en la figura 4-25.

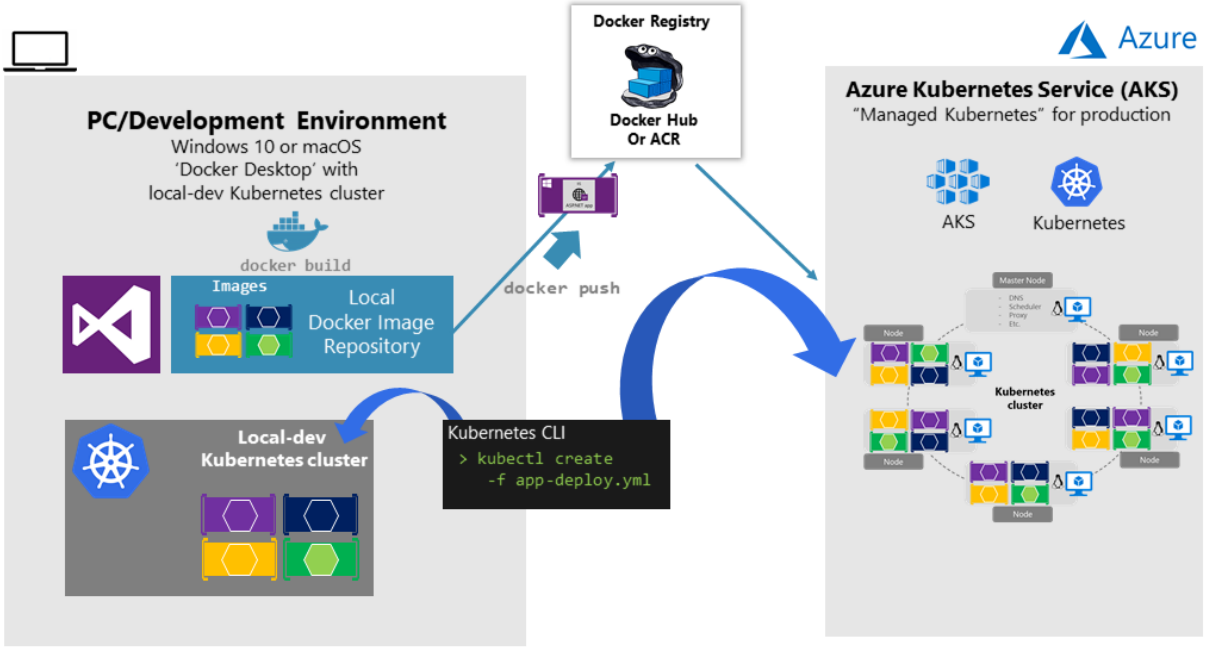


Figura 4-25. Ejecución de Kubernetes en el equipo de desarrollo y la nube

Introducción a Azure Kubernetes Service (AKS)

Para empezar a usar AKS, implemente un clúster de AKS desde Azure Portal o mediante la CLI. Para más información sobre la implementación de un clúster de Kubernetes en Azure, consulte Inicio rápido: Implementación de un clúster de Azure Kubernetes Service (AKS).

No hay cuotas para el software instalado de forma predeterminada como parte de AKS. Todas las opciones predeterminadas se implementan con el software de código abierto. AKS está disponible en varias máquinas virtuales en Azure. Se cobra únicamente por las instancias de proceso que se elijan, así como por los otros recursos subyacentes de la infraestructura que se utilicen, por ejemplo, la red y el almacenamiento. No hay ningún cargo incremental para AKS.

La opción de implementación de producción predeterminada para Kubernetes es usar gráficos Helm, los cuales veremos en la sección siguiente.

Implementación con gráficos de Helm en clústeres de Kubernetes

Al implementar una aplicación en un clúster de Kubernetes, puede usar la herramienta de CLI kubectl.exe original mediante archivos de implementación basados en el formato nativo (archivos .yaml), como ya se mencionó en la sección anterior. Pero, para

aplicaciones de Kubernetes más complejas, como al implementar aplicaciones complejas basadas en microservicios, se recomienda usar Helm.

Los gráficos de Helm le ayudan a definir, establecer la versión, instalar, compartir, actualizar o revertir incluso la aplicación más compleja de Kubernetes.

Adicionalmente, el uso de Helm se recomienda porque otros entornos de Kubernetes en Azure, como Azure Dev Spaces, también se basan en los gráficos de Helm.

La Cloud Native Computing Foundation (CNCF) mantiene Helm en colaboración con Microsoft, Google, Bitnami y la comunidad de colaboradores de Helm.

Para obtener más información sobre la implementación de gráficos de Helm y Kubernetes, vea la entrada de blog Uso de gráficos de Helm para implementar eShopOnContainers en AKS.

Recursos adicionales

- **Guía de inicio rápido: Implementación de un clúster de Azure Kubernetes Service (AKS)**
  https://learn.microsoft.com/azure/aks/kubernetes-walkthrough-portal
- **Azure Dev Spaces**
  https://learn.microsoft.com/azure/dev-spaces/azure-dev-spaces
- **Kubernetes** El sitio oficial.
  https://kubernetes.io/

Anterior Siguiente

Proceso de desarrollo para aplicaciones basadas en Docker

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Desarrolle aplicaciones .NET en contenedores como prefiera, ya sea con un entorno de desarrollo integrado (IDE) centrado en Visual Studio y las herramientas de Visual Studio para Docker, o centrado en CLI/editor con Docker CLI y Visual Studio Code.

Entorno de desarrollo para aplicaciones de Docker

Opciones de herramientas de desarrollo: IDE o editor

Ya sea que prefiera un IDE completo y eficaz o un editor ligero y ágil, Microsoft tiene herramientas que puede usar para desarrollar aplicaciones de Docker.

Visual Studio (para Windows). El desarrollo de aplicaciones .NET 8 basado en Docker con Visual Studio requiere Visual Studio 2022, versión 17.0 o posterior. Visual Studio 2022 incluye herramientas para Docker ya integradas. Las herramientas de Docker permiten desarrollar, ejecutar y validar las aplicaciones directamente en el entorno de Docker de destino. Puede presionar `F5` para ejecutar y depurar la aplicación (contenedor único o varios contenedores) directamente en un host de Docker, o presione `CTRL` + `F5` para editar y actualizar la aplicación

sin tener que recompilar el contenedor. Este IDE es la opción de desarrollo más eficaz para aplicaciones basadas en Docker.

Visual Studio Code y la CLI de Docker. Si prefiere un editor ligero y multiplataforma que admita cualquier lenguaje de desarrollo, puede usar Visual Studio Code y la CLI de Docker. Este IDE es un enfoque de desarrollo multiplataforma para macOS, Linux y Windows. Además, Visual Studio Code admite extensiones para Docker, como IntelliSense para Dockerfiles y tareas de acceso directo para ejecutar comandos de Docker desde el editor.

Al instalar Docker Desktop, puede usar una sola CLI de Docker para compilar aplicaciones para Windows y Linux.

Recursos adicionales

- **Visual Studio**. Sitio oficial.
  https://visualstudio.microsoft.com/vs/
- **Código de Visual Studio**. Sitio oficial.
  https://code.visualstudio.com/download
- **Docker Desktop para Windows**
  https://hub.docker.com/editions/community/docker-ce-desktop-windows
- **Docker Desktop para Mac**
  https://hub.docker.com/editions/community/docker-ce-desktop-mac

Lenguajes y marcos de .NET para contenedores de Docker

Como se mencionó en las secciones anteriores de esta guía, puede usar .NET Framework, .NET 8 o el proyecto Mono de código abierto al desarrollar aplicaciones .NET en contenedor de Docker. Puede desarrollar en C#, F#o Visual Basic cuando tenga como destino contenedores de Linux o Windows, en función de qué marco de .NET esté en uso. Para obtener más información sobre los lenguajes de .NET, consulte la entrada de blog Estrategia de lenguaje .NET.

Siguiente anterior

Flujo de trabajo de desarrollo para aplicaciones de Docker

Artículo • 10/09/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El ciclo de vida de desarrollo de una aplicación se inicia en el equipo de cada desarrollador, donde se programa la aplicación con el lenguaje preferido y se prueba en el entorno local. Con este flujo de trabajo, no importa el lenguaje, el marco ni la plataforma que se elija, ya que siempre se desarrollan y se prueban contenedores de Docker en local.

Cada contenedor (una instancia de una imagen de Docker) incluye los siguientes componentes:

- Una selección de sistema operativo, por ejemplo, una distribución de Linux, Windows Nano Server o Windows Server Core.
- Archivos agregados durante el desarrollo, por ejemplo, archivos binarios de código fuente y aplicación.
- Información de configuración, como configuración de entorno y dependencias.

Flujo de trabajo para desarrollar aplicaciones basadas en contenedor de Docker

En esta sección se explica el flujo de trabajo de desarrollo de *bucle interno* para aplicaciones basadas en contenedor de Docker. Flujo de trabajo de bucle interno significa que no se tiene en cuenta el flujo de trabajo general de DevOps, que puede incluir hasta implementación en producción, y solo se centra en el trabajo de desarrollo realizado en el equipo del desarrollador. Los pasos iniciales para configurar el entorno no se incluyen, ya que se realizan solo una vez.

Una aplicación se compone de sus propios servicios, además de bibliotecas adicionales (dependencias). Estos son los pasos básicos que normalmente se realizan al compilar una aplicación de Docker, como se muestra en la figura 5-1.

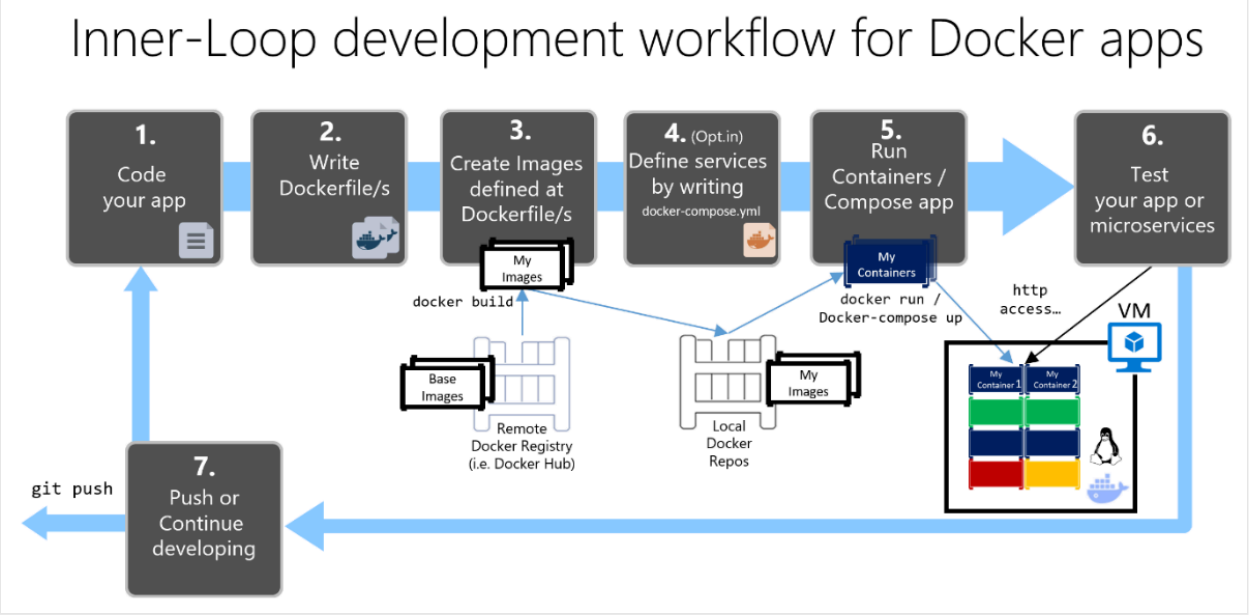


Figura 5-1. Flujo de trabajo paso a paso para el desarrollo de aplicaciones en contenedor de Docker

En esta sección se detalla el proceso completo y se explica cada paso importante centrándose en un entorno de Visual Studio.

Cuando se usa un enfoque de desarrollo de editor/CLI (por ejemplo, Visual Studio Code más la CLI de Docker en macOS o Windows), es necesario conocer cada paso, generalmente más detalladamente que si se usa Visual Studio. Para obtener más información sobre cómo trabajar en un entorno de CLI, vea el libro electrónico Containerized Docker Application lifecycle with Microsoft Platforms and Tools (Ciclo de vida de las aplicaciones en contenedor de Docker con plataformas y herramientas de Microsoft).

Al usar Visual Studio 2022, muchos de esos pasos se controlan de forma automática, lo que mejora considerablemente la productividad. Esto es así especialmente con Visual Studio 2022 y cuando el destino son aplicaciones de varios contenedores. Por ejemplo, con un solo clic, Visual Studio agrega `Dockerfile` y el archivo `docker-`

`compose.yml` a los proyectos con la configuración de la aplicación. Al ejecutar la aplicación en Visual Studio, compila la imagen de Docker y ejecuta la aplicación de varios contenedores directamente en Docker; incluso permite depurar varios contenedores al mismo tiempo. Estas características aumentan la velocidad de desarrollo.

Pero que Visual Studio realice esos pasos automáticamente no significa que no sea necesario saber lo que ocurre en segundo plano con Docker. Por lo tanto, la guía siguiente detalla cada paso.

Paso 1. Empezar a programar y crear la aplicación inicial o la base de referencia del servicio

El desarrollo de una aplicación de Docker es similar al desarrollo de una aplicación sin Docker. La diferencia es que al desarrollar para Docker, la aplicación o los servicios que se están implementando y probando se ejecutan en contenedores de Docker en el entorno local (una instalación de máquina virtual de Linux realizada por Docker o directamente Windows si se usan contenedores de Windows).

Configurar el entorno local con Visual Studio

Para empezar, asegúrese de que tiene instalado Docker Desktop para Windows, como se explica en estas instrucciones:

Introducción a Docker Desktop para Windows

Además, se necesita Visual Studio 2022 versión 17.0, con la carga de trabajo **Desarrollo web y .ASP NET** instalada, tal como se muestra en la figura 5-2.

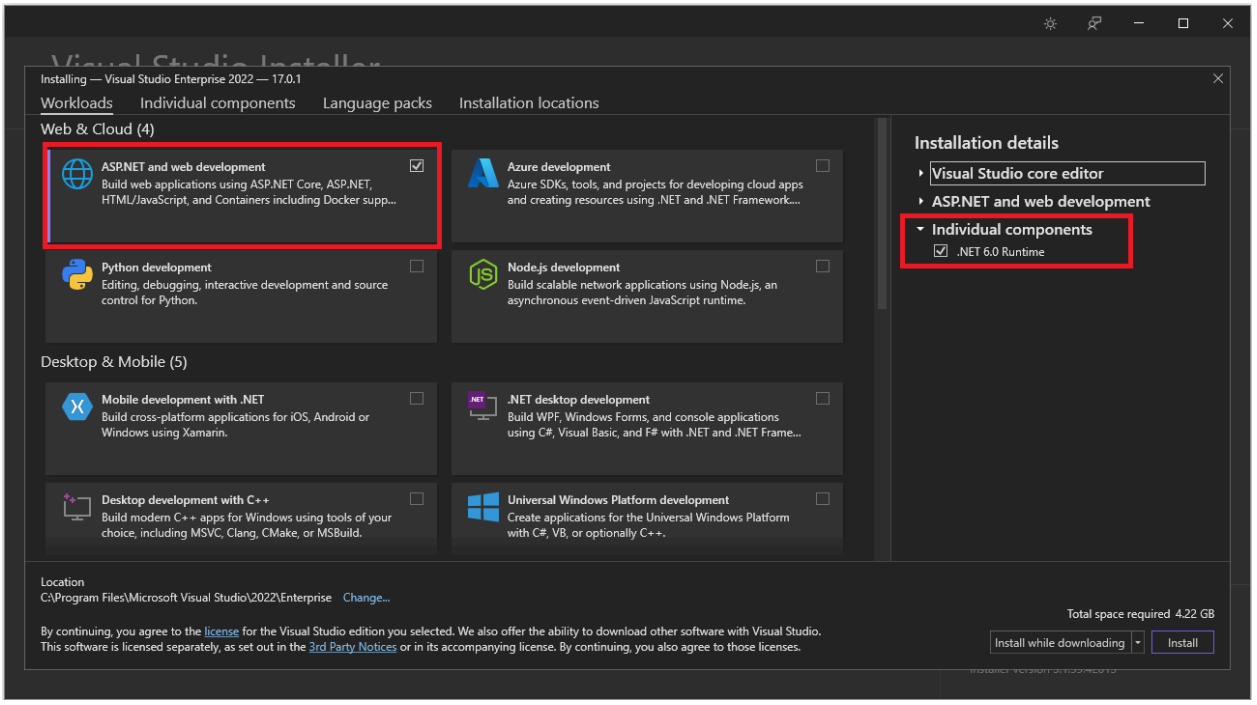


Figura 5-2. Selección de la carga de trabajo **Desarrollo web y ASP.NET** durante la instalación de Visual Studio 2022

Puede empezar a programar la aplicación en .NET sin formato (normalmente en .NET Core o versiones posteriores si va a usar contenedores) incluso antes de habilitar Docker en la aplicación, e implementar y probar en Docker. Pero se recomienda empezar a trabajar en Docker tan pronto como sea posible, ya que es el entorno real y se pueden detectar los problemas a la mayor brevedad. Se recomienda encarecidamente porque Visual Studio facilita tanto el trabajo con Docker que casi parece transparente: el mejor ejemplo al depurar aplicaciones de varios contenedores desde Visual Studio.

Recursos adicionales

- **Introducción a Docker Desktop para Windows**
  https://docs.docker.com/docker-for-windows/

- **Visual Studio 2022**
  https://visualstudio.microsoft.com/downloads/

Paso 2. Crear un Dockerfile relacionado con una imagen base existente de .NET

Necesita un Dockerfile para cada imagen personalizada que quiera compilar; también necesita un Dockerfile para cada contenedor que se vaya a implementar, tanto si se implementa automáticamente desde Visual Studio como manualmente mediante la CLI de Docker (comandos docker run y docker-compose). Si la aplicación contiene un único servicio personalizado, necesita un solo Dockerfile. Si la aplicación contiene varios servicios (como en una arquitectura de microservicios), necesita un Dockerfile para cada servicio.

El Dockerfile se coloca en la carpeta raíz de la aplicación o el servicio. Contiene los comandos que indican a Docker cómo configurar y ejecutar la aplicación o el servicio en un contenedor. Puede crear un Dockerfile de forma manual en el código y agregarlo al proyecto junto con las dependencias de .NET.

Con Visual Studio y sus herramientas para Docker, esta tarea solo exige unos clics. Al crear un proyecto en Visual Studio 2022, hay una opción denominada **Habilitar compatibilidad con Docker**, como se muestra en la figura 5-3.

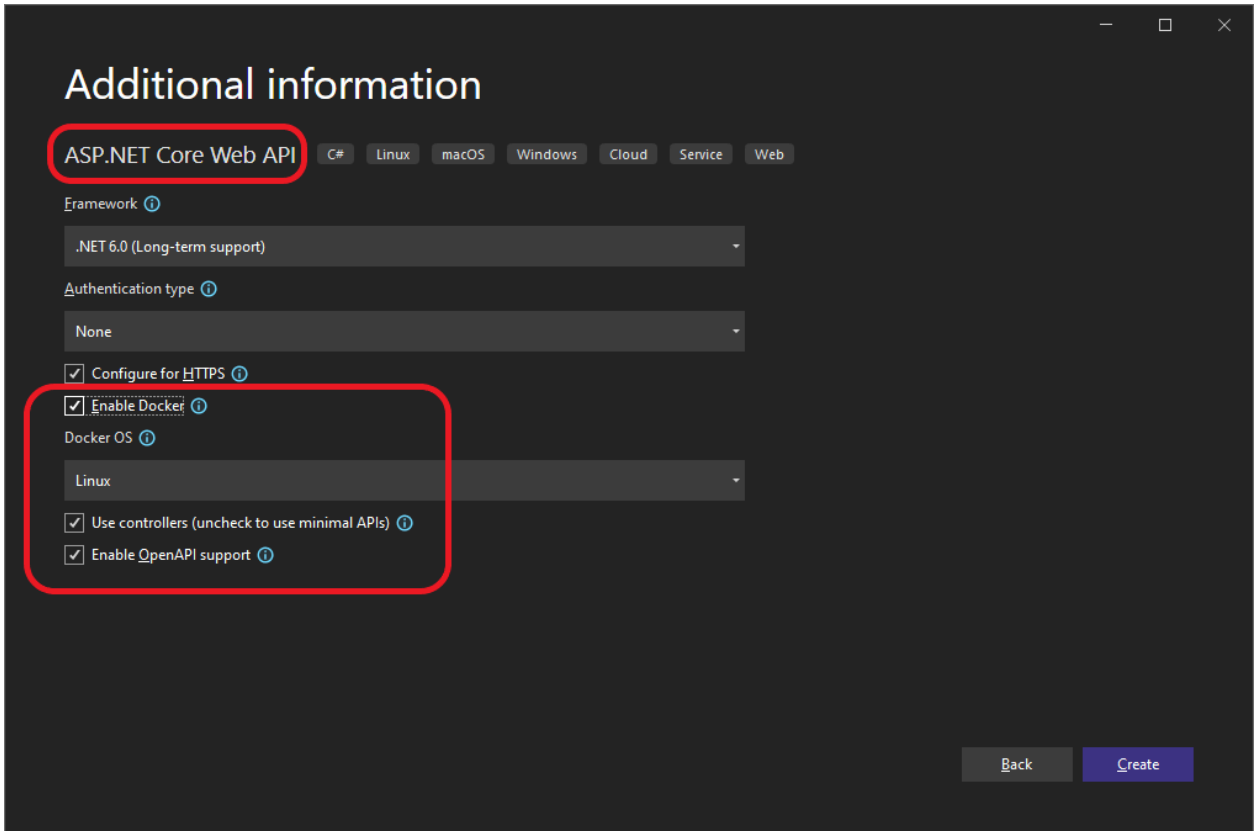


Figura 5-3. Habilitación de la compatibilidad con Docker al crear un proyecto de ASP.NET Core en Visual Studio 2022

También puede habilitar la compatibilidad con Docker en un proyecto de aplicación web de ASP.NET Core existente haciendo clic con el botón derecho en el proyecto, en el **Explorador de soluciones**, y seleccionando **Agregar**>**Compatibilidad con Docker...** , como se muestra en la figura 5-4.

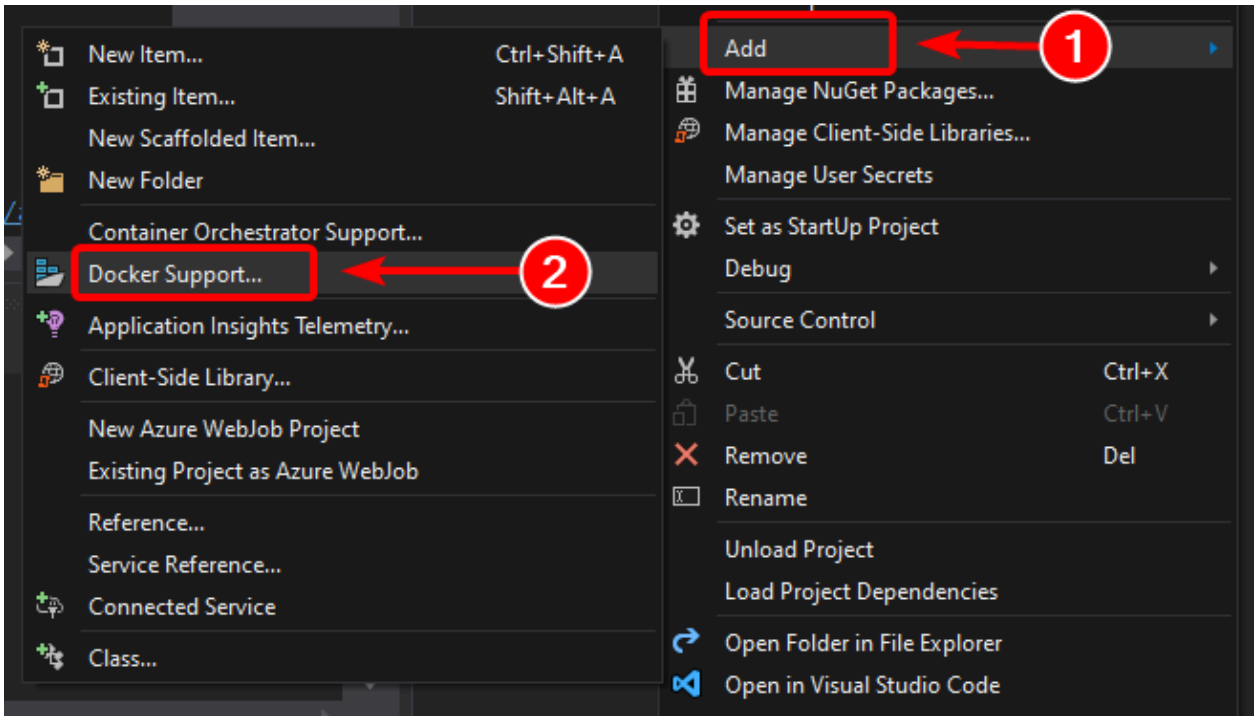


Figura 5-4. Habilitación de la compatibilidad con Docker en un proyecto existente de Visual Studio 2022

Esta acción agrega un *Dockerfile* al proyecto con la configuración necesaria y solo está disponible en los proyectos de ASP.NET Core.

De forma similar, Visual Studio también puede agregar un archivo `docker-compose.yml` para toda la solución con la opción **Agregar** > **Compatibilidad con el orquestador de contenedores...**. En el paso 4 se examina esta opción más detalladamente.

Uso de una imagen de Docker de .NET oficial existente

Normalmente se compila una imagen personalizada para el contenedor además de una imagen base que se obtiene de un repositorio oficial como el registro Docker Hub. Eso es precisamente lo que sucede en segundo plano cuando se habilita la compatibilidad con Docker en Visual Studio. El Dockerfile usa una imagen `dotnet/core/aspnet` existente.

Anteriormente se ha explicado qué imágenes y repositorios de Docker se pueden usar según el marco de trabajo y el sistema operativo elegidos. Por ejemplo, si quiere usar ASP.NET Core (Linux o Windows), la imagen que se debe usar es `mcr.microsoft.com/dotnet/aspnet:8.0`. Por lo tanto, debe especificar qué imagen base

de Docker va a usar para el contenedor. Se hace mediante la incorporación de `FROM mcr.microsoft.com/dotnet/aspnet:8.0` al Dockerfile. Visual Studio lo hace de forma automática, pero si va a actualizar la versión, actualice este valor.

El uso de un repositorio de imágenes de .NET oficial de Docker Hub con un número de versión garantiza que haya las mismas características de lenguaje disponibles en todos los equipos (incluido el desarrollo, las pruebas y la producción).

En el ejemplo siguiente se muestra un Dockerfile de ejemplo para un contenedor de ASP.NET Core.

Dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", " MySingleContainerWebApp.dll "]
```

En este caso, la imagen se basa en la versión 8.0 de la imagen de Docker de ASP.NET Core oficial (multiarquitectura para Linux y Windows). Es el valor `FROM mcr.microsoft.com/dotnet/aspnet:8.0`. (Para obtener más información sobre esta imagen base, vea la página de la imagen de Docker de ASP.NET Core). En el Dockerfile, también debe indicar a Docker que escuche en el puerto TCP que se vaya a usar en tiempo de ejecución (en este caso, el puerto 80, como se ha configurado con el valor EXPOSE).

Puede especificar otros valores de configuración en el Dockerfile, según el lenguaje y el marco que use. Por ejemplo, la línea ENTRYPOINT con `["dotnet", "MySingleContainerWebApp.dll"]` indica a Docker que ejecute una aplicación .NET. Si usa el SDK y la CLI de .NET (dotnet CLI) para compilar y ejecutar la aplicación .NET, este valor sería diferente. La conclusión es que la línea ENTRYPOINT y otros valores pueden variar según el lenguaje y la plataforma que se elijan para la aplicación.

Recursos adicionales

- **Compilación de imágenes de Docker para aplicaciones de ASP.NET Core** https://learn.microsoft.com/dotnet/core/docker/building-net-docker-images
- **Crear imágenes de contenedor** En la documentación oficial de Docker. https://docs.docker.com/get-started/docker-concepts/building-images/

- **Mantenerse actualizado con imágenes de contenedor de .NET**
  https://devblogs.microsoft.com/dotnet/staying-up-to-date-with-net-container-images/

- **Uso conjunto de .NET y Docket: actualización de DockerCon de 2018**
  https://devblogs.microsoft.com/dotnet/using-net-and-docker-together-dockercon-2018-update/

Uso de repositorios de imágenes multiarquitectura

Un solo repositorio puede contener variantes de plataforma, como una imagen de Linux y una imagen de Windows. Esta característica permite a los proveedores como Microsoft (creadores de imágenes base) crear un único repositorio que cubra varias plataformas (es decir, Linux y Windows). Por ejemplo, el repositorio .NET disponible en el registro de Docker Hub proporciona compatibilidad con Linux y Windows Nano Server mediante el mismo nombre de repositorio.

Si especifica una etiqueta, se toma como destino una plataforma explícita, como en los casos siguientes:

- `mcr.microsoft.com/dotnet/aspnet:8.0-bullseye-slim`
  Destinos: solo entorno de ejecución de .NET 8 en Linux

- `mcr.microsoft.com/dotnet/aspnet:8.0-nanoserver-ltsc2022`
  Destinos: solo entorno de ejecución de .NET 8 en Windows Nano Server

Pero, si se especifica el mismo nombre de imagen, incluso con la misma etiqueta, las imágenes multiarquitectura (como la imagen `aspnet`) usan la versión de Linux o Windows según el sistema operativo del host de Docker que se vaya a implementar, como se muestra en el ejemplo siguiente:

- `mcr.microsoft.com/dotnet/aspnet:8.0`
  Arquitectura múltiple: solo entorno de ejecución de .NET 8 en Linux o Windows Nano Server según el sistema operativo del host de Docker

De esta forma, al extraer una imagen de un host de Windows, se extrae la variante de Windows, y al extraer el mismo nombre de imagen de un host de Linux, se extrae la variante de Linux.

Compilaciones de varias fases en Dockerfile

El Dockerfile es similar a un script por lotes. Es similar a lo que haría si tuviera que configurar el equipo desde la línea de comandos.

Comienza con una imagen base que configura el contexto inicial, es como el sistema de archivos de inicio, que se coloca sobre el sistema operativo del host. No es un sistema operativo, pero se puede considerar como "el" sistema operativo dentro del contenedor.

La ejecución de cada línea de comandos crea una nueva capa en el sistema de archivos con los cambios de la anterior, por lo que, cuando se combinan, generan el sistema de archivos resultante.

Dado que cada nueva capa "descansa" sobre la anterior y el tamaño de la imagen resultante aumenta con cada comando, las imágenes pueden llegar a tener un gran tamaño si tienen que incluir, por ejemplo, el SDK necesario para compilar y publicar una aplicación.

Aquí es donde las compilaciones de varias fases entran en escena (a partir de Docker 17.05 y posterior) para hacer su magia.

La idea central es que puede separar el proceso de ejecución del Dockerfile en fases, donde una fase es una imagen inicial seguida de uno o más comandos, y la última fase determina el tamaño final de la imagen.

En resumen, las compilaciones de varias fases permiten dividir la creación en "fases" distintas y, luego, ensamblar la imagen final al tomar solo los directorios pertinentes de las fases intermedias. La estrategia general para usar esta característica es:

1. Usar una imagen base de SDK (no importa su tamaño), con todo lo necesario para compilar y publicar la aplicación en una carpeta

2. Usar una imagen base pequeña de solo el entorno de ejecución y copiar la carpeta de publicación de la fase anterior para generar una pequeña imagen final.

Probablemente la mejor manera de comprender las fases es analizar un archivo Dockerfile en detalle, línea a línea, así que vamos a comenzar con el Dockerfile inicial creado por Visual Studio al agregar compatibilidad con Docker a un proyecto y, luego, realizaremos algunas optimizaciones.

El Dockerfile inicial podría ser algo parecido a esto:

Dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 80

FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY src/Services/Catalog/Catalog.API/Catalog.API.csproj …
```

```
 8  COPY
src/BuildingBlocks/HealthChecks/src/Microsoft.AspNetCore.HealthChecks …
 9  COPY
src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions.HealthChecks …
10  COPY src/BuildingBlocks/EventBus/IntegrationEventLogEF/ …
11  COPY src/BuildingBlocks/EventBus/EventBus/EventBus.csproj …
12  COPY
src/BuildingBlocks/EventBus/EventBusRabbitMQ/EventBusRabbitMQ.csproj …
13  COPY
src/BuildingBlocks/EventBus/EventBusServiceBus/EventBusServiceBus.csproj …
14  COPY src/BuildingBlocks/WebHostCustomization/WebHost.Customization …
15  COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions …
16  COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions …
17  RUN dotnet restore src/Services/Catalog/Catalog.API/Catalog.API.csproj
18  COPY . .
19  WORKDIR /src/src/Services/Catalog/Catalog.API
20  RUN dotnet build Catalog.API.csproj -c Release -o /app
21
22  FROM build AS publish
23  RUN dotnet publish Catalog.API.csproj -c Release -o /app
24
25  FROM base AS final
26  WORKDIR /app
27  COPY --from=publish /app .
28  ENTRYPOINT ["dotnet", "Catalog.API.dll"]
```

Y estos son los detalles, línea a línea:

- **Línea 1:** Comience una fase con una imagen base "pequeña" de solo el entorno de ejecución, denomínela **base** para referencia.
- **Línea 2:** Cree el directorio **/app** de la imagen.
- **Línea 3:** Exponga el puerto **80**.
- **Línea 5:** Comience una nueva fase con una imagen "grande" para compilar y publicar. Denomínela **build** como referencia.
- **Línea 6:** Cree un directorio **/src** en la imagen.
- **Línea 7:** Hasta la línea 16, copie los archivos del proyecto **.csproj** a los que se hace referencia para poder restaurar los paquetes más adelante.
- **Línea 17:** Restaure los paquetes del proyecto **Catalog.API** y los proyectos a los que se hace referencia.
- **Línea 18:** Copie **todo el árbol de directorio de la solución** (excepto los archivos o directorios incluidos en el archivo **.dockerignore**) en el directorio **/src** de la imagen.

- **Línea 19:** Cambie la carpeta actual al proyecto **Catalog.API**.
- **Línea 20:** Compile el proyecto (y otras dependencias del proyecto) y use como salida el directorio **/app** de la imagen.
- **Línea 22:** Comience una nueva fase a partir de la compilación. Denomínela **publish** como referencia.
- **Línea 23:** Publique el proyecto (y las dependencias) y use como salida el directorio **/app** de la imagen.
- **Línea 25:** Comience una nueva fase a partir de **base** y denomínela **final**.
- **Línea 26:** Cambie el directorio actual a **/app**.
- **Línea 27:** Copie el directorio **/app** de la fase **publish** en el directorio actual.
- **Línea 28:** Defina el comando que se va a ejecutar cuando se inicie el contenedor.

Ahora vamos a examinar algunas optimizaciones para mejorar el rendimiento del proceso completo, lo que, en el caso de eShopOnContainers, significa aproximadamente 22 minutos o más para compilar la solución completa en contenedores de Linux.

Puede aprovechar la característica de caché de capas de Docker, que es bastante sencilla: si la imagen base y los comandos son los mismos que algunos ejecutados previamente, puede usar la capa resultante sin necesidad de ejecutar los comandos, con lo que se ahorra algo de tiempo.

Así, vamos a centrarnos en la fase **build**, las líneas 5 y 6 son prácticamente iguales, pero las líneas 7-17 son diferentes para cada servicio de eShopOnContainers, así que se tienen que ejecutar cada vez, pero si ha cambiado las líneas 7-16 a:

Dockerfile

```
COPY . .
```

Luego, sería igual para cada servicio, se copiaría la solución completa y se crearía una capa más grande pero:

1. El proceso de copia solo se ejecutaría la primera vez (y al recompilar si se modifica un archivo) y se usaría la memoria caché para todos los demás servicios y
2. Puesto que la imagen más grande se produce en una fase intermedia, no afecta al tamaño final de la imagen.

La siguiente optimización importante implica al comando `restore` ejecutado en la línea 17, que también es diferente para cada servicio de eShopOnContainers. Si cambia esa línea a:

Dockerfile

```
RUN dotnet restore
```

Restauraría los paquetes de toda la solución, pero, de nuevo, lo haría una sola vez, en lugar de las 15 veces con la estrategia actual.

Pero `dotnet restore` únicamente se ejecuta si hay un solo archivo de proyecto o solución en la carpeta, así que lograrlo es un poco más complicado y la forma de solucionarlo, sin entrar en demasiados detalles, es esta:

1. Agregue las líneas siguientes a **.dockerignore**:
   - `*.sln`, para omitir todos los archivos de solución del árbol de la carpeta principal
   - `!eShopOnContainers-ServicesAndWebApps.sln`, para incluir solo este archivo de solución.
2. Incluya el argumento `/ignoreprojectextensions:.dcproj` en `dotnet restore`, de modo que también omita el proyecto docker-compose y solo restaure los paquetes de la solución eShopOnContainers-ServicesAndWebApps.

Para la optimización final, resulta que la línea 20 es redundante, ya que la línea 23 también compila la aplicación y viene, básicamente, justo después de la línea 20, así que ahí tenemos otro comando que usa mucho tiempo.

El archivo resultante es entonces:

Dockerfile

```
 1  FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
 2  WORKDIR /app
 3  EXPOSE 80
 4
 5  FROM mcr.microsoft.com/dotnet/sdk:8.0 AS publish
 6  WORKDIR /src
 7  COPY . .
 8  RUN dotnet restore /ignoreprojectextensions:.dcproj
 9  WORKDIR /src/src/Services/Catalog/Catalog.API
10  RUN dotnet publish Catalog.API.csproj -c Release -o /app
11
12  FROM base AS final
```

```
WORKDIR /app
COPY --from=publish /app .
ENTRYPOINT ["dotnet", "Catalog.API.dll"]
```

Creación de la imagen base desde cero

Puede crear su propia imagen base de Docker desde cero. Este escenario no se recomienda para usuarios que se están iniciando en Docker, pero si quiere establecer los bits específicos de su propia imagen base, puede hacerlo.

Recursos adicionales

- **Multi-arch .NET Core images (Imágenes de .NET Core multiarquitectura)**. https://github.com/dotnet/announcements/issues/14
- **Create a base image (Crear una imagen base)** . Documentación oficial de Docker. https://docs.docker.com/develop/develop-images/baseimages/

Paso 3. Crear las imágenes de Docker personalizadas e insertar la aplicación o el servicio en ellas

Debe crear una imagen relacionada para cada servicio de la aplicación. Si la aplicación está formada por un único servicio o aplicación web, solo necesita una imagen.

Tenga en cuenta que las imágenes de Docker se compilan automáticamente en Visual Studio. Los pasos siguientes solo son necesarios para el flujo de trabajo de editor/CLI y se explican para aclarar lo que sucede en segundo plano.

Como desarrollador, debe desarrollar y probar en local hasta que inserte una característica o cambio completados en el sistema de control de código fuente (por ejemplo, en GitHub). Esto significa que tiene que crear las imágenes de Docker e implementar contenedores en un host de Docker local (máquina virtual de Windows o Linux) y ejecutar, probar y depurar en esos contenedores locales.

Para crear una imagen personalizada en el entorno local mediante la CLI de Docker y el Dockerfile, puede usar el comando docker build, como se muestra en la figura 5-5.

```
PS C:\dev\netcore-webapi-microservice-docker> docker build -t cesardl/netcore-webapi-microservice-docker:first .
Sending build context to Docker daemon 1.148 MB
Step 1 : FROM microsoft/dotnet:latest
latest: Pulling from microsoft/dotnet
5c90d4a2d1a8: Downloading [=================>                                  ] 18.34 MB/51.35 MB
ab30c63719b1: Downloading [==================================================> ] 18.48 MB/18.55 MB
c6072700a242: Downloading [=====================>                              ] 18.34 MB/42.53 MB
121d7eef6c20: Waiting
eb57cf4f29ee: Waiting
b2c5ae2d325b: Waiting
```

Figura 5-5. Creación de una imagen personalizada de Docker

De forma opcional, en lugar de ejecutar directamente docker build desde la carpeta del proyecto, primero puede generar una carpeta que se pueda implementar con las bibliotecas de .NET y los binarios necesarios mediante la ejecución de `dotnet publish` y, luego, usar el comando `docker build`.

Esto crea una imagen de Docker con el nombre `cesardl/netcore-webapi-microservice-docker:first`. En este caso, `:first` es una etiqueta que representa una versión específica. Puede repetir este paso para cada imagen personalizada que tenga que crear para la aplicación de Docker compuesta.

Cuando una aplicación se compone de varios contenedores (es decir, es una aplicación de varios contenedores), también puede usar el comando `docker-compose up --build` para compilar todas las imágenes relacionadas con un solo comando al usar los metadatos expuestos en los archivos relacionados docker-compose.yml.

Puede encontrar las imágenes existentes en el repositorio local mediante el comando docker images, como se muestra en la figura 5-6.

```
PS C:\dev\netcore-webapi-microservice-docker> docker images
REPOSITORY                                      TAG                 IMAGE ID            CREATED             SIZE
cesardl/netcore-webapi-microservice-docker      first               384c4ac1809b        4 minutes ago       579.8 MB
microsoft/dotnet                                latest              49aaf5daa850        30 hours ago        548.6 MB
ubuntu                                          latest              cf62323fa025        5 days ago          125 MB
hello-world                                     latest              c54a2cc56cbb        12 days ago         1.848 kB
```

Figura 5-6. Visualización de imágenes existentes mediante el comando docker images

Creación de imágenes de Docker con Visual Studio

Cuando se usa Visual Studio para crear un proyecto con compatibilidad con Docker, no se crea una imagen de forma explícita. La imagen se crea automáticamente al presionar F5 (o Ctrl+F5) para ejecutar el servicio o la aplicación a los que se ha aplicado Docker. Este paso es automático en Visual Studio y no lo verá, pero es importante saber lo que ocurre en segundo plano.

Paso 4. Definir los servicios en docker-compose.yml al compilar una aplicación de Docker de varios contenedores

El archivo docker-compose.yml permite definir un conjunto de servicios relacionados para implementarlos como una aplicación compuesta con comandos de implementación. También configura sus relaciones de dependencia y la configuración de ejecución.

Para usar un archivo docker-compose.yml, debe crear el archivo en la carpeta de solución principal o raíz, con contenido similar al del ejemplo siguiente:

yml

```yml
version: '3.4'

services:

  webmvc:
    image: eshop/web
    environment:
      - CatalogUrl=http://catalog-api
      - OrderingUrl=http://ordering-api
    ports:
      - "80:80"
    depends_on:
      - catalog-api
      - ordering-api

  catalog-api:
    image: eshop/catalog-api
    environment:
      - ConnectionString=Server=sqldata;Port=1433;Database=CatalogDB;…
    ports:
      - "81:80"
    depends_on:
      - sqldata

  ordering-api:
    image: eshop/ordering-api
    environment:
      - ConnectionString=Server=sqldata;Database=OrderingDb;…
    ports:
```

```
      - "82:80"
    extra_hosts:
      - "CESARDLBOOKVHD:10.0.75.1"
    depends_on:
      - sqldata

  sqldata:
    image: mcr.microsoft.com/mssql/server:latest
    environment:
      - SA_PASSWORD=[PLACEHOLDER]
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
```

> ⓘ **Importante**
>
> Microsoft recomienda usar el flujo de autenticación más seguro disponible. Si se conecta a Azure SQL, el método de autenticación recomendado es **Identidades administradas para recursos de Azure**.

Este archivo docker-compose.yml es una versión simplificada y combinada. Contiene datos de configuración estáticos para cada contenedor (como el nombre de la imagen personalizada), que siempre son necesarios, junto con información de configuración que puede depender del entorno de implementación, como la cadena de conexión. En secciones posteriores se enseña a dividir la configuración de docker-compose.yml en varios archivos docker-compose y a reemplazar los valores según el entorno y el tipo de ejecución (depuración o versión).

El ejemplo de archivo docker-compose.yml define cuatro servicios: el servicio `webmvc` (una aplicación web), dos microservicios (`ordering-api` y `basket-api`) y un contenedor de fuente de datos, `sqldata`, según el servidor de SQL Server para Linux que se ejecute como contenedor. Cada servicio se implementa como un contenedor, por lo que se necesita una imagen de Docker para cada uno de ellos.

El archivo docker-compose.yml especifica no solo qué contenedores se van a usar, sino cómo se configuran individualmente. Por ejemplo, la definición del contenedor `webmvc` en el archivo .yml:

- Usa una imagen `eshop/web:latest` precompilada. Pero también puede configurar la imagen de modo que se compile como parte de la ejecución de docker-compose con una configuración adicional basada en una compilación: sección del archivo docker-compose.
- Inicializa dos variables de entorno (CatalogUrl y OrderingUrl).

- Reenvía el puerto 80 expuesto en el contenedor al puerto 80 externo del equipo de host.
- Vincula la aplicación web al catálogo y el servicio de orden con el valor depends_on. Esto hace que el servicio espere hasta que se inician los servicios.

Se volverá a hablar del archivo docker-compose.yml en una sección posterior, cuando se trate la implementación de microservicios y aplicaciones de varios contenedores.

Trabajo con docker-compose.yml en Visual Studio 2022

Además de agregar un Dockerfile a un proyecto, como se ha mencionado antes, Visual Studio 2017 (a partir de la versión 15.8 en adelante) puede agregar compatibilidad de orquestador con Docker Compose a una solución.

Cuando se agrega compatibilidad de orquestador de contenedores, como se muestra en la figura 5-7, por primera vez, Visual Studio crea el Dockerfile para el proyecto y un nuevo proyecto (sección servicio) en la solución con varios archivos `docker-compose*.yml` globales y, luego, agrega el proyecto a esos archivos. Luego puede abrir los archivos docker-compose.yml y actualizarlos con otras características.

Repita esta operación para cada proyecto que quiera incluir en el archivo docker-compose.yml.

En el momento de redactar este artículo, Visual Studio es compatible con los orquestadores de **Docker Compose**.

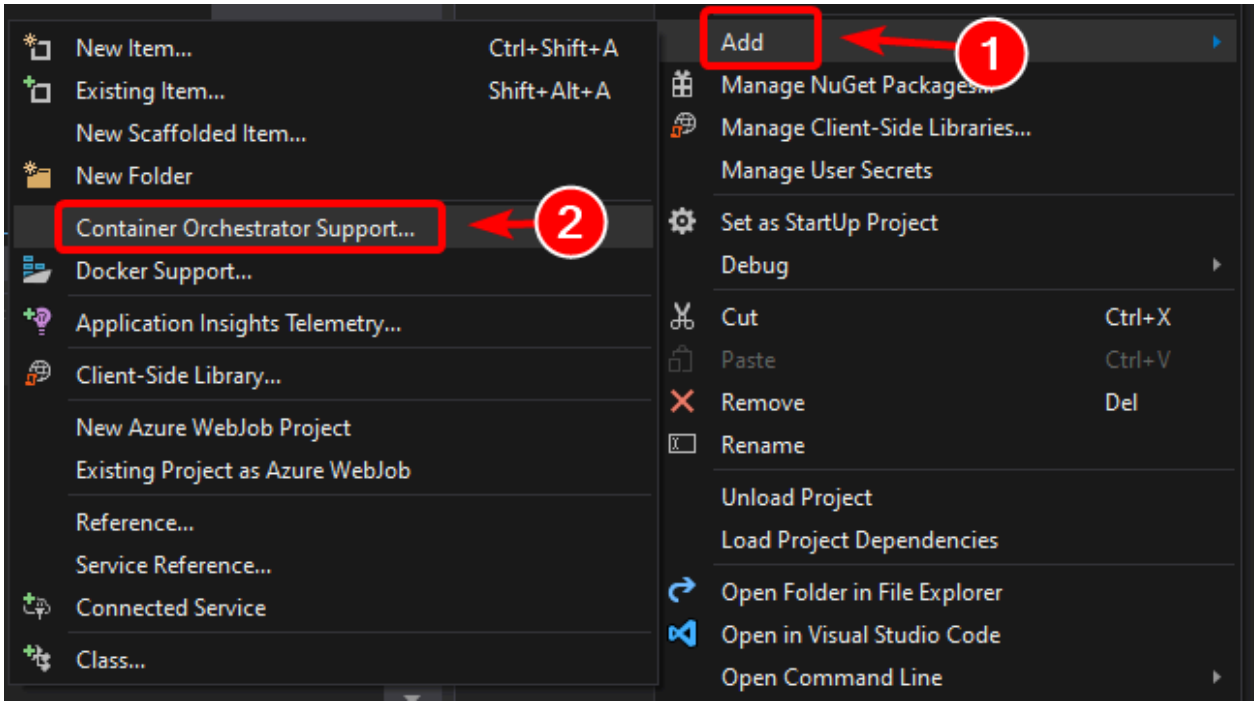


Figura 5-7. Compatibilidad agregada con Docker en Visual Studio 2022 al hacer clic con el botón derecho en un proyecto de ASP.NET Core

Después de agregar compatibilidad de orquestador a la solución en Visual Studio, también se ve un nuevo nodo (en el archivo de proyecto `docker-compose.dcproj`) en el Explorador de soluciones que contiene los archivos docker-compose.yml agregados, como se muestra en la figura 5-8.

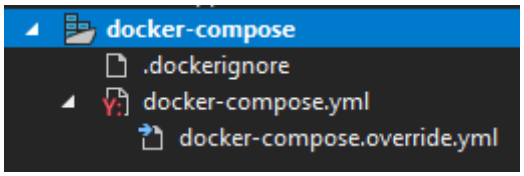


Figura 5-8. El nodo de árbol **docker-compose** se agrega en el Explorador de soluciones de Visual Studio 2022

Puede implementar una aplicación de varios contenedores con un único archivo docker-compose.yml mediante el comando `docker-compose up`. Pero Visual Studio agrega un grupo de ellos para que pueda reemplazar valores en función del entorno (desarrollo o producción) y el tipo de ejecución (versión o depuración). Esta capacidad se explica en secciones posteriores.

Paso 5. Compilar y ejecutar la aplicación

Si la aplicación solo tiene un contenedor, puede ejecutarla mediante su implementación en el host de Docker (máquina virtual o servidor físico). Sin embargo, si la aplicación contiene varios servicios, se puede implementar como aplicación compuesta, ya sea mediante un solo comando de la CLI `docker-compose up)` o con Visual Studio, que usará ese comando en segundo plano. Echemos un vistazo a las distintas opciones.

Opción A: Ejecución de una aplicación de un solo contenedor

Uso de la CLI de Docker

Puede ejecutar un contenedor de Docker mediante el comando `docker run`, como se muestra en la figura 5-9:

Consola

```
docker run -t -d -p 80:5000 cesardl/netcore-webapi-microservice-docker:first
```

El comando anterior crea una nueva instancia de contenedor a partir de la imagen especificada cada vez que se ejecuta. Puede usar el parámetro `--name` para asignar un nombre al contenedor y, luego, usar `docker start {name}` (o el identificador del contenedor o el nombre automático) para ejecutar una instancia de contenedor existente.

```
PS C:\dev\netcore-webapi-microservice-docker> docker run -t -d -p 80:5000 cesardl/netcore-webapi-microservice-docker:first
d96975a683b0a9411595816f63be6c135801878b8a85181a4d86dc848ea4ca6f
```

Figura 5-9. Ejecución de un contenedor de Docker mediante el comando docker run

En este caso, el comando enlaza el puerto interno 5000 del contenedor con el puerto 80 del equipo de host. Esto significa que el host escucha en el puerto 80 y reenvía al puerto 5000 del contenedor.

El hash que se muestra es el identificador del contenedor y además se le ha asignado un nombre legible aleatorio si no se ha usado la opción `--name`.

Uso de Visual Studio

Si no ha agregado compatibilidad de orquestador de contenedores, también puede ejecutar una aplicación de un solo contenedor si presiona `Ctrl`+`F5` y además puede usar `F5` para depurar la aplicación del contenedor. El contenedor se ejecuta localmente mediante docker run.

Opción B: Ejecución de una aplicación de varios contenedores

En la mayoría de los escenarios de empresa, una aplicación de Docker se compone de varios servicios, lo que significa que hay que ejecutar una aplicación de varios contenedores, como se muestra en la figura 5-10.

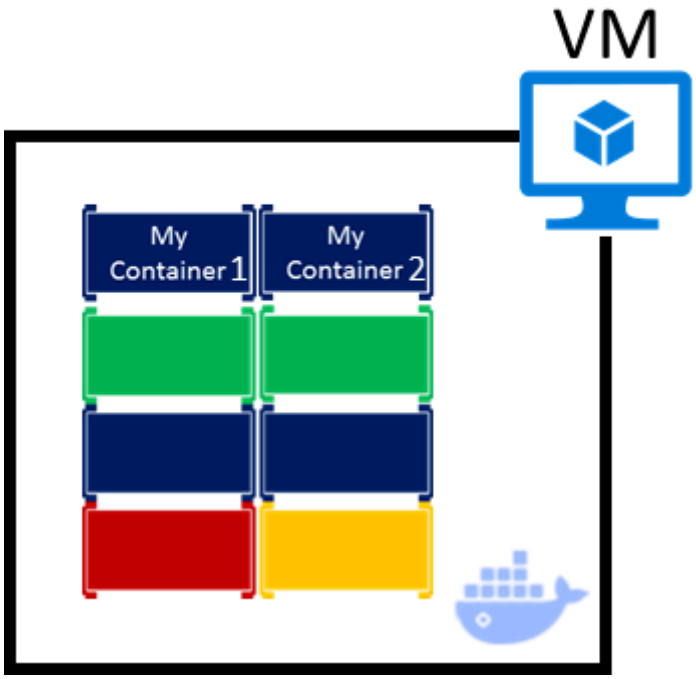


Figura 5-10. Máquina virtual con contenedores de Docker implementados

Uso de la CLI de Docker

Para ejecutar una aplicación de varios contenedores con la CLI de Docker, use el comando `docker-compose up`. Este comando usa el archivo **docker-compose.yml** que hay en el nivel de solución para implementar una aplicación de varios contenedores. La figura 5-11 muestra los resultados de la ejecución del comando desde el directorio principal de la solución, que contiene el archivo docker-compose.yml.

```
PS C:\Dev\WebApplication> docker-compose up
Recreating webapplication_webapplication_1
Attaching to webapplication_webapplication_1
webapplication_1   | Hosting environment: Production
webapplication_1   | Content root path: /app
webapplication_1   | Now listening on: http://*:80
webapplication_1   | Application started. Press Ctrl+C to shut down.
```

Figura 5-11. Resultados del ejemplo al ejecutar el comando docker-compose up

Después de la ejecución del comando docker-compose up, la aplicación y sus contenedores relacionados se implementan en el host de Docker, como se muestra en la figura 5-10.

Uso de Visual Studio

La ejecución de una aplicación de varios contenedores mediante Visual Studio 2019 no puede ser más sencilla. Simplemente presione `Ctrl`+`F5` para ejecutar o `F5` para depurar, como de costumbre, con lo que se configura el proyecto **docker-compose**

como proyecto de inicio. Visual Studio controla toda la configuración necesaria, por lo que puede crear puntos de interrupción como de costumbre y depurar lo que finalmente se convierte en procesos independientes que se ejecutan en "servidores remotos", con el depurador ya adjuntado, con facilidad.

Como se ha mencionado antes, cada vez que se agrega compatibilidad con soluciones de Docker a un proyecto de una solución, ese proyecto se configura en el archivo global (nivel de solución) docker-compose.yml, lo que permite ejecutar o depurar la solución completa al mismo tiempo. Visual Studio inicia un contenedor para cada proyecto que tiene habilitada la compatibilidad con soluciones de Docker y realiza todos los pasos internos automáticamente (dotnet publish, docker build, etc.).

Si quiere echar un vistazo al trabajo monótono, vea el archivo:

`{root solution folder}\obj\Docker\docker-compose.vs.debug.g.yml`

Lo importante aquí es que, como se muestra en la figura 5-12, en Visual Studio 2019 hay un comando adicional de **Docker** para la acción de la tecla F5. Esta opción permite ejecutar o depurar una aplicación de varios contenedores mediante la ejecución de todos los contenedores definidos en los archivos docker-compose.yml en el nivel de solución. La capacidad de depurar las soluciones de varios contenedores significa que puede establecer varios puntos de interrupción, cada uno en un proyecto diferente (contenedor) y, durante la depuración desde Visual Studio, detenerse en los puntos de interrupción definidos en los distintos proyectos y en ejecución en contenedores diferentes.

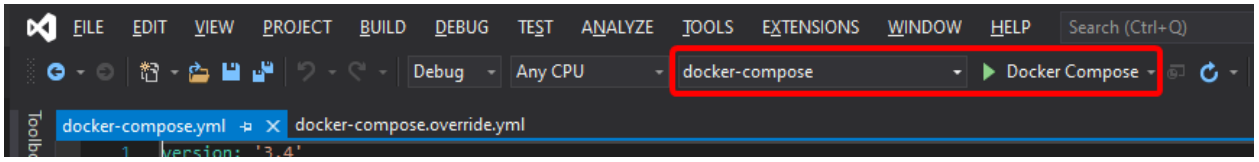


Figura 5-12. Ejecución de aplicaciones de varios contenedores en Visual Studio 2022

Recursos adicionales

- **Implementación de un contenedor de ASP.NET en un host remoto de Docker** https://learn.microsoft.com/visualstudio/containers/hosting-web-apps-in-docker

Nota sobre las pruebas y la implementación con orquestadores

Los comandos docker-compose up y docker run (o la ejecución y depuración de los contenedores en Visual Studio) son adecuados para probar contenedores en el entorno

de desarrollo. Sin embargo, no debe usar este enfoque para implementaciones de producción donde se deba tener como destino orquestadores como Kubernetes o Service Fabric. Si usa Kubernetes, tiene que usar pods para organizar los contenedores y los servicios para conectarlos en red. También se usan implementaciones organizar la creación y la modificación de pods.

Paso 6. Probar la aplicación de Docker con el host local de Docker

Este paso varía en función de lo que haga la aplicación. En una aplicación web de .NET sencilla implementada como un único contenedor o servicio, puede acceder al servicio si abre un explorador en el host de Docker y va a ese sitio, como se muestra en la figura 5-13. (Si la configuración del Dockerfile asigna el contenedor a un puerto del host distinto al 80, incluya el puerto del host en la dirección URL).

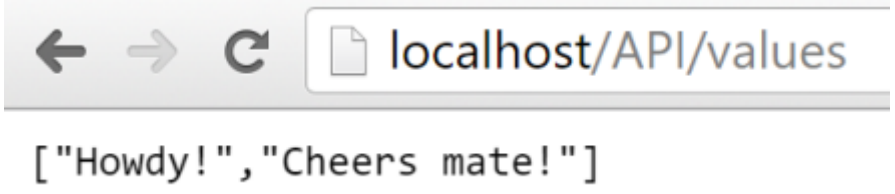


Figura 5-13. Ejemplo de prueba de la aplicación de Docker en local mediante localhost

Si localhost no apunta a la IP del host de Docker (de forma predeterminada, al usar Docker CE, debería hacerlo), para ir al servicio, use la dirección IP de la tarjeta de red del equipo.

Esta dirección URL en el explorador usa el puerto 80 para el ejemplo de contenedor determinado que se está analizando. Pero, internamente, las solicitudes se redirigen al puerto 5000, porque así es como se ha implementado con el comando docker run, como se ha explicado en el paso anterior.

También puede probar la aplicación con la CURL del terminal, como se muestra en la figura 5-14. En una instalación de Docker en Windows, el valor predeterminado de la IP del host de Docker es siempre 10.0.75.1, además de la dirección IP real del equipo.

```
PS C:\dev\netcore-webapi-microservice-docker> curl http://10.0.75.1/API/values


StatusCode        : 200
StatusDescription : OK
Content           : ["Howdy!","Cheers mate!"]
RawContent        : HTTP/1.1 200 OK
                    Transfer-Encoding: chunked
                    Content-Type: application/json; charset=utf-8
                    Date: Thu, 14 Jul 2016 19:48:18 GMT
                    Server: Kestrel

                    ["Howdy!","Cheers mate!"]
Forms             : {}
Headers           : {[Transfer-Encoding, chunked], [Content-Type, application/json;
                    charset=utf-8], [Date, Thu, 14 Jul 2016 19:48:18 GMT], [Server, Kestrel]}
Images            : {}
InputFields       : {}
Links             : {}
ParsedHtml        : mshtml.HTMLDocumentClass
RawContentLength  : 25
```

Figura 5-14. Ejemplo de prueba de la aplicación de Docker en local mediante CURL

Prueba y depuración de contenedores con Visual Studio 2022

Al ejecutar y depurar los contenedores con Visual Studio 2022, puede depurar la aplicación de .NET prácticamente de la misma manera que como lo haría al realizar la ejecución sin contenedores.

Pruebas y depuración sin Visual Studio

Si su desarrollo se basa en el enfoque de editor/CLI, la depuración de contenedores es más difícil y es probable que prefiera hacerlo mediante la generación de seguimientos.

Recursos adicionales

- **Inicio rápido: Docker en Visual Studio.**
  https://learn.microsoft.com/visualstudio/containers/container-tools

- **Depuración de aplicaciones en un contenedor de Docker local**
  https://learn.microsoft.com/visualstudio/containers/edit-and-refresh

Flujo de trabajo simplificado al desarrollar contenedores con Visual Studio

En la práctica, el flujo de trabajo cuando se usa Visual Studio es mucho más sencillo que si se usa el enfoque de editor/CLI. La mayoría de los pasos que necesita Docker relacionados con el Dockerfile y los archivos docker-compose.yml están ocultos o se han simplificado con Visual Studio, como se muestra en la figura 5-15.

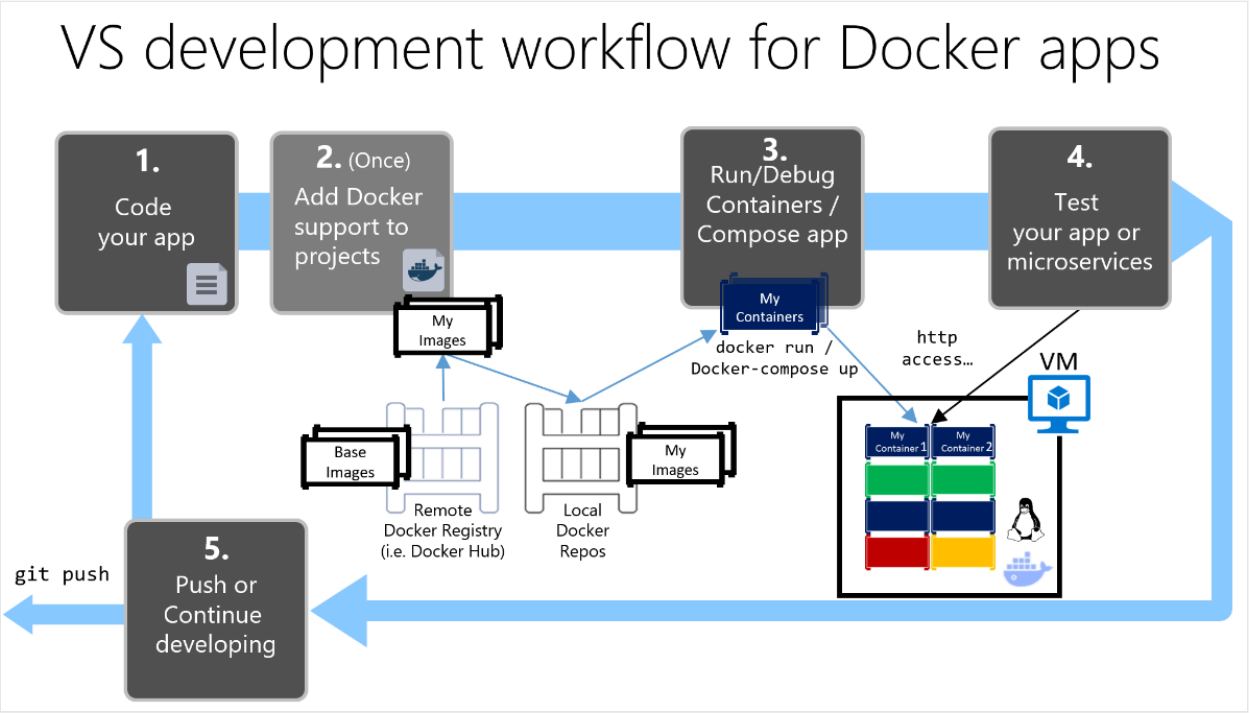


Figura 5-15. Flujo de trabajo simplificado al desarrollar con Visual Studio

Además, debe realizar el paso 2 (agregar compatibilidad con Docker a los proyectos) una sola vez. Por lo tanto, el flujo de trabajo es similar a las tareas de desarrollo habituales cuando se usa .NET para cualquier otro desarrollo. Debe saber qué está sucediendo en segundo plano (el proceso de compilación de imágenes, qué imágenes base usa, la implementación de contenedores, etc.) y, a veces, también debe editar el Dockerfile o el archivo docker-compose.yml para personalizar comportamientos. Pero con Visual Studio se simplifica enormemente la mayor parte del trabajo, lo que mejora mucho la productividad.

Uso de comandos de PowerShell en un Dockerfile para configurar contenedores de Windows

Los contenedores de Windows permiten convertir las aplicaciones de Windows existentes en imágenes de Docker e implementarlas con las mismas herramientas que el resto del ecosistema de Docker. Para usar contenedores de Windows, ejecute comandos de PowerShell en el Dockerfile, como se muestra en el ejemplo siguiente:

Dockerfile

```
FROM mcr.microsoft.com/windows/servercore
LABEL Description="IIS" Vendor="Microsoft" Version="10"
RUN powershell -Command Add-WindowsFeature Web-Server
CMD [ "ping", "localhost", "-t" ]
```

En este caso se usa una imagen base de Windows Server Core (el valor FROM) y se instala IIS con un comando de PowerShell (el valor RUN). Del mismo modo, también se pueden usar comandos de PowerShell para configurar otros componentes como ASP.NET 4.x, .NET Framework 4.6 o cualquier otro software de Windows. Por ejemplo, el siguiente comando en un Dockerfile configura ASP.NET 4.5:

Dockerfile

```
RUN powershell add-windowsfeature web-asp-net45
```

Recursos adicionales

- **aspnet-docker/Dockerfile.** Comandos de PowerShell de ejemplo para ejecutar desde Dockerfiles a fin de incluir características de Windows. https://github.com/Microsoft/aspnet-docker/blob/master/4.7.1-windowsservercore-ltsc2016/runtime/Dockerfile

Anterior Siguiente

Diseñar y desarrollar aplicaciones .NET basadas en varios contenedores y microservicios

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El desarrollo de aplicaciones de microservicios en contenedores significa que está creando aplicaciones de varios contenedores. Sin embargo, una aplicación de varios contenedores también podría ser más sencilla (por ejemplo, una aplicación de tres niveles) y podría no compilarse con una arquitectura de microservicios.

Anteriormente se planteó la pregunta "¿Es necesario Docker al compilar una arquitectura de microservicios?" La respuesta es un claro no. Docker es un habilitador y puede proporcionar ventajas significativas, pero los contenedores y Docker no son un requisito difícil para los microservicios. Por ejemplo, podría crear una aplicación basada en microservicios con o sin Docker al usar Azure Service Fabric, que admite microservicios que se ejecutan como procesos simples o como contenedores de Docker.

Sin embargo, si sabe cómo diseñar y desarrollar una aplicación basada en microservicios que también se basa en contenedores de Docker, podrá diseñar y desarrollar cualquier otro modelo de aplicación más sencillo. Por ejemplo, puede diseñar una aplicación de tres niveles que también requiera un enfoque de varios contenedores. Debido a eso, y dado que las arquitecturas de microservicios son una tendencia importante en el mundo del contenedor,

esta sección se centra en una implementación de arquitectura de microservicios mediante contenedores de Docker.

Siguiente anterior

Diseño de una aplicación orientada a microservicios

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Esta sección se centra en el desarrollo de una aplicación empresarial hipotética 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, a continuación, devuelve respuestas HTML, JSON o XML. Se dirá que la aplicación debe admitir varios clientes, incluidos los exploradores de escritorio que ejecutan aplicaciones de página única (SPA), aplicaciones web tradicionales, aplicaciones web móviles y aplicaciones móviles nativas. La aplicación también puede exponer una API para que los terceros los consuman. También debe ser capaz de integrar sus microservicios o aplicaciones externas de forma asincrónica, por lo que ese enfoque ayudará 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 responsables de controlar la interfaz de usuario y consumir servicios remotos.

- Lógica de dominio o de negocios. Este componente es la lógica de dominio de la aplicación.
- Lógica de acceso a la base de datos. Este componente consta de componentes de acceso a datos responsables de acceder a 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 una alta escalabilidad, al tiempo que permite que sus subsistemas verticales escalen horizontalmente de forma autónoma, ya que determinados subsistemas requerirán más escalabilidad que otras.

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

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 empresariales de la aplicación.
- Los nuevos miembros del equipo deben ser productivos rápidamente y la aplicación debe ser fácil de entender y modificar.
- La aplicación tendrá una evolución a largo plazo y reglas de negocio cambiantes.
- Necesita una buena capacidad de mantenimiento a largo plazo, lo que significa tener agilidad al implementar nuevos cambios en el futuro, a la vez que puede actualizar varios subsistemas con un impacto mínimo en los demás subsistemas.
- Le interesa la integración y la implementación continuas de la aplicación.
- Quiere aprovechar las tecnologías emergentes (marcos, lenguajes de programación, etc.) a la vez que evoluciona la aplicación. No desea realizar migraciones completas de la aplicación al pasar a nuevas tecnologías, ya que eso daría lugar a costos elevados e afectaría a la previsibilidad y la estabilidad de la aplicación.

Elección de una arquitectura

¿Cuál debe ser la arquitectura de implementación de aplicaciones? Las especificaciones de la aplicación, junto con el contexto de desarrollo, sugieren encarecidamente que debe diseñar la

aplicación descomponiéndolo en subsistemas autónomos en forma de microservicios y contenedores colaboradores, donde un microservicio es un contenedor.

En este enfoque, cada servicio (contenedor) implementa un conjunto de funciones cohesivas y estrechamente relacionadas. Por ejemplo, una aplicación puede constar de servicios como el servicio de catálogo, el servicio de pedidos, el servicio de cesta, el servicio de perfil 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, especialmente cuando se propagan actualizaciones con eventos de integración.

Los microservicios se desarrollan e implementan como contenedores independientemente entre sí. Este enfoque significa 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 le permite desacoplar completamente de otros microservicios. Cuando sea necesario, la coherencia entre las bases de datos de diferentes microservicios se logra mediante eventos de integración de nivel de aplicación (a través de un bus de eventos lógicos), como se controla en Segregación de responsabilidades de comandos y consultas (CQRS). Por ese motivo, las restricciones de negocio deben adoptar la coherencia final entre los múltiples microservicios y bases de datos relacionadas.

eShopOnContainers: una 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 empresarial hipotético que no conoce, hemos seleccionado un dominio empresarial conocido, es decir, una aplicación simplificada de comercio electrónico (e-shop) que presenta un catálogo de productos, toma pedidos de clientes, comprueba el inventario y realiza otras funciones empresariales. Este código fuente de la aplicación basada en contenedores está disponible en el repositorio de GitHub eShopOnContainers .

La aplicación consta de varios subsistemas, entre los que se incluyen varias interfaces de usuario de tienda en los front-end (una aplicación web y una aplicación móvil nativa), junto con los microservicios y contenedores de back-end necesarios para todas las operaciones del lado del 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.

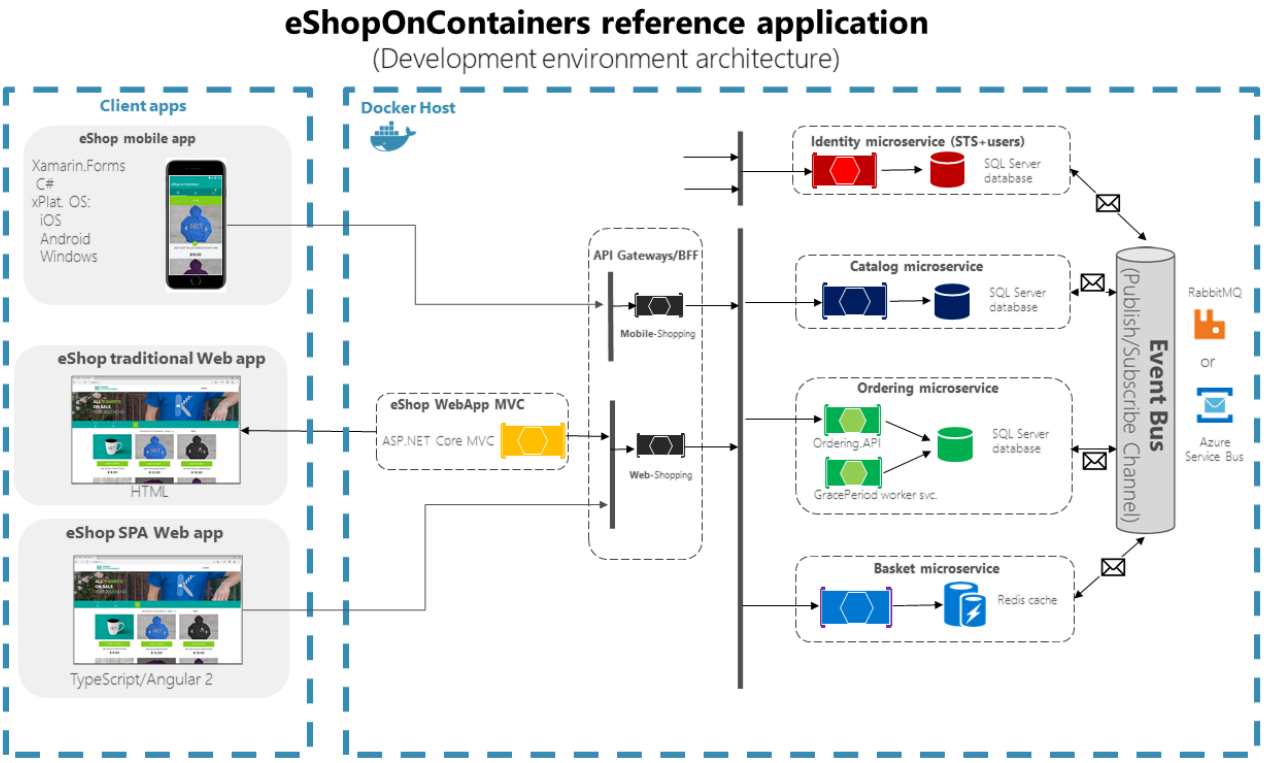


Figura 6-1. La arquitectura de la aplicación de referencia de eShopOnContainers para el entorno de desarrollo

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 los microservicios a través de la puerta de enlace de API.

Entorno de hospedaje. En la figura 6-1, verá varios contenedores implementados dentro de un único host de Docker. Eso sería el caso al implementar en un único host de Docker con el comando docker-compose up. Sin embargo, si usa un orquestador o un clúster de contenedor, cada contenedor podría ejecutarse en un host (nodo) diferente y cualquier nodo podría ejecutar cualquier número de contenedores, como se explicó anteriormente en la sección arquitectura.

Arquitectura de comunicación. La aplicación eShopOnContainers usa dos tipos de comunicación, según el tipo de acción funcional (consultas frente a actualizaciones y transacciones):

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

- Comunicación asincrónica basada en eventos. Esta comunicación se produce a través de un bus de eventos para propagar las actualizaciones entre microservicios o para integrarse con aplicaciones externas. El bus de eventos se puede implementar con

cualquier tecnología de infraestructura de agente de mensajería como RabbitMQ o mediante buses de servicio 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 ejecutan como contenedores a través de las direcciones URL públicas publicadas por las puertas de enlace de API.

Soberanía de datos por microservicio

En la aplicación de ejemplo, cada microservicio posee su propia base de datos o origen de datos, aunque todas las bases de datos de SQL Server se implementan como un único contenedor. Esta decisión de diseño solo se tomó para facilitar a un desarrollador obtener el código de GitHub, clonarlo y abrirlo en Visual Studio o Visual Studio Code. O bien, facilita la compilación de las imágenes personalizadas de Docker mediante la CLI de .NET y la CLI de Docker y, a continuación, implementarlas y ejecutarlas en un entorno de desarrollo de Docker. En cualquier caso, el uso de contenedores para orígenes de datos permite a los desarrolladores compilar e implementarlos en cuestión de minutos sin tener que aprovisionar una base de datos externa ni ningún otro origen de datos con dependencias difíciles en la infraestructura (nube o local).

En un entorno de producción real, para lograr alta disponibilidad y escalabilidad, las bases de datos deben basarse en servidores de bases de datos en la nube o en el entorno local, pero no en contenedores.

Por lo tanto, las unidades de implementación de microservicios (e incluso para las bases de datos de esta aplicación) son contenedores de Docker y la aplicación de referencia es una aplicación de varios contenedores que adopta principios de 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 evolucionar. Concretamente:

- Es fácil que un desarrollador comprenda y empiece a trabajar rápidamente con una 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 rápidamente, lo que hace que los desarrolladores sean productivos.
- Cada microservicio se puede diseñar, desarrollar e implementar independientemente de otros microservicios, lo que proporciona agilidad, ya que es más fácil implementar nuevas versiones de 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 utilizados al escalar horizontalmente que una arquitectura monolítica.

Puede dividir el trabajo de desarrollo entre varios equipos. Cada servicio puede ser propiedad de un único equipo de desarrollo. Cada equipo puede administrar, desarrollar, implementar y escalar su servicio independientemente del resto de los equipos.

Los problemas son más aislados. Si hay un problema en un servicio, solo ese servicio se ve afectado inicialmente (excepto cuando se usa el diseño incorrecto, con dependencias directas entre microservicios) y otros servicios pueden seguir controlando las solicitudes. En cambio, un componente que no funciona correctamente en una arquitectura de implementación monolítica puede reducir todo el sistema, especialmente cuando implica recursos, como una pérdida de memoria. Además, cuando se resuelve un problema en un microservicio, puede implementar solo el microservicio afectado sin afectar al resto de la aplicación.

Puede usar las tecnologías más recientes. Dado que puede empezar a desarrollar servicios de forma independiente y ejecutarlos en paralelo (gracias a los contenedores y .NET), puede empezar a usar las tecnologías y marcos más recientes de forma expediente en lugar de estar bloqueados en una pila o marco antiguos 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 a los desarrolladores cuando diseñan y compilan los servicios. Por ejemplo, los desarrolladores

deben implementar la comunicación entre servicios mediante protocolos como HTTP o AMQP, lo que agrega complejidad para las pruebas y el control de excepciones. También agrega latencia al sistema.

Complejidad de la implementación. Una aplicación que tiene docenas de tipos de microservicios y necesita alta escalabilidad (debe poder crear muchas instancias por servicio y equilibrar esos servicios en muchos hosts) significa un alto grado de complejidad de implementación para las operaciones y la administración de TI. Si no usa una infraestructura orientada a microservicios (como un orquestador y un programador), esa complejidad adicional puede requerir mucho más esfuerzos de desarrollo que la propia aplicación empresarial.

Transacciones atómicas. Las transacciones atómicas entre varios microservicios normalmente no son posibles. Los requisitos de negocio deben adoptar la coherencia final entre varios microservicios. Para obtener más información, consulte los desafíos del procesamiento de mensajes idempotentes.

Se han aumentado las necesidades de recursos globales (memoria total, unidades y recursos de red para todos los servidores o hosts). En muchos casos, cuando se reemplaza una aplicación monolítica por un enfoque de microservicios, la cantidad de recursos globales iniciales necesarios 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 el 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 al evolucionar las aplicaciones monolíticas, el aumento del uso de recursos suele ser un buen equilibrio para las aplicaciones a largo plazo.

Problemas con la comunicación directa de cliente a microservicio. Cuando la aplicación es grande, con docenas de microservicios, hay desafíos y limitaciones si la aplicación requiere comunicaciones directas de cliente a microservicio. Un problema es una posible discrepancia 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 muchas solicitudes independientes para redactar la interfaz de usuario, lo que puede ser ineficaz a través de Internet y sería poco práctico a través de una red móvil. Por lo tanto, las solicitudes de la aplicación cliente al sistema back-end deben minimizarse.

Otro problema con las comunicaciones directas de cliente a microservicio es que algunos microservicios podrían estar usando protocolos que no son fáciles de usar web. Un servicio podría usar un protocolo binario, mientras que otro servicio podría usar la mensajería AMQP. Esos protocolos no son compatibles con cortafuegos y se utilizan mejor internamente.

Normalmente, una aplicación debe usar protocolos como HTTP y WebSockets para la comunicación fuera del firewall.

Otro inconveniente con este enfoque directo de cliente a servicio es que dificulta la refactorización de los contratos de esos microservicios. Con el tiempo, es posible que los desarrolladores quieran cambiar la forma en que el sistema se divide en servicios. Por ejemplo, pueden combinar dos servicios o dividir un servicio en dos o más servicios. Sin embargo, 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 arquitectura, al diseñar y compilar una aplicación compleja basada en microservicios, puede considerar el uso de varias puertas de enlace de API específicas en lugar del enfoque de comunicación de cliente a microservicio más sencillo.

Creación de particiones de los microservicios. Por último, independientemente del enfoque que tome para la arquitectura de microservicios, otro desafío consiste en decidir cómo crear particiones de una aplicación de un extremo a otro en varios microservicios. Como se indica en la sección de arquitectura de la guía, hay varias técnicas y enfoques que puede adoptar. Básicamente, debe identificar las áreas de la aplicación que se desacoplan de las otras áreas y que tienen un número bajo de dependencias difíciles. En muchos casos, este enfoque se alinea con los servicios de creación de particiones por caso de uso. Por ejemplo, en nuestra aplicación de tienda electrónica, tenemos un servicio de pedidos responsable de toda la lógica de negocios relacionada con el proceso de pedido. También tenemos el servicio de catálogo y el servicio de cesta que implementan otras funcionalidades. Idealmente, cada servicio debe tener solo un pequeño conjunto de responsabilidades. Este enfoque es similar al principio de responsabilidad única (SRP) aplicado a las clases, que indica que una clase solo debe tener una razón para cambiar. Pero en este caso, se trata de microservicios, por lo que el ámbito será mayor que una sola clase. La mayor parte de todo, un microservicio debe ser autónomo, de un extremo a otro, incluida la responsabilidad de sus propios orígenes de datos.

Patrones de diseño y arquitectura externos frente a internos

La arquitectura externa es la arquitectura de microservicio compuesta por varios servicios, siguiendo los principios descritos en la sección arquitectura de esta guía. Sin embargo, dependiendo de la naturaleza de cada microservicio, e independientemente de la arquitectura de microservicios de alto nivel que elija, es habitual y a veces recomendable tener diferentes arquitecturas internas, cada una basada en diferentes patrones, para diferentes microservicios. Los microservicios incluso pueden usar diferentes tecnologías y lenguajes de programación. La figura 6-2 ilustra esta diversidad.

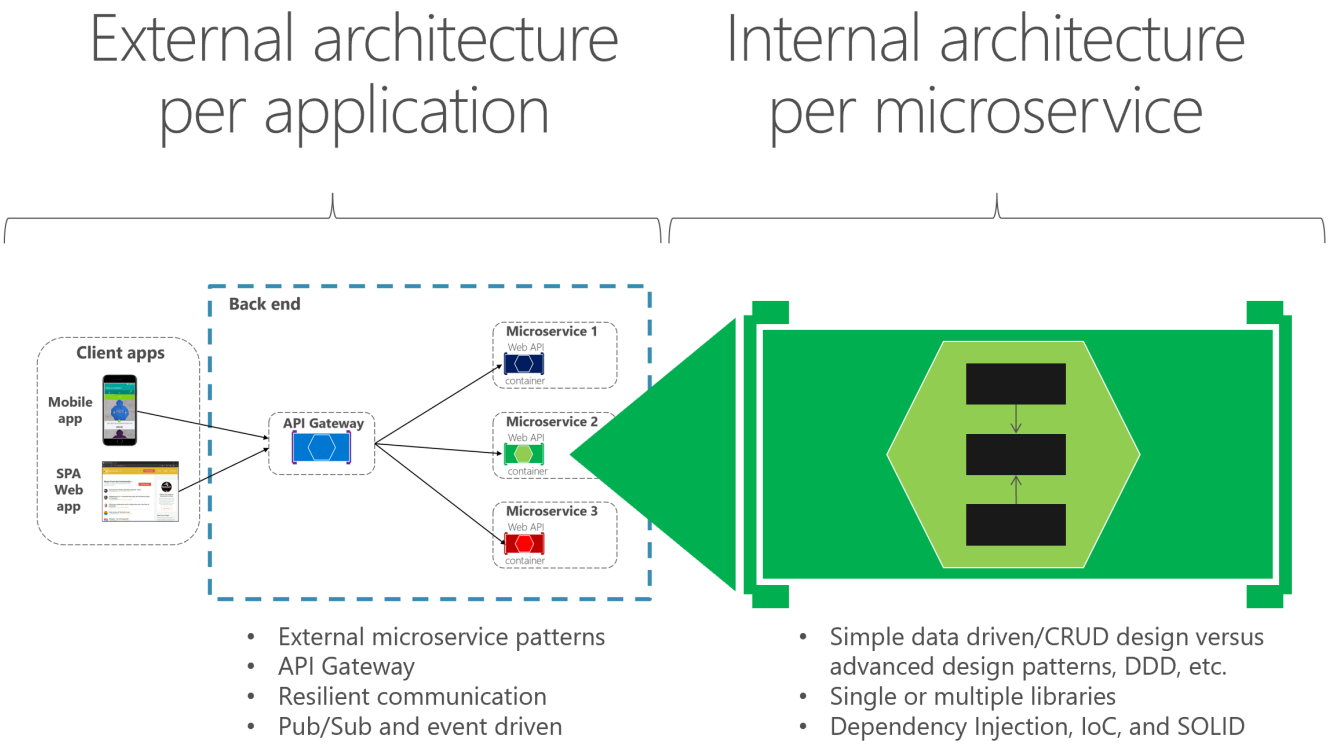


Figura 6-2. Arquitectura y diseño externos frente a internos

Por ejemplo, en nuestro *ejemplo eShopOnContainers* , los microservicios de catálogo, cesta y perfil de usuario son simples (básicamente, subsistemas CRUD). Por lo tanto, su arquitectura interna y diseño son sencillos. Sin embargo, es posible que tenga otros microservicios, como el microservicio de ordenación, que es más complejo y representa reglas de negocio cambiantes con un alto grado de complejidad del dominio. En casos como estos, es posible que quiera implementar patrones más avanzados dentro de un microservicio determinado, como los definidos con enfoques de diseño controlado por dominio (DDD), como estamos haciendo en el microservicio de pedidos *de eShopOnContainers* . (Revisaremos estos patrones DDD en la sección posterior que explica la implementación del microservicio de pedidos *de eShopOnContainers*).

Otra razón para una tecnología diferente 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 tiene como destino dominios de inteligencia artificial y aprendizaje automático, en lugar de un lenguaje de programación más orientado a objetos como C#.

La línea inferior es que cada microservicio puede tener una arquitectura interna diferente basada en diferentes patrones de diseño. No todos los microservicios deben implementarse mediante patrones DDD avanzados, ya que eso sería sobreinsgenierlos. De forma similar, los microservicios complejos con lógica de negocios que cambian constantemente no deben implementarse como componentes CRUD, o bien puede acabar con código de baja calidad.

El nuevo mundo: varios patrones arquitectónicos y microservicios políglota

Hay muchos patrones arquitectónicos usados por arquitectos de software y desarrolladores. A continuación se muestran algunos (mezclando estilos de arquitectura y patrones de arquitectura):

- CRUD simple, de nivel único y de capa única.
- N capas tradicional.
- N capas de diseño controlado por dominios.
- Arquitectura limpia (como se usa con eShopOnWeb)
- Segregación de responsabilidades de comandos y consultas (CQRS).
- arquitectura deEvent-Driven (EDA).

También puede compilar microservicios con muchas tecnologías y lenguajes, como ASP.NET Core Web API, NancyFx, ASP.NET Core SignalR (disponible con .NET Core 2 o posterior), F#, Node.js, Python, Java, C++, GoLang, etc.

El punto importante es que ningún patrón o estilo de arquitectura concreto, ni ninguna tecnología determinada, es adecuado para todas las situaciones. En la figura 6-3 se muestran algunos enfoques y tecnologías (aunque no en ningún orden determinado) que se podrían usar en distintos microservicios.

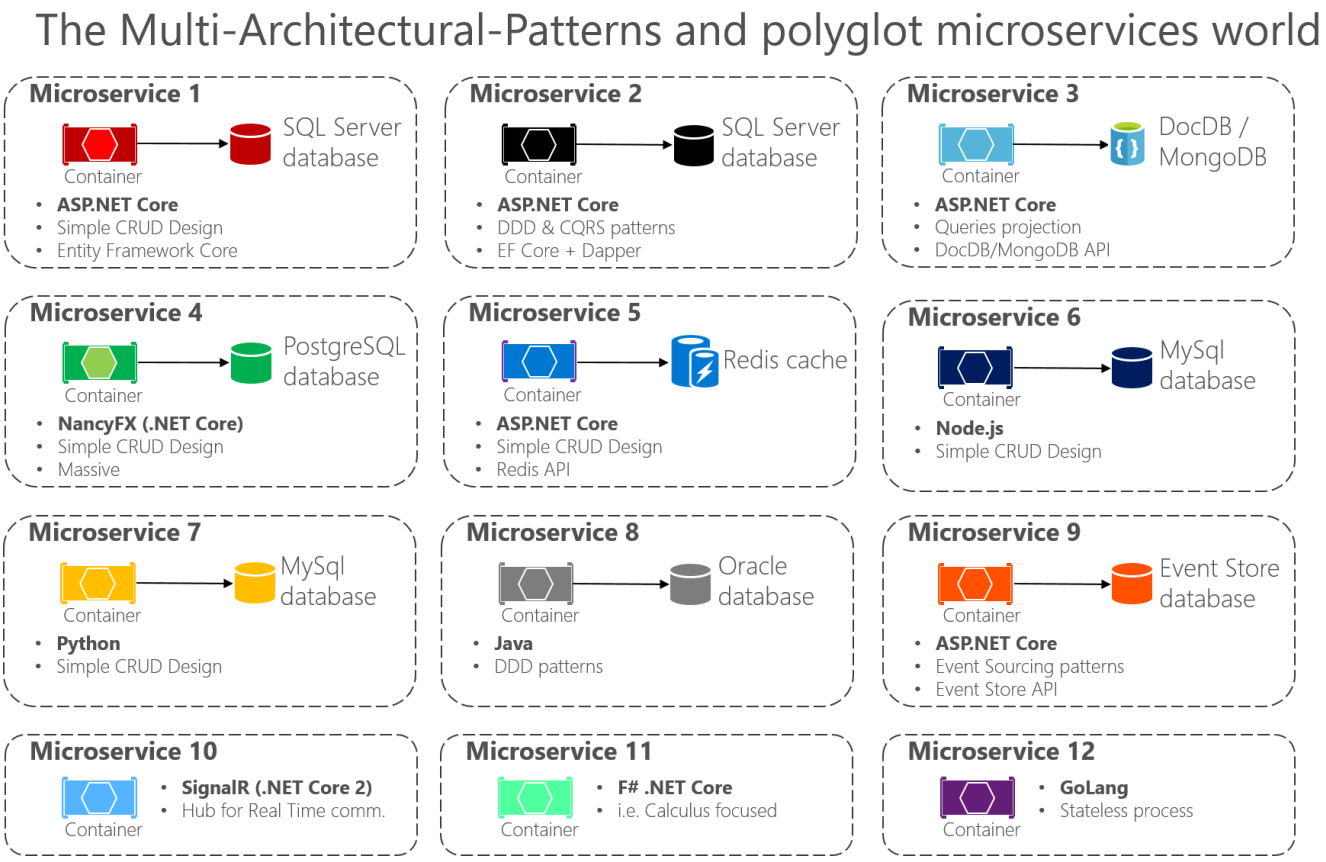


Figura 6-3. Patrones multiarquitectónicos y el mundo de microservicios políglotas

El patrón de arquitectura múltiple y los microservicios políglota significa que puede mezclar y combinar lenguajes y tecnologías según las necesidades de cada microservicio y pueden seguir comunicándose entre sí. Como se muestra en la figura 6-3, en aplicaciones compuestas de muchos microservicios (contextos enlazados en terminología de diseño controlado por dominio o simplemente "subsistemas" como microservicios autónomos), puede implementar cada microservicio de una manera diferente. Cada uno puede tener un patrón de arquitectura diferente y usar diferentes lenguajes y bases de datos en función de la naturaleza de la aplicación, los requisitos empresariales y las prioridades. En algunos casos, los microservicios pueden ser similares. Pero esto no suele ser el caso, ya que los requisitos y límites de contexto 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 DDD. Pero para su dominio principal o negocio principal, es posible que tenga que aplicar patrones más avanzados para abordar la complejidad empresarial con reglas de negocio cambiantes.

Especialmente cuando se trata de aplicaciones grandes compuestas por varios subsistemas, no se debe aplicar una arquitectura de nivel superior única basada en un único patrón de arquitectura. Por ejemplo, CQRS no debe aplicarse como una arquitectura de nivel superior para una aplicación completa, pero puede ser útil para un conjunto específico de servicios.

No hay una solución mágica ni un patrón de arquitectura correcto para cada caso determinado. No puede tener "un patrón de arquitectura para gobernarlos todos". En función de las prioridades de cada microservicio, debe elegir un enfoque diferente para cada uno, como se explica en las secciones siguientes.

Siguiente anterior

Creación de un microservicio CRUD sencillo controlado por datos

Artículo • 10/09/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En esta sección se describe cómo crear un microservicio sencillo que lleve a cabo operaciones de creación, lectura, actualización y eliminación (CRUD) en un origen de datos.

Diseño de un microservicio CRUD sencillo

Desde un punto de vista de diseño, este tipo de microservicio en contenedor es muy sencillo. Quizás el problema para resolver es sencillo o la implementación es solo una prueba de concepto.

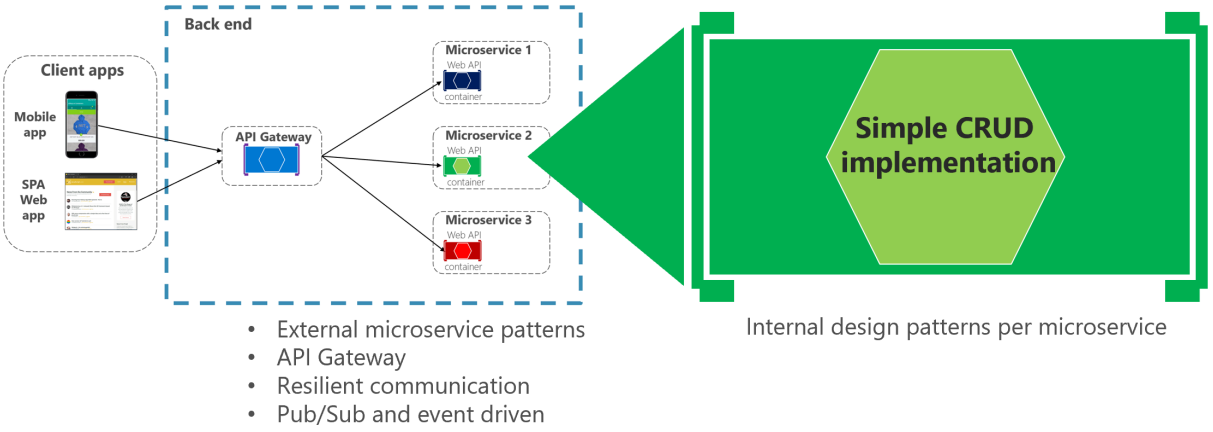


Figura 6-4. Diseño interno de microservicios CRUD sencillos

Un ejemplo de este tipo de servicio sencillo controlado por datos es el microservicio de catálogo de la aplicación de ejemplo eShopOnContainers. Este tipo de servicio implementa toda su funcionalidad en un solo proyecto de API Web de ASP.NET Core que incluye las clases para su modelo de datos, su lógica de negocios y su código de acceso a datos. También almacena los datos relacionados en una base de datos que ejecuta SQL Server (como otro contenedor para fines de desarrollo y pruebas), pero también podría ser cualquier host de SQL Server normal, como se muestra en la Figura 6-5.

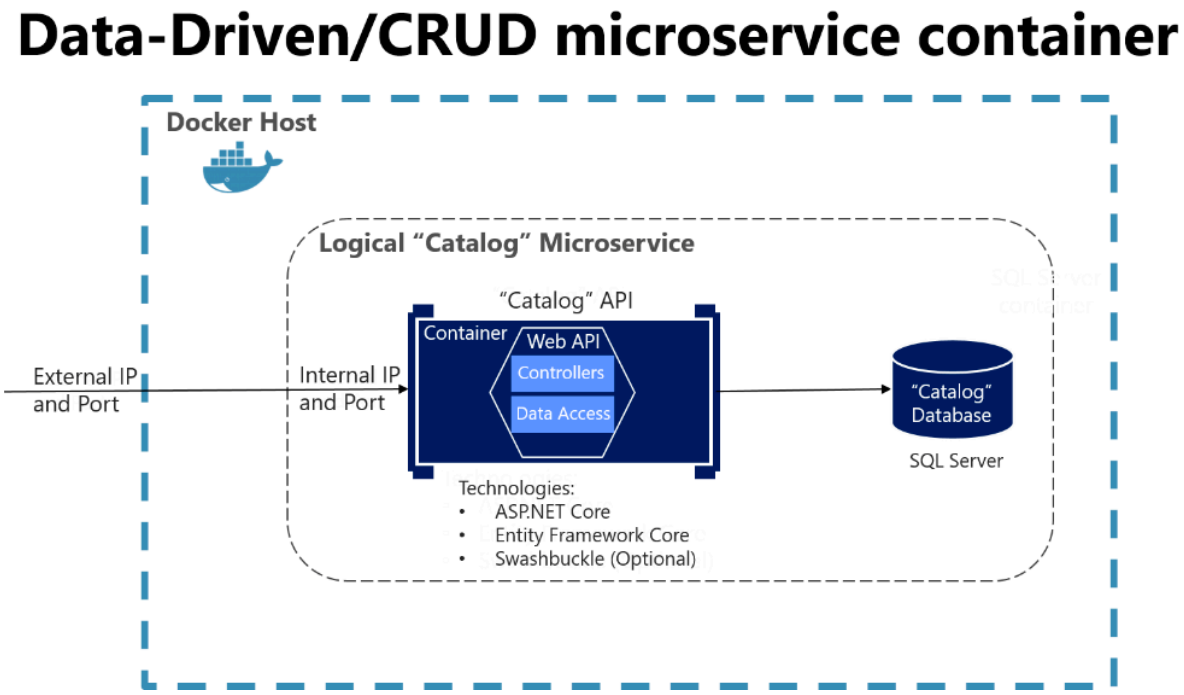


Figura 6-5. Diseño de un microservicio CRUD sencillo controlado por datos

En el diagrama anterior se muestra el microservicio lógico Catalog, que incluye su base de datos Catalog, que puede estar o no en el mismo host de Docker. Tener la base de datos en el mismo host de Docker podría ser bueno para el desarrollo, pero no para producción. Para desarrollar este tipo de servicio, solo necesita ASP.NET Core y una ORP

o API de acceso a datos, como Entity Framework Core. También puede generar automáticamente metadatos Swagger a través de Swashbuckle, para proporcionar una descripción de lo que ofrece el servicio, tal como se describe en la sección siguiente.

Tenga en cuenta que ejecutar un servidor de base de datos como SQL Server en un contenedor de Docker es muy útil para entornos de desarrollo, porque puede poner en marcha todas sus dependencias sin tener que proporcionar una base de datos local o en la nube. Este enfoque resulta útil al ejecutar pruebas de integración. Pero no se recomienda ejecutar un servidor de base de datos en un contenedor para entornos de producción, ya que normalmente no se obtiene alta disponibilidad con ese método. En un entorno de producción de Azure, le recomendamos que utilice la base de datos SQL de Azure o cualquier otra tecnología de base de datos que pueda proporcionar alta disponibilidad y alta escalabilidad. Por ejemplo, para un enfoque NoSQL, es posible que elija CosmosDB.

Por último, al editar los archivos de metadatos de Dockerfile y docker-compose.yml, puede configurar cómo se creará la imagen de este contenedor, es decir, la imagen base que se usará y la configuración de diseño, como los nombres internos y externos y los puertos TCP.

Implementación de un microservicio CRUD sencillo con ASP.NET Core

Para implementar un microservicio CRUD sencillo con .NET y Visual Studio, primero debe crear un proyecto de API web de ASP.NET Core sencillo (que se ejecute en .NET para que pueda ejecutarse en un host de Docker para Linux), como se muestra en la figura 6-6.

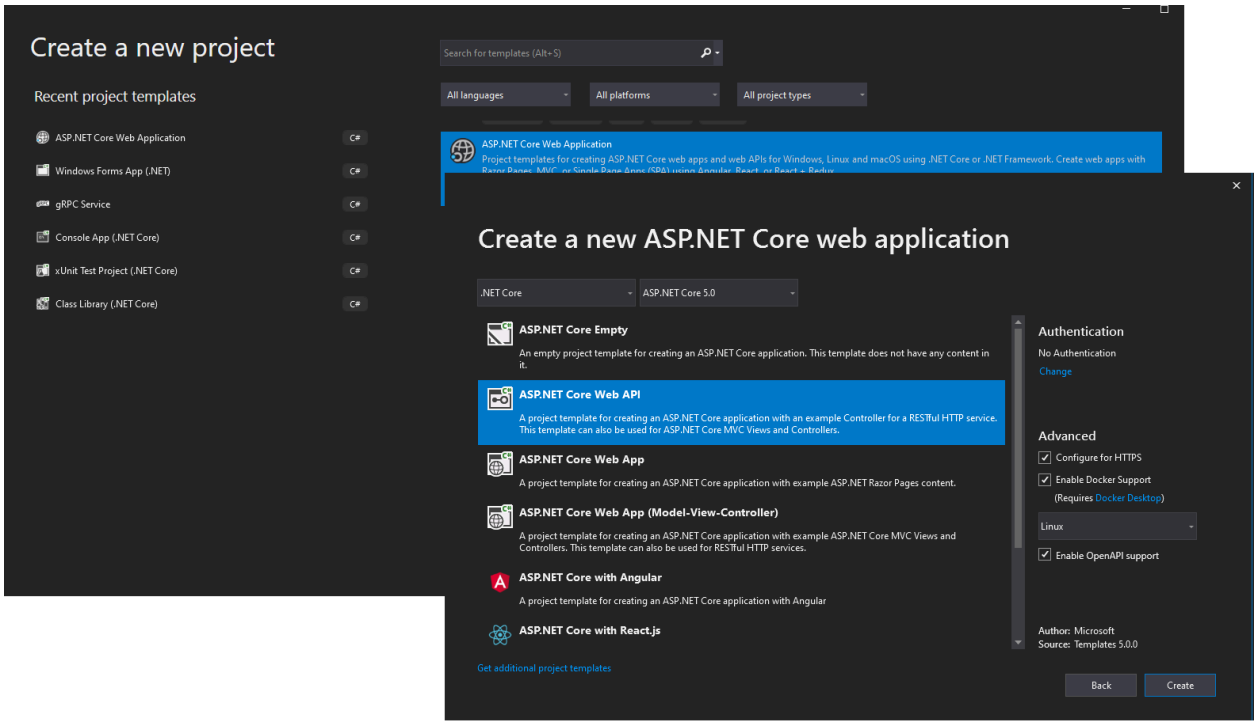


Figura 6-6. Creación de un proyecto de API web de ASP.NET Core en Visual Studio 2019

Para crear un proyecto de API web de ASP.NET Core, seleccione primero una aplicación web de ASP.NET Core y, después, seleccione el tipo de API. Después de crear el proyecto, puede implementar los controladores MVC como lo haría en cualquier otro proyecto de API Web, mediante la API de Entity Framework u otra API. En un nuevo proyecto de API Web, puede ver que la única dependencia que tiene de ese microservicio es el mismo ASP.NET Core. Internamente, dentro de la dependencia *Microsoft.AspNetCore.All*, hace referencia a Entity Framework y a muchos otros paquetes NuGet de .NET, como se muestra en la figura 6-7.

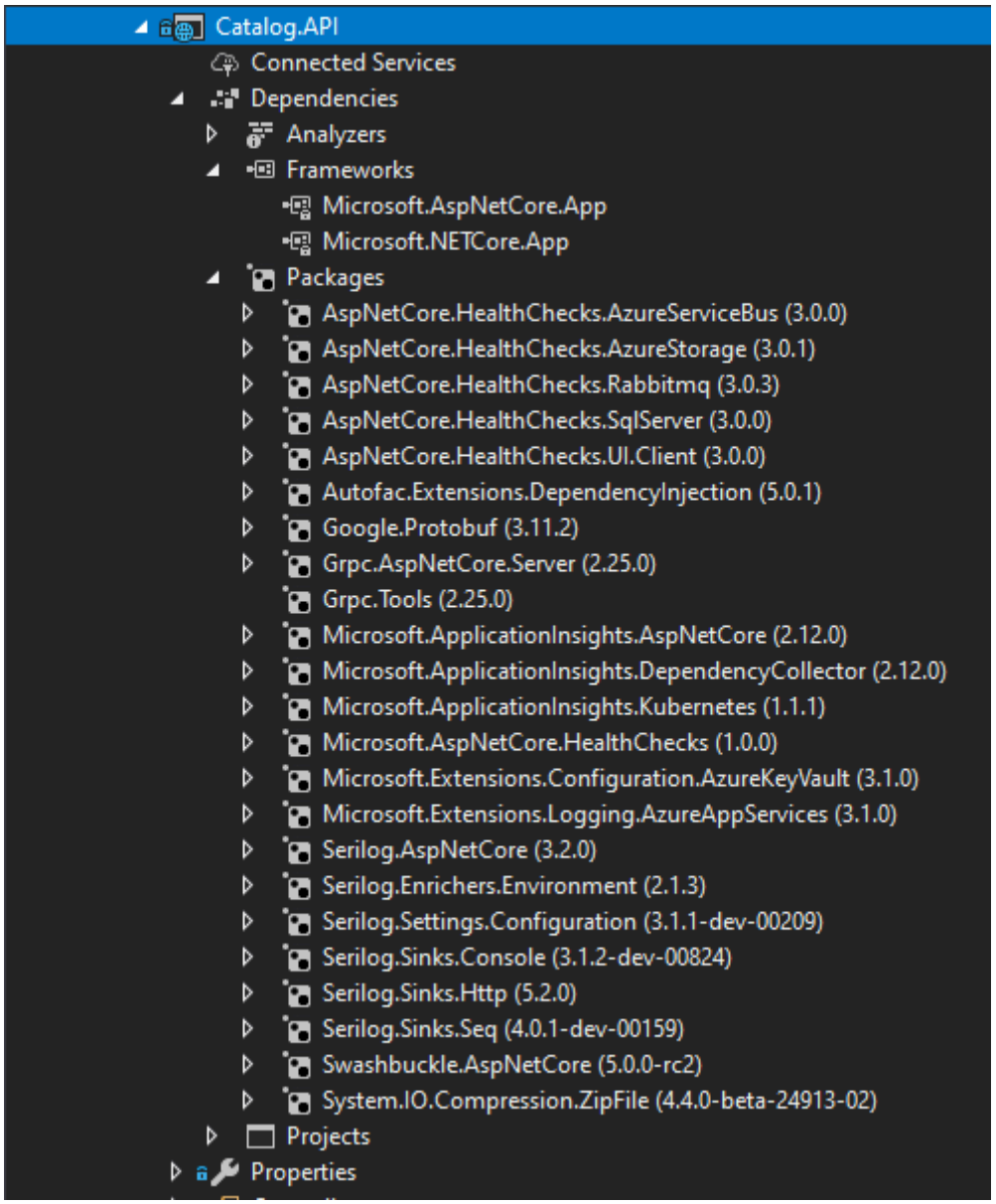


Figura 6-7. Dependencias en un microservicio API Web de CRUD sencillo

El proyecto de API incluye referencias al paquete NuGet Microsoft.AspNetCore.App, que a su vez incluye referencias a todos los paquetes esenciales. También podría incluir otros paquetes.

Implementación de servicios API Web de CRUD con Entity Framework Core

Entity Framework (EF) Core es una versión ligera, extensible y multiplataforma de la popular tecnología de acceso a datos Entity Framework. EF Core es un asignador relacional de objetos (ORM) que permite a los desarrolladores de .NET trabajar con una base de datos mediante objetos .NET.

El microservicio de catálogo usa EF y el proveedor de SQL Server porque su base de datos se está ejecutando en un contenedor con la imagen de SQL Server para Linux Docker. Pero la base de datos podría implementarse en cualquier SQL Server, como en

una base de datos SQL de Azure o Windows local. Lo único que debe cambiar es la cadena de conexión en el microservicio ASP.NET Web API.

El modelo de datos

Con EF Core, el acceso a datos se realiza utilizando un modelo. Un modelo se compone de clases de entidad (modelo de dominio) y un contexto derivado (DbContext) que representa una sesión con la base de datos, lo que permite consultar y guardar los datos. Puede generar un modelo a partir de una base de datos existente, codificar manualmente un modelo para que coincida con la base de datos, o bien usar técnicas de migración de EF para crear una base de datos a partir del modelo, mediante el enfoque Code First, que facilita que la base de datos evolucione a medida que el modelo cambia en el tiempo. Para el microservicio de catálogo, se ha utilizado el último enfoque. Puede ver un ejemplo de la clase de entidad CatalogItem en el siguiente ejemplo de código, que es una sencilla clase de entidad de objeto CLR estándar (POCO).

C#

```
public class CatalogItem
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public string PictureFileName { get; set; }
    public string PictureUri { get; set; }
    public int CatalogTypeId { get; set; }
    public CatalogType CatalogType { get; set; }
    public int CatalogBrandId { get; set; }
    public CatalogBrand CatalogBrand { get; set; }
    public int AvailableStock { get; set; }
    public int RestockThreshold { get; set; }
    public int MaxStockThreshold { get; set; }

    public bool OnReorder { get; set; }
    public CatalogItem() { }

    // Additional code ...
}
```

También necesita un DbContext que represente una sesión con la base de datos. Para el microservicio de catálogo, la clase CatalogContext se deriva de la clase base DbContext, tal como se muestra en el ejemplo siguiente:

C#

```
public class CatalogContext : DbContext
{
    public CatalogContext(DbContextOptions<CatalogContext> options) :
base(options)
    { }
    public DbSet<CatalogItem> CatalogItems { get; set; }
    public DbSet<CatalogBrand> CatalogBrands { get; set; }
    public DbSet<CatalogType> CatalogTypes { get; set; }

    // Additional code ...
}
```

Puede tener implementaciones `DbContext` adicionales. Por ejemplo, en el microservicio Catalog.API de ejemplo, hay un segundo `DbContext` denominado `CatalogContextSeed`, en que rellena automáticamente los datos de ejemplo la primera vez que intenta acceder a la base de datos. Este método es útil para los datos de demostración y también para escenarios de pruebas automatizadas.

En `DbContext`, se usa el método `OnModelCreating` para personalizar las asignaciones de entidades de objeto y base de datos, y otros puntos de extensibilidad de EF.

Consulta de los datos desde controladores de API web

Normalmente, las instancias de sus clases de entidad se recuperan de la base de datos mediante Language Integrated Query (LINQ), como se muestra en el ejemplo siguiente:

C#

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _catalogContext;
    private readonly CatalogSettings _settings;
    private readonly ICatalogIntegrationEventService
_catalogIntegrationEventService;

    public CatalogController(
        CatalogContext context,
        IOptionsSnapshot<CatalogSettings> settings,
        ICatalogIntegrationEventService catalogIntegrationEventService)
    {
        _catalogContext = context ?? throw new
ArgumentNullException(nameof(context));
        _catalogIntegrationEventService = catalogIntegrationEventService
            ?? throw new
ArgumentNullException(nameof(catalogIntegrationEventService));

        _settings = settings.Value;
```

```
        context.ChangeTracker.QueryTrackingBehavior =
QueryTrackingBehavior.NoTracking;
    }

    // GET api/v1/[controller]/items[?pageSize=3&pageIndex=10]
    [HttpGet]
    [Route("items")]
    [ProducesResponseType(typeof(PaginatedItemsViewModel<CatalogItem>),
(int)HttpStatusCode.OK)]
    [ProducesResponseType(typeof(IEnumerable<CatalogItem>),
(int)HttpStatusCode.OK)]
    [ProducesResponseType((int)HttpStatusCode.BadRequest)]
    public async Task<IActionResult> ItemsAsync(
        [FromQuery]int pageSize = 10,
        [FromQuery]int pageIndex = 0,
        string ids = null)
    {
        if (!string.IsNullOrEmpty(ids))
        {
            var items = await GetItemsByIdsAsync(ids);

            if (!items.Any())
            {
                return BadRequest("ids value invalid. Must be comma-
separated list of numbers");
            }

            return Ok(items);
        }

        var totalItems = await _catalogContext.CatalogItems
            .LongCountAsync();

        var itemsOnPage = await _catalogContext.CatalogItems
            .OrderBy(c => c.Name)
            .Skip(pageSize * pageIndex)
            .Take(pageSize)
            .ToListAsync();

        itemsOnPage = ChangeUriPlaceholder(itemsOnPage);

        var model = new PaginatedItemsViewModel<CatalogItem>(
            pageIndex, pageSize, totalItems, itemsOnPage);

        return Ok(model);
    }
    //...
}
```

Guardado de datos

Los datos se crean, se eliminan y se modifican en la base de datos mediante instancias de las clases de entidad. Puede agregar código similar al siguiente ejemplo codificado de forma rígida (datos simulados, en este caso) a sus controladores de la API web.

```C#
var catalogItem = new CatalogItem() {CatalogTypeId=2, CatalogBrandId=2,
                                     Name="Roslyn T-Shirt", Price = 12};
_context.Catalog.Add(catalogItem);
_context.SaveChanges();
```

Inserción de dependencias en los controladores de ASP.NET Core y API web

En ASP.NET Core, puede usar la inserción de dependencias desde el principio. No es necesario que configure un contenedor de inversión de control (IoC) de terceros, aunque, si lo desea, puede conectar su contenedor de IoC preferido a la infraestructura de ASP.NET Core. En este caso, puede insertar directamente el DBContext de EF requerido o los repositorios adicionales a través del constructor del controlador.

En la clase `CatalogController` mencionada anteriormente, el tipo `CatalogContext`, que se hereda de `DbContext`, se inserta junto con los demás objetos necesarios en el constructor `CatalogController()`.

Una opción importante que hay que configurar en el proyecto de Web API es el registro de la clase DbContext en el contenedor de IoC del servicio. Normalmente se hace en el archivo *Program.cs*, llamando al método `builder.Services.AddDbContext<CatalogContext>()`, tal como se muestra en el siguiente ejemplo **simplificado**:

```C#
// Additional code...

builder.Services.AddDbContext<CatalogContext>(options =>
{
    options.UseSqlServer(builder.Configuration["ConnectionString"],
    sqlServerOptionsAction: sqlOptions =>
    {
        sqlOptions.MigrationsAssembly(
            typeof(Program).GetTypeInfo().Assembly.GetName().Name);

        //Configuring Connection Resiliency:
        sqlOptions.
            EnableRetryOnFailure(maxRetryCount: 5,
            maxRetryDelay: TimeSpan.FromSeconds(30),
```

```
            errorNumbersToAdd: null);
    });

    // Changing default behavior when client evaluation occurs to throw.
    // Default in EFCore would be to log warning when client evaluation is
done.
    options.ConfigureWarnings(warnings => warnings.Throw(
        RelationalEventId.QueryClientEvaluationWarning));
});
```

> ⓘ **Importante**
>
> Microsoft recomienda usar el flujo de autenticación más seguro disponible. Si se conecta a Azure SQL, el método de autenticación recomendado es [Identidades administradas para recursos de Azure](#).

Recursos adicionales

- **Consulta de datos**
  https://learn.microsoft.com/ef/core/querying/index
- **Guardado de datos**
  https://learn.microsoft.com/ef/core/saving/index

Variables de entorno y cadena de conexión de la base de datos utilizadas por contenedores de Docker

Puede usar la configuración de ASP.NET Core y agregar una propiedad ConnectionString al archivo settings.json, tal como se muestra en el ejemplo siguiente:

JSON

```
{
    "ConnectionString": "Server=tcp:127.0.0.1,5433;Initial
Catalog=Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=
[PLACEHOLDER]",
    "ExternalCatalogBaseUrl": "http://host.docker.internal:5101",
    "Logging": {
        "IncludeScopes": false,
        "LogLevel": {
            "Default": "Debug",
            "System": "Information",
            "Microsoft": "Information"
```

```
        }
    }
}
```

El archivo settings.json puede tener valores predeterminados para la propiedad ConnectionString o para cualquier otra propiedad. Pero estas propiedades se reemplazarán por los valores de las variables de entorno que se especifican en el archivo docker-compose.override.yml, al usar Docker.

Desde los archivos docker-compose.yml o docker-compose.override.yml, puede inicializar estas variables de entorno para que Docker las configure como variables de entorno del sistema operativo, como se muestra en el siguiente archivo docker-compose.override.yml (la cadena de conexión y otras líneas se encapsulan en este ejemplo, pero no lo harán en su propio archivo).

yml

```yml
# docker-compose.override.yml

#
catalog-api:
  environment:
    -
ConnectionString=Server=sqldata;Database=Microsoft.eShopOnContainers.Service
s.CatalogDb;User Id=sa;Password=[PLACEHOLDER]
    # Additional environment variables for this service
  ports:
    - "5101:80"
```

> ⓘ **Importante**
>
> Microsoft recomienda usar el flujo de autenticación más seguro disponible. Si se conecta a Azure SQL, el método de autenticación recomendado es **Identidades administradas para recursos de Azure**.

Los archivos docker-compose.yml en el nivel de solución no solo son más flexibles que los archivos de configuración en el nivel de proyecto o de microservicio, sino que también son más seguros si reemplaza las variables de entorno declaradas en los archivos docker-compose con valores establecidos en las herramientas de implementación, como las tareas de implementación del Docker de Azure DevOps Services.

Por último, puede obtener ese valor del código mediante `builder.Configuration["ConnectionString"]`, tal como se muestra en un ejemplo de

código anterior.

Pero, en entornos de producción, puede ser que le interese analizar otras formas de almacenar secretos, como las cadenas de conexión. Una manera excelente de administrar los secretos de aplicación consiste en usar Azure Key Vault.

Azure Key Vault ayuda a almacenar y proteger las claves criptográficas y los secretos que usan la aplicaciones y los servicios en la nube. Un secreto es todo aquello sobre lo que quiera mantener un control estricto, como las claves de API, las cadenas de conexión, las contraseñas, etc. Asimismo, un control estricto incluye el registro del uso, el establecimiento de la caducidad y la administración del acceso, *entre otros aspectos*.

Azure Key Vault permite un nivel de control detallado del uso de secretos de la aplicación sin necesidad de que otras personas los conozcan. Incluso se puede definir que los secretos vayan rotando para mejorar la seguridad sin interrumpir las operaciones ni el desarrollo.

Es necesario registrar las aplicaciones en la instancia de Active Directory de la organización, de modo que puedan usar el almacén de claves.

Puede consultar la *documentación de conceptos de Key Vault* para obtener más detalles.

Implementación del control de versiones en las API web de ASP.NET

A medida que cambian los requisitos empresariales, pueden agregarse nuevas colecciones de recursos, las relaciones entre recursos pueden cambiar y la estructura de los datos en los recursos se puede modificar. Actualizar una API web para controlar requisitos nuevos es un proceso relativamente sencillo, pero debe tener en cuenta los efectos que estos cambios tendrán en las aplicaciones cliente que consumen la API web. Aunque el desarrollador que diseña e implementa una API web tiene control total sobre dicha API, no tiene el mismo grado de control sobre las aplicaciones cliente creadas por organizaciones de terceros que funcionan de forma remota.

El control de versiones permite que una API web indique las características y los recursos que expone. De este modo, una aplicación cliente puede enviar solicitudes a una versión específica de una característica o de un recurso. Existen varios enfoques para implementar el control de versiones:

- Control de versiones de URI
- Control de versiones de cadena de consulta
- Control de versiones de encabezado

El control de versiones de URI y de cadena de consulta son los más fáciles de implementar. El control de versiones de encabezado es una buena opción. Pero el control de versiones de encabezado no es tan explícito y sencillo como el control de versiones de URI. Como el control de versiones de URI es el más sencillo y explícito, es el que utiliza la aplicación de ejemplo eShopOnContainers.

Con el control de versiones de URI, como se muestra en la aplicación de ejemplo eShopOnContainers, cada vez que modifique la API web o cambie el esquema de recursos, agregará un número de versión al URI de cada recurso. Los URI existentes deben continuar funcionando como antes, devolviendo los recursos que conforman el esquema que coincide con la versión solicitada.

Como se muestra en el ejemplo de código siguiente, la versión se puede establecer mediante el atributo Route del controlador de la API web, lo que hace que la versión se explicite en el URI (v1 en este caso).

C#

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    // Implementation ...
```

Este mecanismo de control de versiones es sencillo y depende del servidor que enruta la solicitud al punto de conexión adecuado. Pero para utilizar un control de versiones más sofisticado y adoptar el mejor método al utilizar REST, debe usar hipermedia e implementar HATEOAS (hipertexto como motor del estado de la aplicación).

Recursos adicionales

- **Control de versiones de API de ASP.NET** \ https://github.com/dotnet/aspnet-api-versioning
- **Scott Hanselman. ASP.NET Core RESTful Web API versioning made easy (Control de versiones simplificado de API web RESTful de ASP.NET Core)** https://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx
- **Control de versiones de una API web RESTful** https://learn.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api

- **Roy Fielding. Versioning, Hypermedia, and REST (Control de versiones, hipermedia y REST)**
  https://www.infoq.com/articles/roy-fielding-on-versioning

Generación de metadatos de descripción de Swagger desde la API web de ASP.NET Core

Swagger es un marco de código abierto de uso común, respaldado por una gran variedad de herramientas que le permite diseñar, compilar, documentar y utilizar las API RESTful. Se está convirtiendo en el estándar para el dominio de metadatos de la descripción de API. Debe incluir los metadatos de descripción de Swagger con cualquier tipo de microservicio, tanto si está controlado por datos como si está controlado por dominios de forma más avanzada, tal como se explica en la sección siguiente.

El núcleo de Swagger es su especificación, que son los metadatos de descripción de la API en un archivo JSON o YAML. La especificación crea el contrato RESTful para la API, donde se detallan todos sus recursos y operaciones en formatos legibles por máquinas y por humanos, para que se puedan desarrollar, descubrir e integrar de forma sencilla.

La especificación es la base de la especificación OpenAPI (OAS) y se desarrolla en una comunidad abierta, transparente y colaborativa para estandarizar la forma en que se definen las interfaces RESTful.

La especificación define la estructura de descubrimiento de un servicio y la forma de entender sus capacidades. Para obtener más información, incluido un editor web y ejemplos de especificaciones de Swagger de empresas como Spotify, Uber, Slack y Microsoft, consulte el sitio web de Swagger (https://swagger.io).

¿Por qué usar Swagger?

Las razones principales para generar metadatos de Swagger para las API son las siguientes:

Capacidad de otros productos de utilizar e integrar las API automáticamente . Swagger es compatible con docenas de productos y herramientas comerciales, así como con muchas bibliotecas y marcos. Microsoft tiene productos y herramientas de alto nivel que pueden utilizar automáticamente API basadas en Swagger, como las siguientes:

- AutoRest. Puede generar automáticamente clases de cliente de .NET para llamar a Swagger. Esta herramienta se puede utilizar desde la interfaz de la línea de

comandos y también se integra con Visual Studio para que pueda utilizarse fácilmente desde la interfaz gráfica de usuario.

- Microsoft Flow. También puede utilizar e integrar la API automáticamente en un flujo de trabajo de Microsoft Flow de alto nivel, aunque no tenga conocimientos de programación.
- Microsoft PowerApps. Puede utilizar la API automáticamente desde aplicaciones móviles PowerApps creadas con PowerApps Studio, aunque no tenga conocimientos de programación.
- Azure App Service Logic Apps. También puede utilizar e integrar automáticamente su API en una Azure App Service Logic App, aunque no tenga conocimientos de programación.

Capacidad de generar documentación de la API automáticamente. Al crear API RESTful a gran escala, como aplicaciones complejas basadas en microservicios, tiene que controlar muchos de los puntos de conexión con diferentes modelos de datos diferentes que se utilizan en las cargas de solicitud y respuesta. Tener una documentación adecuada y un explorador de API potente, como se consigue con Swagger, es fundamental para que su API tenga éxito y los desarrolladores la adopten.

Microsoft Flow, PowerApps y Azure Logic Apps usan los metadatos de Swagger para aprender a usar las API y conectarse a ellas.

Hay varias opciones para automatizar la generación de metadatos de Swagger para las aplicaciones de API REST de ASP.NET Core, en forma de páginas de ayuda de API funcionales, basadas en *swagger-ui*.

Probablemente la más conocida sea Swashbuckle, que actualmente se usa en eShopOnContainers y que trataremos con más detalle en esta guía, pero también existe la opción de usar NSwag, que puede generar clientes de API de Typescript y C#, así como controladores de C#, a partir de una especificación de OpenAPI o Swagger, e incluso mediante el examen del archivo .dll que contiene los controladores, con NSwagStudio.

Cómo se automatiza la generación de metadatos de la API de Swagger con el paquete NuGet de Swashbuckle

Generar metadatos de Swagger manualmente (en un archivo JSON o YAML) puede resultar muy pesado. Pero puede automatizar la detección de API de servicios ASP.NET Web API mediante el uso del paquete NuGet de Swashbuckle para generar dinámicamente metadatos de la API de Swagger.

Swashbuckle genera automáticamente metadatos de Swagger para sus proyectos de ASP.NET Web API. Admite proyectos de ASP.NET Core Web API, proyectos tradicionales de ASP.NET Web API y cualquier otro tipo, como la aplicación API de Azure, la aplicación móvil de Azure o los microservicios Azure Service Fabric basados en ASP.NET. También admite API web sencillas implementadas en contenedores, como es el caso de la aplicación de referencia.

Swashbuckle combina el explorador de API y Swagger o swagger-ui para proporcionar una experiencia de detección y documentación increíble a los consumidores de la API. Además de su motor generador de metadatos de Swagger, Swashbuckle también contiene una versión insertada de swagger-ui, que se usará automáticamente cuando se haya instalado Swashbuckle.

Esto significa que puede complementar su API con una bonita interfaz de usuario de descubrimiento para ayudar a los desarrolladores a usar su API. Para ello se requiere una cantidad pequeña de código y mantenimiento, puesto que se genera automáticamente, lo que le permite centrarse en la creación de la API. El resultado para el explorador de API se parece a la Figura 6-8.

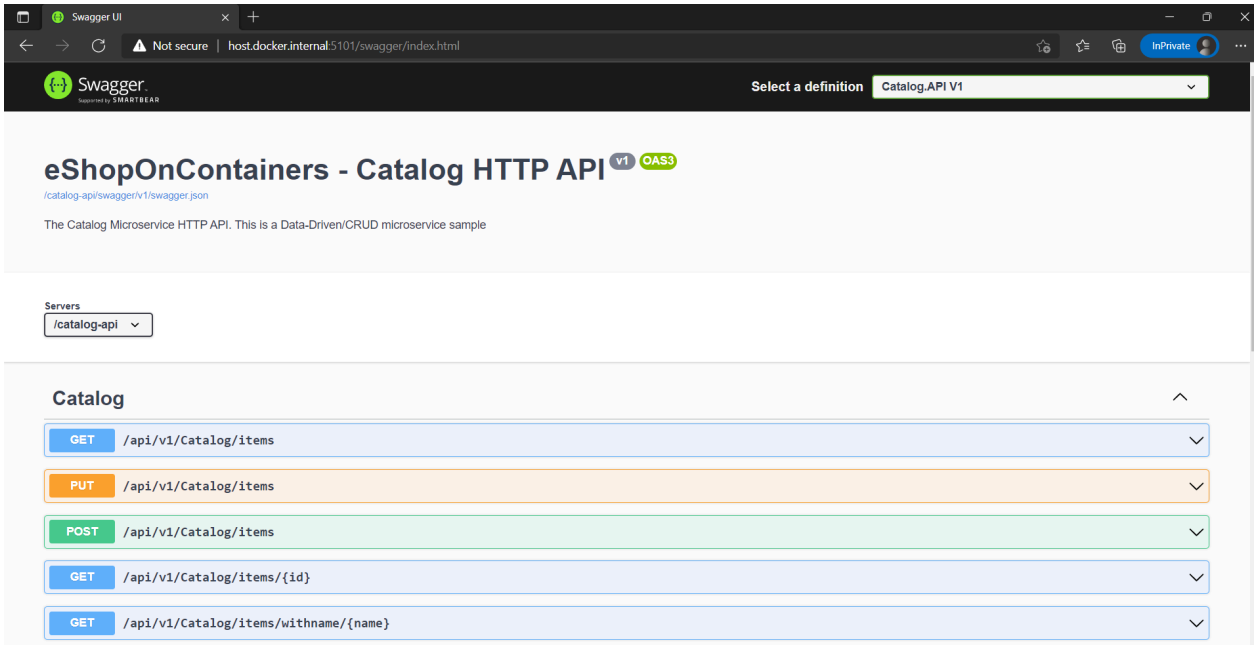


Figura 6-8. Explorador de API de Swashbuckle basado en metadatos de Swagger: microservicio del catálogo eShopOnContainers

La documentación de API de la interfaz de usuario de Swagger generada por Swashbuckle incluye todas las acciones publicadas. Pero aquí lo más importante no es el explorador de API. Cuando tenga una API web que se pueda describir en metadatos de Swagger, la API podrá usarse sin problemas desde herramientas basadas en Swagger, incluidos los generadores de código de clase proxy de cliente que pueden tener varias plataformas como destino. Por ejemplo, tal y como se ha mencionado, AutoRest genera automáticamente clases de cliente .NET. Pero también están disponibles

herramientas como swagger-codegen, que permiten que se genere automáticamente código de bibliotecas de cliente de API, códigos auxiliares de servidor y documentación.

En la actualidad, Swashbuckle consta de cinco paquetes NuGet internos que se engloban en el metapaquete general Swashbuckle.AspNetCore para las aplicaciones ASP.NET Core.

Después de instalar estos paquetes NuGet en el proyecto de Web API, debe configurar Swagger en la clase *Program.cs*, como en el siguiente código **simplificado**:

C#

```
// Add framework services.

builder.Services.AddSwaggerGen(options =>
{
    options.DescribeAllEnumsAsStrings();
    options.SwaggerDoc("v1", new OpenApiInfo
    {
        Title = "eShopOnContainers - Catalog HTTP API",
        Version = "v1",
        Description = "The Catalog Microservice HTTP API. This is a Data-
Driven/CRUD microservice sample"
    });
});

// Other startup code...

app.UseSwagger();

if (app.Environment.IsDevelopment())
{
    app.UseSwaggerUI(c =>
    {
        c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
    });
}
```

Una vez hecho esto, puede iniciar la aplicación y examinar los siguientes puntos de conexión JSON y de interfaz de usuario de Swagger utilizando direcciones URL como estas:

Consola

```
http://<your-root-url>/swagger/v1/swagger.json

http://<your-root-url>/swagger/
```

Anteriormente, vio la interfaz de usuario generada creada por Swashbuckle para una dirección URL como `http://<your-root-url>/swagger`. En la figura 6-9 también puede ver cómo se puede probar cualquier método de API.

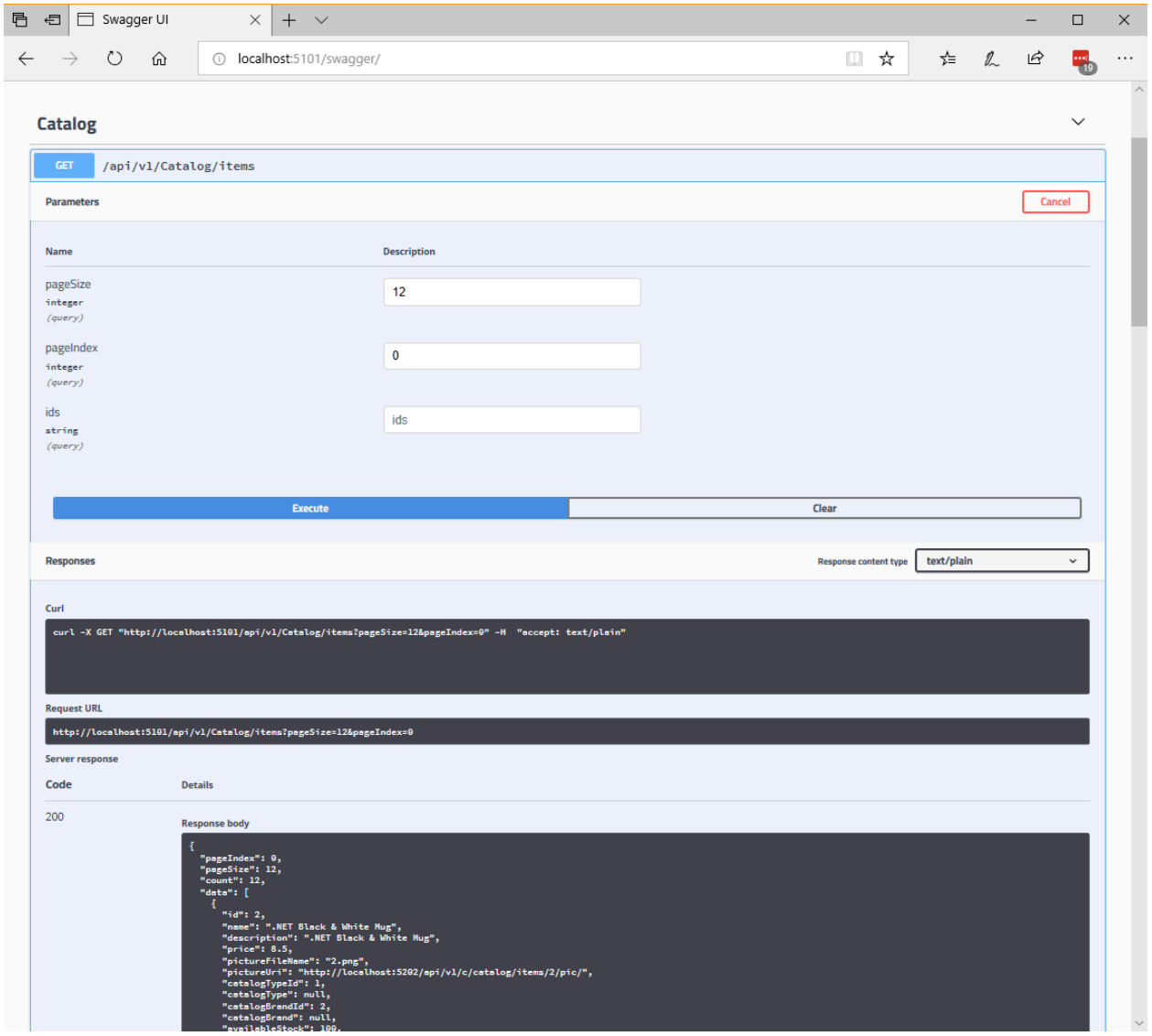


Figura 6-9. Interfaz de usuario de Swashbuckle poniendo a prueba el método de API de catálogo o elementos

En los detalles de la API de interfaz de usuario de Swagger se muestra un ejemplo de la respuesta y se puede usar para ejecutar la API real, que es muy útil para la detección por parte de los desarrolladores. Para ver los metadatos JSON de Swagger generados a partir del microservicio eShopOnContainers (que es lo que usan las herramientas debajo), realice una solicitud `http://<your-root-url>/swagger/v1/swagger.json` mediante la extensión cliente REST de Visual Studio Code.

Recursos adicionales

- **Páginas de ayuda de ASP.NET Core Web API con Swagger**
  https://learn.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-

swagger

- **Introducción a Swashbuckle y ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/tutorials/getting-started-with-swashbuckle

- **Introducción a NSwag y ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/tutorials/getting-started-with-nswag

Anterior Siguiente

Definir su aplicación de varios contenedores con docker-compose.yml

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En esta guía, el archivo docker-compose.yml se introdujo en la sección Paso 4. Defina los servicios en docker-compose.yml al compilar una aplicación de Docker de varios contenedores. Sin embargo, hay maneras adicionales de usar los archivos docker-compose que merecen la pena explorar con más detalle.

Por ejemplo, puede describir explícitamente cómo desea implementar la aplicación de varios contenedores en el archivo docker-compose.yml. Opcionalmente, también puede describir cómo va a compilar las imágenes personalizadas de Docker. (Las imágenes personalizadas de Docker también se pueden compilar con la CLI de Docker).

Básicamente, se define cada uno de los contenedores que desea implementar más determinadas características para cada implementación de contenedor. Una vez que tenga un archivo de descripción de implementación de varios contenedores, puede implementar toda la solución en una sola acción orquestada por el comando de la CLI docker-compose up , o bien puede implementarla de forma transparente desde Visual Studio. De lo contrario, tendría que usar la CLI de Docker para implementar contenedor por contenedor en varios pasos mediante el comando `docker run` desde la línea de comandos. Por lo tanto, cada servicio definido en docker-compose.yml debe especificar exactamente una imagen o compilación.

Otras claves son opcionales y son análogas a sus `docker run` homólogos de línea de comandos.

El siguiente código YAML es la definición de un archivo global pero único docker-compose.yml posible para el ejemplo eShopOnContainers. Este código no es el archivo docker-compose real de eShopOnContainers. En su lugar, es una versión simplificada y consolidada en un solo archivo, que no es la mejor manera de trabajar con archivos docker-compose, como se explicará más adelante.

yml

```yml
version: '3.4'

services:
  webmvc:
    image: eshop/webmvc
    environment:
      - CatalogUrl=http://catalog-api
      - OrderingUrl=http://ordering-api
      - BasketUrl=http://basket-api
    ports:
      - "5100:80"
    depends_on:
      - catalog-api
      - ordering-api
      - basket-api

  catalog-api:
    image: eshop/catalog-api
    environment:
      - ConnectionString=Server=sqldata;Initial Catalog=CatalogData;User
Id=sa;Password=[PLACEHOLDER]
    expose:
      - "80"
    ports:
      - "5101:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
    depends_on:
      - sqldata

  ordering-api:
    image: eshop/ordering-api
    environment:
      - ConnectionString=Server=sqldata;Database=Services.OrderingDb;User
Id=sa;Password=[PLACEHOLDER]
    ports:
      - "5102:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
```

```
    depends_on:
      - sqldata

  basket-api:
    image: eshop/basket-api
    environment:
      - ConnectionString=sqldata
    ports:
      - "5103:80"
    depends_on:
      - sqldata

  sqldata:
    environment:
      - SA_PASSWORD=[PLACEHOLDER]
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"

  basketdata:
    image: redis
```

La clave raíz de este archivo es services. En esa clave, se definen los servicios que desea implementar y ejecutar al ejecutar el `docker-compose up` comando o al implementar desde Visual Studio mediante este archivo docker-compose.yml. En este caso, el archivo docker-compose.yml tiene varios servicios definidos, como se describe en la tabla siguiente.

Expandir tabla

Nombre del servicio	Descripción
webmvc	Contenedor que incluye la aplicación ASP.NET Core MVC que consume los microservicios del lado servidor de C#
catalog-api	Contenedor que incluye el microservicio Catalog ASP.NET Core Web API
ordering-api	Contenedor que incluye el microservicio Ordering ASP.NET Core Web API
sqldata	Contenedor que ejecuta SQL Server para Linux, que contiene las bases de datos de microservicios
basket-api	Contenedor con el microservicio Basket ASP.NET Core Web API
basketdata	Contenedor que ejecuta el servicio de caché de REDIS, con la base de datos de cesta como caché de REDIS

Un contenedor de API de servicio web simple

Si nos centramos en un único contenedor, el microservicio de contenedor catalog-api tiene una definición sencilla:

```yml
  catalog-api:
    image: eshop/catalog-api
    environment:
      - ConnectionString=Server=sqldata;Initial Catalog=CatalogData;User
Id=sa;Password=[PLACEHOLDER]
    expose:
      - "80"
    ports:
      - "5101:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
    depends_on:
      - sqldata
```

Este servicio en contenedor tiene la siguiente configuración básica:

- Se basa en la imagen **personalizada de eshop/catalog-api** . Por simplicidad, no hay ninguna compilación: configuración de clave en el archivo. Esto significa que la imagen debe haberse compilado previamente (con la compilación de Docker) o se ha descargado (con el comando docker pull) desde cualquier registro de Docker.

- Define una variable de entorno denominada ConnectionString con la cadena de conexión que Entity Framework usará para acceder a la instancia de SQL Server que contiene el modelo de datos de catálogo. En este caso, el mismo contenedor de SQL Server contiene varias bases de datos. Por lo tanto, necesita menos memoria en la máquina de desarrollo para Docker. Sin embargo, también puede implementar un contenedor de SQL Server para cada base de datos de microservicios.

- El nombre de SQL Server es **sqldata**, que es el mismo nombre que se usa para el contenedor que ejecuta la instancia de SQL Server para Linux. Esto es conveniente; poder usar esta resolución de nombres (interna para el host de Docker) resolverá la dirección de red, por lo que no es necesario conocer la dirección IP interna de los contenedores a los que accede desde otros contenedores.

Dado que la cadena de conexión se define mediante una variable de entorno, puede establecer esa variable a través de un mecanismo diferente y en otro momento. Por ejemplo, puede establecer una cadena de conexión diferente al implementar en producción en los hosts finales o establecerla desde las canalizaciones de CI/CD en Azure DevOps Services o en su sistema de DevOps de su preferencia.

- Expone el puerto 80 para el acceso interno al servicio **catalog-api** dentro del host de Docker. El host es actualmente una máquina virtual Linux porque se basa en una imagen de Docker para Linux, pero puede configurar el contenedor para que se ejecute en una imagen de Windows en su lugar.
- Reenvía el puerto expuesto 80 en el contenedor al puerto 5101 en la máquina host de Docker (la máquina virtual Linux).
- Vincula el servicio web al servicio **sqldata** (la instancia de SQL Server para la base de datos linux que se ejecuta en un contenedor). Al especificar esta dependencia, el contenedor catalog-api no se iniciará hasta que el contenedor sqldata ya se haya iniciado; Este aspecto es importante porque catalog-api debe tener primero en funcionamiento la base de datos de SQL Server. Sin embargo, este tipo de dependencia de contenedor no es suficiente en muchos casos, ya que Docker solo comprueba en el nivel de contenedor. A veces, es posible que el servicio (en este caso, SQL Server) todavía no esté listo, por lo que es aconsejable implementar la lógica de reintento con retroceso exponencial en los microservicios de cliente. De este modo, si un contenedor de dependencias no está listo durante un breve tiempo, la aplicación seguirá siendo resistente.
- Está configurado para permitir el acceso a servidores externos: la configuración de extra_hosts permite acceder a servidores externos o máquinas fuera del host de Docker (es decir, fuera de la máquina virtual Linux predeterminada, que es un host de Docker de desarrollo), como una instancia local de SQL Server en el equipo de desarrollo.

También hay otras configuraciones más avanzadas `docker-compose.yml` que analizaremos en las secciones siguientes.

Uso de archivos docker-compose para tener como destino varios entornos

Los `docker-compose.*.yml` archivos son archivos de definición y se pueden usar en varias infraestructuras que entienden ese formato. La herramienta más sencilla es el comando docker-compose.

Por lo tanto, mediante el comando docker-compose puede tener como destino los siguientes escenarios principales.

Entornos de desarrollo

Al desarrollar aplicaciones, es importante poder ejecutar una aplicación en un entorno de desarrollo aislado. Puede usar el comando de la CLI de docker-compose para crear ese entorno o Visual Studio, que usa docker-compose en segundo plano.

El archivo docker-compose.yml permite configurar y documentar todas las dependencias de servicio de la aplicación (otros servicios, caché, bases de datos, colas, etc.). Con el comando de la CLI docker-compose, puede crear e iniciar uno o varios contenedores para cada dependencia con un solo comando (docker-compose up).

Los archivos docker-compose.yml son archivos de configuración interpretados por el motor de Docker, pero también sirven como archivos de documentación convenientes sobre la composición de la aplicación de varios contenedores.

Entornos de prueba

Una parte importante de cualquier proceso de implementación continua (CD) o integración continua (CI) son las pruebas unitarias y las pruebas de integración. Estas pruebas automatizadas requieren un entorno aislado para que no se vean afectados por los usuarios ni por ningún otro cambio en los datos de la aplicación.

Con Docker Compose puede crear y destruir ese entorno aislado de un modo muy sencillo ejecutando unos scripts o comandos en el símbolo del sistema, como los comandos siguientes:

Consola

```
docker-compose -f docker-compose.yml -f docker-compose-test.override.yml up -d
./run_unit_tests
docker-compose -f docker-compose.yml -f docker-compose-test.override.yml down
```

Implementaciones de producción

También puede usar Compose para implementar en un motor remoto de Docker. Un caso típico es implementar en una única instancia de host de Docker.

Si usa cualquier otro orquestador (por ejemplo, Azure Service Fabric o Kubernetes), es posible que tenga que agregar opciones de configuración de configuración y metadatos como las de docker-compose.yml, pero en el formato requerido por el otro orquestador.

En cualquier caso, docker-compose es un formato práctico de herramientas y metadatos para los flujos de trabajo de desarrollo, pruebas y producción, aunque el flujo de trabajo de producción puede variar en el orquestador que está usando.

Uso de varios archivos docker-compose para controlar varios entornos

Al fijar como objetivo entornos diferentes, debe usar varios archivos compose. Este enfoque le permite crear varias variantes de configuración en función del entorno.

Invalidar el archivo base docker-compose

Puede usar un solo archivo docker-compose.yml como en los ejemplos simplificados que se muestran en las secciones anteriores. Sin embargo, no se recomienda para la mayoría de las aplicaciones.

De forma predeterminada, Compose lee dos archivos, un docker-compose.yml y un archivo de docker-compose.override.yml opcional. Como se muestra en la figura 6-11, al usar Visual Studio y habilitar la compatibilidad con Docker, Visual Studio también crea un archivo docker-compose.vs.debug.g.yml adicional para depurar la aplicación, puede echar un vistazo a este archivo en la carpeta obj\Docker\ en la carpeta de la solución principal.

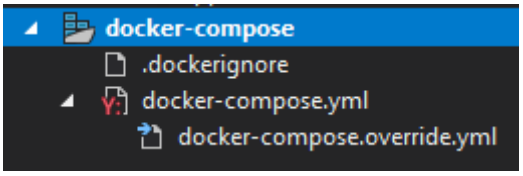


Figura 6-11. archivos "docker-compose" en Visual Studio 2019

Estructura del archivo de proyecto **docker-compose**:

- *.dockerignore* : se usa para pasar por alto los archivos
- *docker-compose.yml* : se usa para componer microservicios
- *docker-compose.override.yml* : se usa para configurar el entorno de microservicios

Puede editar los archivos docker-compose con cualquier editor, como Visual Studio Code o Sublime, y ejecutar la aplicación con el comando docker-compose up.

Por convención, el archivo docker-compose.yml contiene la configuración base y otras opciones estáticas. Esto significa que la configuración del servicio no debe cambiar en función del entorno de implementación que tenga como destino.

El archivo docker-compose.override.yml, como sugiere su nombre, contiene opciones de configuración que invalidan la configuración base, como la configuración que depende del entorno de implementación. También puede tener varios archivos de invalidación con nombres diferentes. Los archivos de sobrescritura suelen contener información adicional necesaria para la aplicación, pero que es específica de un entorno o de una implementación.

Fijar como objetivo varios entornos

Un caso de uso típico es cuando se definen varios archivos de composición para que pueda apuntar a varios entornos, como producción, preproducción, integración continua o desarrollo. Para respaldar estas diferencias, puede dividir la configuración de Compose en varios archivos, como se muestra en la figura 6-12.

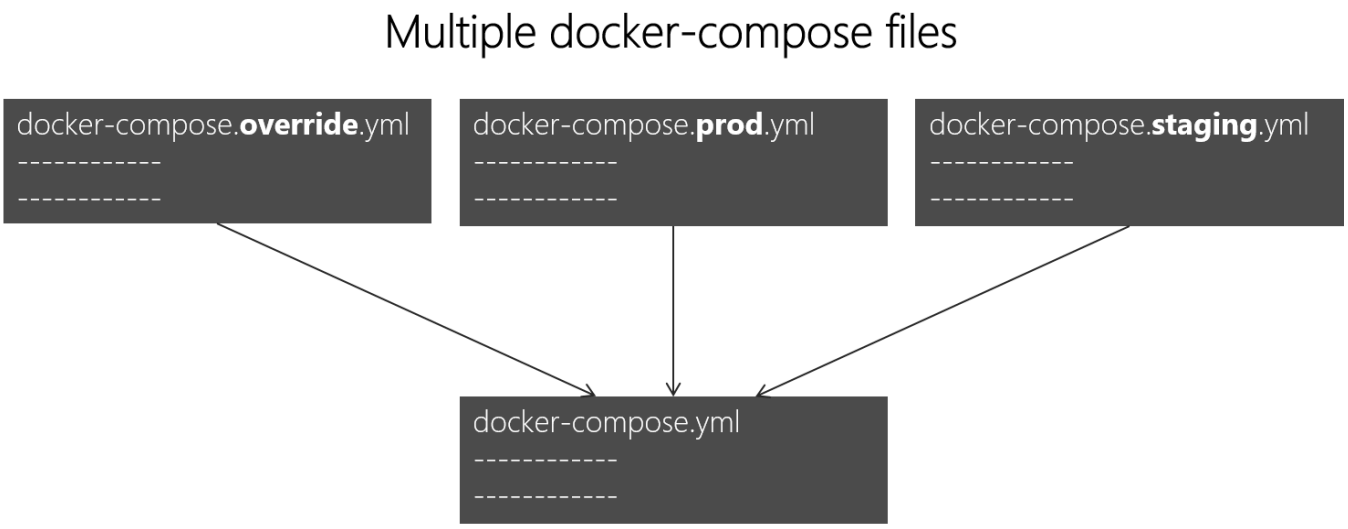


Figura 6-12. Varios archivos docker-compose reemplazan valores en el archivo base docker-compose.yml

Puede combinar varios archivos docker-compose*.yml para controlar distintos entornos. Comienza con el archivo docker-compose.yml base. Este archivo base contiene las opciones de configuración base o estática que no cambian en función del entorno. Por ejemplo, la aplicación eShopOnContainers tiene el siguiente archivo docker-compose.yml (simplificado con menos servicios) como archivo base.

yml

```yml
#docker-compose.yml (Base)
version: '3.4'
services:
  basket-api:
    image: eshop/basket-api:${TAG:-latest}
    build:
      context: .
      dockerfile: src/Services/Basket/Basket.API/Dockerfile
    depends_on:
      - basketdata
      - identity-api
      - rabbitmq

  catalog-api:
    image: eshop/catalog-api:${TAG:-latest}
    build:
      context: .
      dockerfile: src/Services/Catalog/Catalog.API/Dockerfile
    depends_on:
```

```
      - sqldata
      - rabbitmq

  marketing-api:
    image: eshop/marketing-api:${TAG:-latest}
    build:
      context: .
      dockerfile: src/Services/Marketing/Marketing.API/Dockerfile
    depends_on:
      - sqldata
      - nosqldata
      - identity-api
      - rabbitmq

  webmvc:
    image: eshop/webmvc:${TAG:-latest}
    build:
      context: .
      dockerfile: src/Web/WebMVC/Dockerfile
    depends_on:
      - catalog-api
      - ordering-api
      - identity-api
      - basket-api
      - marketing-api

  sqldata:
    image: mcr.microsoft.com/mssql/server:2019-latest

  nosqldata:
    image: mongo

  basketdata:
    image: redis

  rabbitmq:
    image: rabbitmq:3-management
```

Los valores del archivo docker-compose.yml base no deben cambiar debido a distintos entornos de implementación de destino.

Si se centra en la definición del servicio webmvc, por ejemplo, puede ver cómo esa información es mucho igual independientemente del entorno al que se dirija. Tiene la siguiente información:

- Nombre del servicio: webmvc.
- Imagen personalizada del contenedor: eshop/webmvc.
- Comando para compilar la imagen personalizada de Docker, que indica qué Dockerfile se va a usar.

- Dependencias de otros servicios, por lo que este contenedor no se inicia hasta que se hayan iniciado los otros contenedores de servicios dependientes.

Puede tener configuración adicional, pero el punto importante es que en el archivo base docker-compose.yml, solo quiere establecer la información que es común entre entornos. A continuación, en el archivo docker-compose.override.yml o archivos similares para producción o preproducción, debe colocar la configuración específica para cada entorno.

Normalmente, el docker-compose.override.yml se usa para el entorno de desarrollo, como en el ejemplo siguiente de eShopOnContainers:

yml

```yml
#docker-compose.override.yml (Extended config for DEVELOPMENT env.)
version: '3.4'

services:
# Simplified number of services here:

  basket-api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_REDIS_BASKET_DB:-basketdata}
      - identityUrl=http://identity-api
      - IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - AzureServiceBusEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
      - UseLoadTest=${USE_LOADTEST:-False}

    ports:
      - "5103:80"

  catalog-api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_CATALOG_DB:-
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.CatalogDb;User
Id=sa;Password=[PLACEHOLDER]}
      - PicBaseUrl=${ESHOP_AZURE_STORAGE_CATALOG_URL:-
http://host.docker.internal:5202/api/v1/catalog/items/[0]/pic/}
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - AzureStorageAccountName=${ESHOP_AZURE_STORAGE_CATALOG_NAME}
      - AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_CATALOG_KEY}
      - UseCustomizationData=True
```

```
      - AzureServiceBusEnabled=False
      - AzureStorageEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
    ports:
      - "5101:80"

  marketing-api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_MARKETING_DB:-
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.MarketingDb;User
Id=sa;Password=[PLACEHOLDER]}
      - MongoConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}
      - MongoDatabase=MarketingDb
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - identityUrl=http://identity-api
      - IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
      - CampaignDetailFunctionUri=${ESHOP_AZUREFUNC_CAMPAIGN_DETAILS_URI}
      - PicBaseUrl=${ESHOP_AZURE_STORAGE_MARKETING_URL:-
http://host.docker.internal:5110/api/v1/campaigns/[0]/pic/}
      - AzureStorageAccountName=${ESHOP_AZURE_STORAGE_MARKETING_NAME}
      - AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_MARKETING_KEY}
      - AzureServiceBusEnabled=False
      - AzureStorageEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
      - UseLoadTest=${USE_LOADTEST:-False}
    ports:
      - "5110:80"

  webmvc:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - PurchaseUrl=http://webshoppingapigw
      - IdentityUrl=http://10.0.75.1:5105
      - MarketingUrl=http://webmarketingapigw
      - CatalogUrlHC=http://catalog-api/hc
      - OrderingUrlHC=http://ordering-api/hc
      - IdentityUrlHC=http://identity-api/hc
      - BasketUrlHC=http://basket-api/hc
      - MarketingUrlHC=http://marketing-api/hc
      - PaymentUrlHC=http://payment-api/hc
      - SignalrHubUrl=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5202
      - UseCustomizationData=True
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
      - UseLoadTest=${USE_LOADTEST:-False}
    ports:
      - "5100:80"
  sqldata:
```

```
    environment:
      - SA_PASSWORD=[PLACEHOLDER]
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
  nosqldata:
    ports:
      - "27017:27017"
  basketdata:
    ports:
      - "6379:6379"
  rabbitmq:
    ports:
      - "15672:15672"
      - "5672:5672"
```

En este ejemplo, la configuración de invalidación de desarrollo expone algunos puertos al host, define variables de entorno con direcciones URL de redireccionamiento y especifica cadenas de conexión para el entorno de desarrollo. Esta configuración es solo para el entorno de desarrollo.

Cuando se ejecuta `docker-compose up` (o se inicia desde Visual Studio), el comando lee automáticamente las sobreescrituras como si ambos archivos se combinaran.

Supongamos que desea otro archivo compose para el entorno de producción, con valores de configuración, puertos o cadenas de conexión diferentes. Puede crear otro archivo de anulación, como el archivo llamado `docker-compose.prod.yml` con diferentes configuraciones y variables de entorno. Ese archivo puede almacenarse en un repositorio de Git diferente o administrado y protegido por un equipo diferente.

Cómo efectuar una implementación con un archivo de invalidación específico

Para usar varios archivos de invalidación o un archivo de invalidación con un nombre diferente, puede usar la opción -f con el comando docker-compose y especificar los archivos. Compose combina archivos en el orden en que se especifican en la línea de comandos. En el ejemplo siguiente se muestra cómo efectuar la implementación con archivos de invalidación.

Consola

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

Uso de variables de entorno en archivos docker-compose

Es conveniente, especialmente en entornos de producción, poder obtener información de configuración de variables de entorno, como se ha mostrado en los ejemplos anteriores. Puede hacer referencia a una variable de entorno en los archivos docker-compose mediante la sintaxis ${MY_VAR}. La siguiente línea de un archivo docker-compose.prod.yml muestra cómo hacer referencia al valor de una variable de entorno.

```yml
IdentityUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5105
```

Las variables de entorno se crean e inicializan de diferentes maneras, en función del entorno host (Linux, Windows, clúster en la nube, etc.). Sin embargo, un enfoque práctico es usar un archivo .env. Los archivos docker-compose admiten la declaración de variables de entorno predeterminadas en el archivo .env. Estos valores para las variables de entorno son los valores predeterminados. Sin embargo, los valores que podría haber definido en cada uno de los entornos, en el sistema operativo host o en las variables de entorno del clúster, pueden invalidar los valores predeterminados. Este archivo .env se coloca en la carpeta desde la que se ejecuta el comando docker-compose.

En el ejemplo siguiente se muestra un archivo .env como el archivo .env para la aplicación eShopOnContainers.

```sh
# .env file

ESHOP_EXTERNAL_DNS_NAME_OR_IP=host.docker.internal

ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=10.121.122.92
```

Docker-compose espera que cada línea de un archivo .env esté en el formato <variable>=<value>.

Los valores establecidos en el entorno en tiempo de ejecución siempre invalidan los valores definidos dentro del archivo .env. De forma similar, los valores pasados a través de argumentos de línea de comandos también invalidan los valores predeterminados establecidos en el archivo .env.

Recursos adicionales

- **Introducción a Docker Compose**
  https://docs.docker.com/compose/overview/

- **Varios archivos de Compose**
  https://docs.docker.com/compose/multiple-compose-files/

Creación de imágenes de Docker de ASP.NET Core optimizadas

Si está explorando Docker y .NET en fuentes en Internet, encontrará Dockerfiles que muestran la simplicidad de crear una imagen de Docker copiando tu código fuente en un contenedor. Estos ejemplos sugieren que, mediante una configuración sencilla, puede tener una imagen de Docker con el entorno empaquetado con la aplicación. En el ejemplo siguiente se muestra un Dockerfile simple de este tipo.

Dockerfile

```
FROM mcr.microsoft.com/dotnet/sdk:8.0
WORKDIR /app
ENV ASPNETCORE_URLS http://+:80
EXPOSE 80
COPY . .
RUN dotnet restore
ENTRYPOINT ["dotnet", "run"]
```

Un Dockerfile como este funcionará. Sin embargo, puede optimizar sustancialmente las imágenes, especialmente las imágenes de producción.

En el modelo de microservicios y contenedores están iniciando contenedores constantemente. La forma típica de usar contenedores no reinicia un contenedor en reposo, ya que el contenedor es desechable. Los orquestadores (como Kubernetes y Azure Service Fabric) crean nuevas instancias de imágenes. Esto significa que necesitarás optimizar al precompilar la aplicación en su construcción, así que el proceso de creación de instancias será más rápido. Cuando se inicia el contenedor, tendría que estar preparado para ejecutarse. No restaure ni compile durante la ejecución mediante los comandos `dotnet restore` y `dotnet build` de la CLI, como en las entradas de blog sobre .NET y Docker.

El equipo de .NET ha estado realizando un trabajo importante para que .NET y ASP.NET Core sea un marco optimizado para contenedores. No solo es .NET un marco ligero con una superficie de memoria pequeña; el equipo se ha centrado en las imágenes de Docker optimizadas para tres escenarios principales y los ha publicado en el registro de Docker Hub en *dotnet/*, a partir de la versión 2.1:

- **Desarrollo**: la prioridad es la capacidad de iterar y depurar rápidamente los cambios, y donde el tamaño es secundario.

- **Compilación**: la prioridad es compilar la aplicación y la imagen incluye archivos binarios y otras dependencias para optimizar los archivos binarios.
- **Producción**: el enfoque es la implementación rápida y el inicio de contenedores, por lo que estas imágenes se limitan a los archivos binarios y el contenido necesario para ejecutar la aplicación.

El equipo de .NET proporciona algunas variantes básicas en dotnet/, por ejemplo:

- **sdk**: para escenarios de desarrollo y compilación
- **aspnet**: para escenarios de producción de ASP.NET
- **runtime**: para escenarios de producción de .NET
- **runtime-deps**: para escenarios de producción de aplicaciones independientes

Para un inicio más rápido, las imágenes en tiempo de ejecución también configuran automáticamente las direcciones aspnetcore_url en el puerto 80 y usan Ngen para crear una caché de imágenes nativa de ensamblados.

Recursos adicionales

- **Creación de imágenes de Docker optimizadas con ASP.NET Core**https://learn.microsoft.com/archive/blogs/stevelasker/building-optimized-docker-images-with-asp-net-core

- **Compilación de imágenes de Docker para aplicaciones .NET**https://learn.microsoft.com/dotnet/core/docker/building-net-docker-images

Siguiente anterior

Uso de un servidor de bases de datos que se ejecuta como contenedor

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Puede tener las bases de datos (SQL Server, PostgreSQL, MySQL, etc.) en servidores independientes normales, en clústeres locales o en servicios PaaS en la nube, como Azure SQL DB. Sin embargo, para entornos de desarrollo y pruebas, tener las bases de datos que se ejecutan como contenedores es conveniente, ya que no tiene ninguna dependencia externa y simplemente ejecutar el `docker-compose up` comando inicia toda la aplicación. Tener esas bases de datos como contenedores también es excelente para las pruebas de integración, ya que la base de datos se inicia en el contenedor y siempre se rellena con los mismos datos de ejemplo, por lo que las pruebas pueden ser más predecibles.

SQL Server que se ejecuta como un contenedor con una base de datos relacionada con microservicios

En eShopOnContainers, hay un contenedor denominado `sqldata`, tal como se define en el archivo docker-compose.yml , que ejecuta una instancia de SQL Server para Linux con las bases de datos SQL para todos los microservicios que necesitan uno.

Un punto clave en los microservicios es que cada microservicio posee sus datos relacionados, por lo que debe tener su propia base de datos. Sin embargo, las bases de datos pueden estar en cualquier lugar. En este caso, todos están en el mismo contenedor para mantener los requisitos de memoria de Docker lo más bajo posible. Tenga en cuenta que se trata de una solución suficiente para el desarrollo y, quizás, probar pero no para producción.

El contenedor de SQL Server de la aplicación de ejemplo se configura con el siguiente código YAML en el archivo docker-compose.yml, que se ejecuta al ejecutar `docker-compose up`. Tenga en cuenta que el código YAML tiene información de configuración consolidada del archivo genérico docker-compose.yml y del archivo docker-compose.override.yml. (Normalmente, separaría la configuración del entorno de la información base o estática relacionada con la imagen de SQL Server).

yml

```yml
  sqldata:
    image: mcr.microsoft.com/mssql/server:2017-latest
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"
```

De forma similar, en lugar de usar `docker-compose`, el siguiente `docker run` comando puede ejecutar ese contenedor:

PowerShell

```PowerShell
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=Pass@word' -p 5433:1433 -d
mcr.microsoft.com/mssql/server:2017-latest
```

Sin embargo, si va a implementar una aplicación de varios contenedores como eShopOnContainers, es más conveniente usar el `docker-compose up` comando para que implemente todos los contenedores necesarios para la aplicación.

Al iniciar este contenedor de SQL Server por primera vez, el contenedor inicializa SQL Server con la contraseña que proporcione. Una vez que SQL Server se ejecuta como un contenedor, puede actualizar la base de datos mediante la conexión a través de cualquier conexión SQL normal, como desde código de SQL Server Management Studio, Visual Studio o C#.

La aplicación eShopOnContainers inicializa cada base de datos de microservicios con datos de ejemplo al inicializarlos con datos en el inicio, como se explica en la sección siguiente.

Tener SQL Server ejecutándose como contenedor no solo es útil para una demostración en la que es posible que no tenga acceso a una instancia de SQL Server. Como se indicó, también es ideal para entornos de desarrollo y pruebas para que pueda ejecutar fácilmente pruebas de integración a partir de una imagen limpia de SQL Server y datos conocidos mediante la propagación de nuevos datos de ejemplo.

Recursos adicionales

- **Ejecución de la imagen de Docker de SQL Server en Linux, Mac o Windows**
  https://learn.microsoft.com/sql/linux/sql-server-linux-setup-docker
- **Conexión y consulta de SQL Server en Linux con sqlcmd**
  https://learn.microsoft.com/sql/linux/sql-server-linux-connect-and-query-sqlcmd

Propagación con datos de prueba al iniciar la aplicación web

Para agregar datos a la base de datos cuando se inicia la aplicación, puede agregar código como el siguiente al `Main` método de la `Program` clase del proyecto de API web:

C#

```
public static int Main(string[] args)
{
    var configuration = GetConfiguration();

    Log.Logger = CreateSerilogLogger(configuration);

    try
    {
        Log.Information("Configuring web host ({ApplicationContext})...",
AppName);
        var host = CreateHostBuilder(configuration, args);

        Log.Information("Applying migrations ({ApplicationContext})...", AppName);
        host.MigrateDbContext<CatalogContext>((context, services) =>
        {
            var env = services.GetService<IWebHostEnvironment>();
            var settings = services.GetService<IOptions<CatalogSettings>>();
            var logger = services.GetService<ILogger<CatalogContextSeed>>();

            new CatalogContextSeed()
                .SeedAsync(context, env, settings, logger)
                .Wait();
        })
        .MigrateDbContext<IntegrationEventLogContext>((_, __) => { });
```

```
        Log.Information("Starting web host ({ApplicationContext})...", AppName);
        host.Run();

        return 0;
    }
    catch (Exception ex)
    {
        Log.Fatal(ex, "Program terminated unexpectedly ({ApplicationContext})!",
AppName);
        return 1;
    }
    finally
    {
        Log.CloseAndFlush();
    }
}
```

Hay una advertencia importante al aplicar migraciones y inicializar una base de datos durante el inicio del contenedor. Dado que es posible que el servidor de bases de datos no esté disponible por cualquier motivo, debe manejar los reintentos mientras espera a que el servidor esté disponible. La lógica de reintento está gestionada por el método de extensión `MigrateDbContext()`, como se muestra en el código siguiente:

C#

```
public static IWebHost MigrateDbContext<TContext>(
    this IWebHost host,
    Action<TContext,
    IServiceProvider> seeder)
      where TContext : DbContext
{
    var underK8s = host.IsInKubernetes();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;

        var logger = services.GetRequiredService<ILogger<TContext>>();

        var context = services.GetService<TContext>();

        try
        {
            logger.LogInformation("Migrating database associated with context
{DbContextName}", typeof(TContext).Name);

            if (underK8s)
            {
                InvokeSeeder(seeder, context, services);
            }
            else
            {
```

```
                var retry = Policy.Handle<SqlException>()
                    .WaitAndRetry(new TimeSpan[]
                    {
                    TimeSpan.FromSeconds(3),
                    TimeSpan.FromSeconds(5),
                    TimeSpan.FromSeconds(8),
                    });

                //if the sql server container is not created on run docker compose
this
                //migration can't fail for network related exception. The retry
options for DbContext only
                //apply to transient exceptions
                // Note that this is NOT applied when running some orchestrators
(let the orchestrator to recreate the failing service)
                retry.Execute(() => InvokeSeeder(seeder, context, services));
            }

            logger.LogInformation("Migrated database associated with context
{DbContextName}", typeof(TContext).Name);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "An error occurred while migrating the database
used on context {DbContextName}", typeof(TContext).Name);
            if (underK8s)
            {
                throw;          // Rethrow under k8s because we rely on k8s to re-
run the pod
            }
        }
    }

    return host;
}
```

El código siguiente de la clase CatalogContextSeed personalizada rellena los datos.

C#

```
public class CatalogContextSeed
{
    public static async Task SeedAsync(IApplicationBuilder applicationBuilder)
    {
        var context = (CatalogContext)applicationBuilder
            .ApplicationServices.GetService(typeof(CatalogContext));
        using (context)
        {
            context.Database.Migrate();
            if (!context.CatalogBrands.Any())
            {
                context.CatalogBrands.AddRange(
                    GetPreconfiguredCatalogBrands());
```

```
                await context.SaveChangesAsync();
            }
            if (!context.CatalogTypes.Any())
            {
                context.CatalogTypes.AddRange(
                    GetPreconfiguredCatalogTypes());
                await context.SaveChangesAsync();
            }
        }
    }

    static IEnumerable<CatalogBrand> GetPreconfiguredCatalogBrands()
    {
        return new List<CatalogBrand>()
       {
           new CatalogBrand() { Brand = "Azure"},
           new CatalogBrand() { Brand = ".NET" },
           new CatalogBrand() { Brand = "Visual Studio" },
           new CatalogBrand() { Brand = "SQL Server" }
       };
    }

    static IEnumerable<CatalogType> GetPreconfiguredCatalogTypes()
    {
        return new List<CatalogType>()
        {
            new CatalogType() { Type = "Mug"},
            new CatalogType() { Type = "T-Shirt" },
            new CatalogType() { Type = "Backpack" },
            new CatalogType() { Type = "USB Memory Stick" }
        };
    }
}
```

Al ejecutar pruebas de integración, resulta útil tener una manera de generar datos coherentes con las pruebas de integración. Poder crear todo desde cero, incluida una instancia de SQL Server que se ejecuta en un contenedor, es excelente para entornos de prueba.

Base de datos InMemory de EF Core frente a SQL Server que se ejecuta como un contenedor

Otra buena opción al ejecutar pruebas es usar el proveedor de bases de datos InMemory de Entity Framework. Puede especificar esa configuración en el método ConfigureServices de la clase Startup en el proyecto de API web:

C#

```
public class Startup
{
```

```
    // Other Startup code ...
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton<IConfiguration>(Configuration);
        // DbContext using an InMemory database provider
        services.AddDbContext<CatalogContext>(opt => opt.UseInMemoryDatabase());
        //(Alternative: DbContext using a SQL Server provider
        //services.AddDbContext<CatalogContext>(c =>
        //{
            // c.UseSqlServer(Configuration["ConnectionString"]);
            //
        //});
    }

    // Other Startup code ...
}
```

Sin embargo, hay un truco importante. La base de datos en memoria no admite muchas restricciones específicas de una base de datos determinada. Por ejemplo, puede agregar un índice único en una columna del modelo de EF Core y escribir una prueba en la base de datos en memoria para comprobar que no permite agregar un valor duplicado. Pero cuando se usa la base de datos en memoria, no se pueden controlar índices únicos en una columna. Por lo tanto, la base de datos en memoria no se comporta exactamente igual que una base de datos real de SQL Server; no emula las restricciones específicas de la base de datos.

Incluso así, una base de datos en memoria sigue siendo útil para probar y crear prototipos. Pero si desea crear pruebas de integración precisas que tengan en cuenta el comportamiento de una implementación de base de datos específica, debe usar una base de datos real como SQL Server. Para ello, ejecutar SQL Server en un contenedor es una opción excelente y más precisa que el proveedor de bases de datos InMemory de EF Core.

Uso de un servicio de caché de Redis que se ejecuta en un contenedor

Puede ejecutar Redis en un contenedor, especialmente para desarrollo y pruebas y para escenarios de prueba de concepto. Este escenario es conveniente, ya que puede tener todas sus dependencias ejecutándose en contenedores, no solo para sus máquinas de desarrollo local, sino también para sus entornos de prueba en los tubos de CI/CD.

Sin embargo, al ejecutar Redis en producción, es mejor buscar una solución de alta disponibilidad como Redis Microsoft Azure, que se ejecuta como paaS (plataforma como servicio). En tu código, solo tienes que cambiar las cadenas de conexión.

Redis proporciona una imagen de Docker con Redis. Esa imagen está disponible en Docker Hub en esta dirección URL:

https://hub.docker.com/_/redis/

Puede ejecutar directamente un contenerizador de Docker Redis ejecutando el siguiente comando de la CLI de Docker en el sistema de comandos:

Consola

```
docker run --name some-redis -d redis
```

La imagen de Redis incluye expose:6379 (el puerto usado por Redis), por lo que la vinculación de contenedores estándar hará que esté disponible automáticamente para los contenedores vinculados.

En eShopOnContainers, el `basket-api` microservicio usa una caché de Redis que se ejecuta como un contenedor. Ese `basketdata` contenedor se define como parte del archivo *docker-compose.yml de varios* contenedores, como se muestra en el ejemplo siguiente:

yml

```yml
#docker-compose.yml file
#...
  basketdata:
    image: redis
    expose:
      - "6379"
```

Este código del docker-compose.yml define un contenedor denominado `basketdata` en función de la imagen de redis y publica internamente el puerto 6379. Esta configuración significa que solo será accesible desde otros contenedores que se ejecutan en el host de Docker.

Por último, en el archivo *docker-compose.override.yml* , el `basket-api` microservicio del ejemplo eShopOnContainers define la cadena de conexión que se va a usar para ese contenedor de Redis:

yml

```yml
  basket-api:
    environment:
      # Other data ...
      - ConnectionString=basketdata
      - EventBusConnection=rabbitmq
```

Como se mencionó antes, el nombre del microservicio `basketdata` se resuelve mediante el DNS de red interna de Docker.

Implementación de la comunicación basada en eventos entre microservicios (eventos de integración)

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Como se ha descrito anteriormente, cuando se usa la comunicación basada en eventos, un microservicio publica un evento cuando ocurre algo notable, como cuando actualiza una entidad empresarial. Otros microservicios se suscriben a esos eventos. Cuando un microservicio recibe un evento, puede actualizar sus propias entidades empresariales, lo que podría dar lugar a que se publiquen más eventos. Esta es la esencia del concepto de coherencia final. Este sistema de publicación o suscripción normalmente se realiza mediante una implementación de un bus de eventos. El bus de eventos se puede diseñar como una interfaz con la API necesaria para suscribirse y cancelar la suscripción a eventos y publicar eventos. También puede tener una o varias implementaciones basadas en cualquier comunicación entre procesos o mensajería, como una cola de mensajería o un bus de servicio que admita la comunicación asincrónica y un modelo de publicación o suscripción.

Puede usar eventos para implementar transacciones empresariales que abarquen varios servicios, lo que proporciona coherencia final entre esos servicios. Una transacción coherente eventualmente consta de una serie de acciones distribuidas. En cada acción, el microservicio actualiza una entidad empresarial y publica un evento que desencadena la siguiente acción. Tenga en cuenta que la transacción no abarca la persistencia subyacente ni el bus de eventos,

por lo que es necesario controlar la idempotencia. La figura 6-18 siguiente muestra un evento PriceUpdated publicado a través de un bus de eventos, por lo que la actualización de precios se propaga a la Cesta y a otros microservicios.

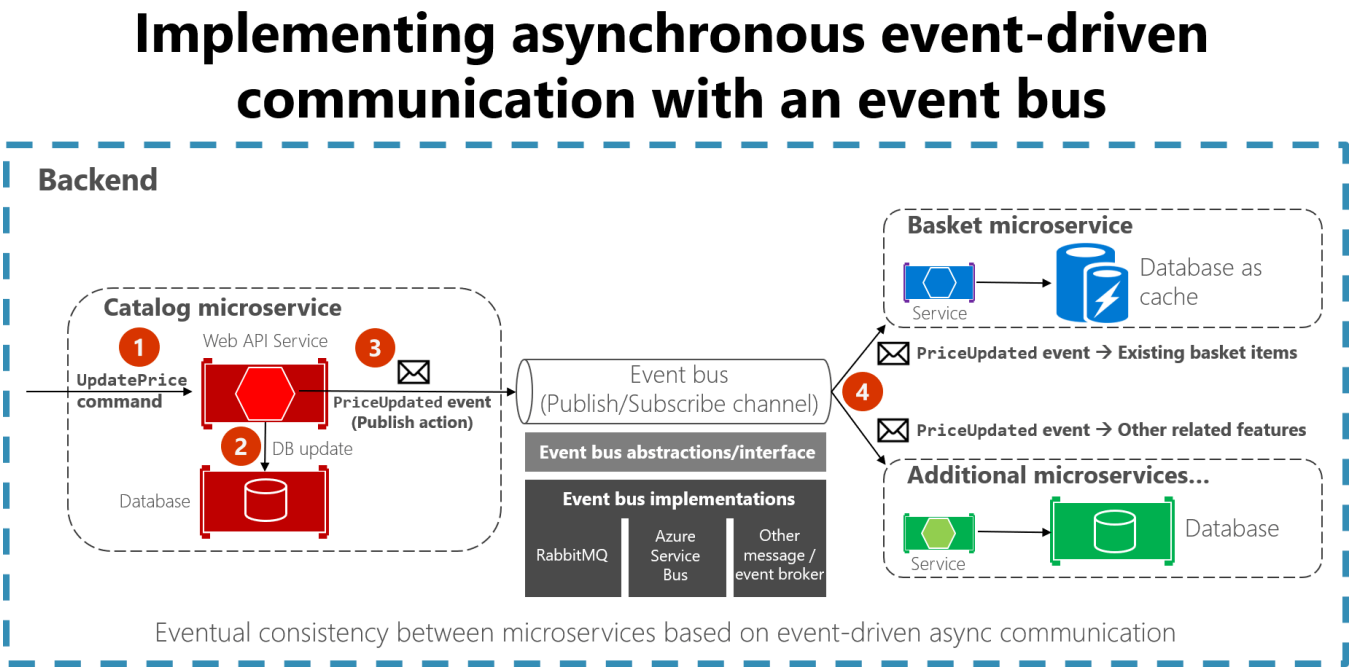


Figura 6-18. Comunicación impulsada por eventos basada en un bus de eventos.

En esta sección se describe cómo puede implementar este tipo de comunicación con .NET mediante una interfaz genérica de bus de eventos, como se muestra en la figura 6-18. Hay varias implementaciones potenciales, cada una con una tecnología o infraestructura diferente, como RabbitMQ, Azure Service Bus o cualquier otro bus de servicio comercial o de código abierto de terceros.

Uso de agentes de mensajes y buses de servicio para sistemas de producción

Como se indica en la sección de arquitectura, puede elegir entre varias tecnologías de mensajería para implementar el bus de eventos abstracto. Pero estas tecnologías están en diferentes niveles. Por ejemplo, RabbitMQ, un transporte de agentes de mensajería, está en un nivel inferior al de productos comerciales como Azure Service Bus, NServiceBus, MassTransit o Brighter. La mayoría de estos productos pueden funcionar sobre RabbitMQ o Azure Service Bus. La elección del producto depende de cuántas características y cuánta escalabilidad instantánea necesita para su aplicación.

Para implementar solo una prueba de concepto del bus de eventos para su entorno de desarrollo, como en el ejemplo eShopOnContainers, una implementación sencilla sobre RabbitMQ que se ejecuta como un contenedor podría ser suficiente. Pero para los sistemas

críticos y de producción que necesitan una alta escalabilidad, es posible que quiera evaluar y usar Azure Service Bus.

Si necesita abstracciones de alto nivel y características más enriquecidas como Sagas para procesos de larga duración que facilitan el desarrollo distribuido, otros buses de servicio comerciales y de código abierto como NServiceBus, MassTransit y Brighter merecen la pena evaluar. En este caso, las abstracciones y la API que se van a utilizar suelen ser las proporcionadas directamente por los buses de servicio de alto nivel, en vez de las propias abstracciones (como las abstracciones de bus de eventos sencillos proporcionadas en eShopOnContainers). Con este propósito, puede analizar eShopOnContainers bifurcado con NServiceBus (ejemplo derivado adicional implementado por Particular Software).

Por supuesto, siempre podría crear sus propias características de Service Bus sobre tecnologías de nivel inferior como RabbitMQ y Docker, pero el trabajo necesario para "reinventar la rueda" podría ser demasiado costoso para una aplicación empresarial personalizada.

Para repetir: las abstracciones de bus de eventos de ejemplo y la implementación que se muestran en el ejemplo eShopOnContainers están diseñadas para usarse solo como prueba de concepto. Una vez que haya decidido que desea tener comunicación asincrónica y controlada por eventos, como se explica en la sección actual, debe elegir el producto de Service Bus que mejor se adapte a sus necesidades de producción.

Eventos de integración

Los eventos de integración se usan para sincronizar el estado del dominio en varios microservicios o sistemas externos. Esta funcionalidad se realiza mediante la publicación de eventos de integración fuera del microservicio. Cuando se publica un evento en varios microservicios receptores (a tantos microservicios como se suscriben al evento de integración), el controlador de eventos adecuado en cada microservicio receptor controla el evento.

Un evento de integración es básicamente una clase de almacenamiento de datos, como en el ejemplo siguiente:

C#

```
public class ProductPriceChangedIntegrationEvent : IntegrationEvent
{
    public int ProductId { get; private set; }
    public decimal NewPrice { get; private set; }
    public decimal OldPrice { get; private set; }

    public ProductPriceChangedIntegrationEvent(int productId, decimal newPrice,
        decimal oldPrice)
    {
        ProductId = productId;
```

```
        NewPrice = newPrice;
        OldPrice = oldPrice;
    }
}
```

Los eventos de integración se pueden definir en el nivel de aplicación de cada microservicio, por lo que se desacoplan de otros microservicios, de forma comparable a cómo se definen ViewModels en el servidor y el cliente. Lo que no se recomienda es compartir una biblioteca de eventos de integración común en varios microservicios; hacerlo sería acoplar esos microservicios con una sola biblioteca de datos de definición de eventos. No desea hacerlo por las mismas razones por las que no desea compartir un modelo de dominio común en varios microservicios: los microservicios deben ser completamente autónomos. Para obtener más información, consulte esta entrada de blog sobre la cantidad de datos que se van a colocar en eventos. Tenga cuidado de no llevar esto demasiado lejos, ya que en esta otra entrada de blog se describen los mensajes deficientes de datos problemáticos que pueden producir. Su diseño de los eventos debe tener como objetivo ser "justo" para las necesidades de sus consumidores.

Solo hay algunos tipos de bibliotecas que debe compartir entre microservicios. Una es las bibliotecas que son bloques de aplicación finales, como la API de cliente de Event Bus, como en eShopOnContainers. Otra es las bibliotecas que constituyen herramientas que también se pueden compartir como componentes nuGet, como serializadores JSON.

El bus de eventos

Un bus de eventos permite la comunicación estilo publicación/suscripción entre microservicios sin necesidad de que los componentes sean conscientes explícitamente entre sí, como se muestra en la figura 6-19.

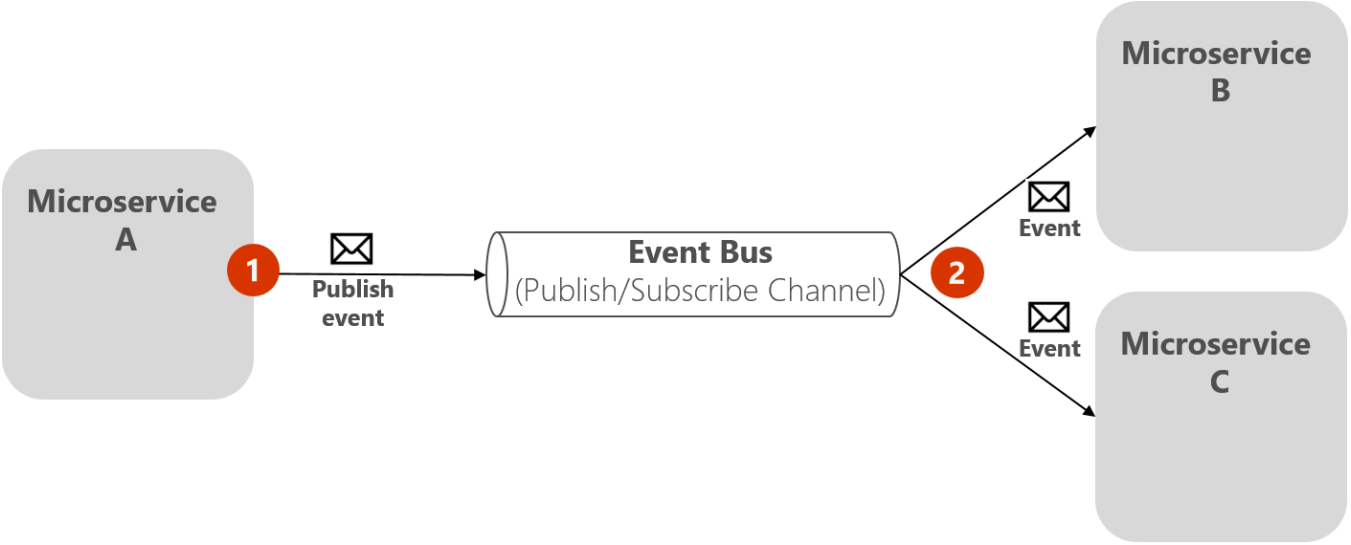


Figura 6-19. Aspectos básicos de publicación/suscripción con un bus de eventos

En el diagrama anterior se muestra que el microservicio A se publica en Event Bus, que se distribuye para suscribir los microservicios B y C, sin que el publicador necesite conocer los suscriptores. El bus de eventos está relacionado con el patrón Observer y el patrón publish-subscribe.

Patrón de observador

En el patrón Observer, el objeto principal (conocido como Observable) notifica a otros objetos interesados (conocidos como observadores) con información relevante (eventos).

Patrón de publicación/suscripción (Pub/Sus)

El propósito del patrón Publish/Subscribe es el mismo que el patrón Observer: quiere notificar a otros servicios cuando se produzcan determinados eventos. Pero hay una diferencia importante entre los patrones Observer y Pub/Sub. En el patrón de observador, la difusión se realiza directamente desde el observable a los observadores, por lo que se "conocen" entre sí. Sin embargo, cuando se usa un patrón Pub/Sub, hay un tercer componente, denominado agente, agente de mensajes o bus de eventos, que el publicador y el suscriptor conocen. Por lo tanto, cuando se usa el patrón Pub/Sub, el publicador y los suscriptores se desacoplan con precisión gracias al bus de eventos o al agente de mensajes mencionado.

El middleman o el bus de eventos

¿Cómo se logra el anonimato entre el publicador y el suscriptor? Una manera fácil es dejar que un intermediario se ocupe de toda la comunicación. Un bus de eventos es uno de esos intermediarios.

Normalmente, un bus de eventos se compone de dos partes:

- Abstracción o interfaz.
- Una o varias implementaciones.

En la figura 6-19 puede ver cómo, desde un punto de vista de la aplicación, el bus de eventos no es más que un canal Pub/Sub. La forma de implementar esta comunicación asincrónica puede variar. Puede tener varias implementaciones para que pueda intercambiar entre ellas, en función de los requisitos del entorno (por ejemplo, entornos de producción frente a entornos de desarrollo).

En la figura 6-20, puede ver una abstracción de un bus de eventos con varias implementaciones basadas en tecnologías de mensajería de infraestructura como RabbitMQ, Azure Service Bus u otro agente de eventos o mensajes.

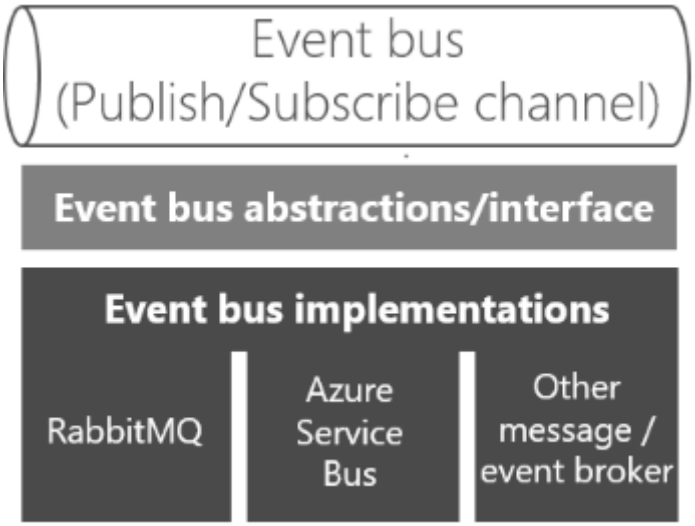


Figura 6- 20. Varias implementaciones de un bus de eventos

Es bueno tener definido el bus de eventos a través de una interfaz para que se pueda implementar con varias tecnologías, como RabbitMQ, Azure Service Bus u otros. Sin embargo, y como se mencionó anteriormente, el uso de sus propias abstracciones (la interfaz de bus de eventos) solo es bueno si necesita características básicas del bus de eventos compatibles con las abstracciones. Si necesita características más enriquecidas de Service Bus, probablemente debe usar la API y las abstracciones proporcionadas por su bus de servicio comercial preferido en lugar de sus propias abstracciones.

Definición de una interfaz de bus de eventos

Comencemos con algún código de implementación para la interfaz de bus de eventos y posibles implementaciones con fines de exploración. La interfaz debe ser genérica y sencilla, como en la siguiente interfaz.

C#

```
public interface IEventBus
{
    void Publish(IntegrationEvent @event);

    void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIntegrationEventHandler<T>;

    void SubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void UnsubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void Unsubscribe<T, TH>()
        where TH : IIntegrationEventHandler<T>
```

```
        where T : IntegrationEvent;
}
```

El `Publish` método es sencillo. El bus de eventos difundirá el evento de integración pasado a cualquier microservicio, o incluso a una aplicación externa, suscrito a ese evento. El microservicio que publica el evento usa este método.

Los `Subscribe` métodos (puede tener varias implementaciones en función de los argumentos) las usan los microservicios que desean recibir eventos. Este método tiene dos argumentos. La primera es el evento de integración al que suscribirse (`IntegrationEvent`). El segundo argumento es el controlador de eventos de integración (o método de devolución de llamada), denominado `IIntegrationEventHandler<T>`, que se ejecutará cuando el microservicio receptor obtiene ese mensaje de evento de integración.

Recursos adicionales

Algunas soluciones de mensajería listas para producción:

- **Azure Service Bus**
  https://learn.microsoft.com/azure/service-bus-messaging/
- **NServiceBus**
  https://particular.net/nservicebus
- **MassTransit**
  https://masstransit-project.com/

Siguiente anterior

Implementación de un bus de eventos con RabbitMQ para el entorno de desarrollo o prueba

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Deberíamos empezar diciendo que si crea el bus de eventos personalizado basado en RabbitMQ que se ejecuta en un contenedor, como hace la aplicación eShopOnContainers, solo debe usarse para los entornos de desarrollo y prueba. No lo use para el entorno de producción, a menos que lo cree como parte de un bus de servicio listo para producción, tal como se describe en la sección Recursos adicionales que aparece a continuación. Un bus de eventos personalizado simple podría carecer de muchas características críticas preparadas para producción que tiene un bus de servicios comerciales.

Una de las implementaciones personalizadas de Event Bus en eShopOnContainers es básicamente una biblioteca que usa la API RabbitMQ. (Hay otra implementación basada en Azure Service Bus).

La implementación del bus de eventos con RabbitMQ permite a los microservicios suscribirse a eventos, publicar eventos y recibir eventos, como se muestra en la figura 6-21.

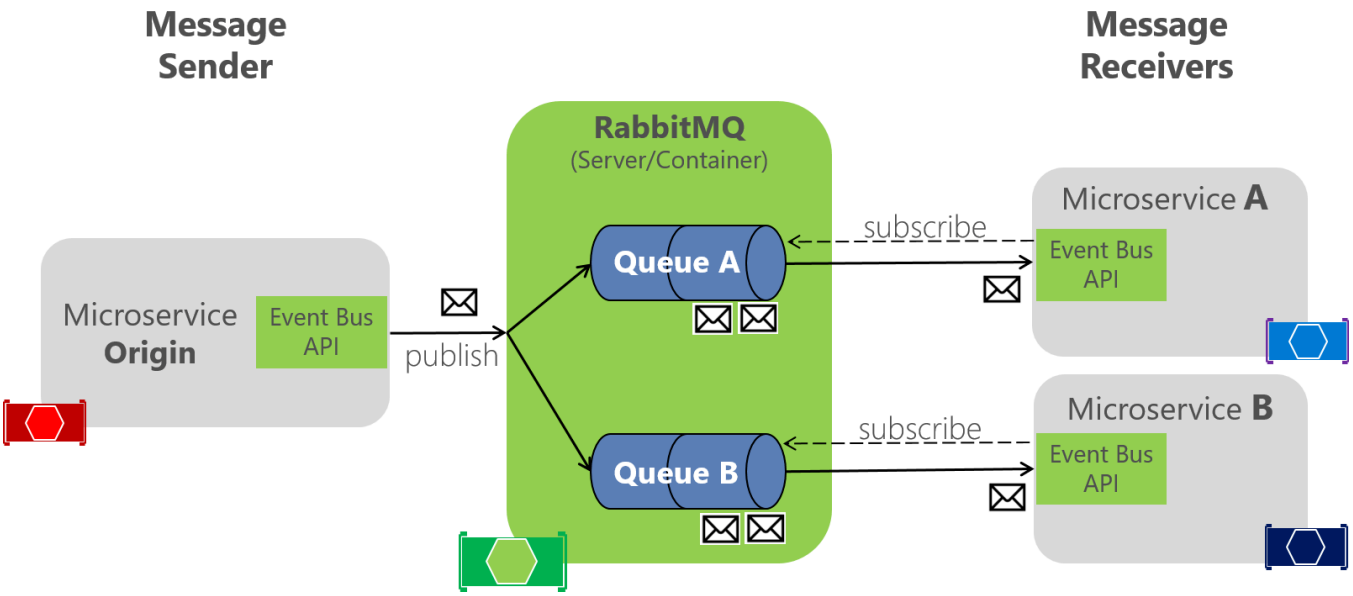


Figura 6-21. Implementación de RabbitMQ de un bus de eventos

RabbitMQ funciona como intermediario entre el publicador de mensajes y los suscriptores, para controlar la distribución. En el código, la clase EventBusRabbitMQ implementa la interfaz IEventBus genérica. Esta implementación se basa en la inserción de dependencias para que pueda cambiar de esta versión de desarrollo/prueba a una versión de producción.

```
C#

public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Implementation using RabbitMQ API
    //...
}
```

La implementación de RabbitMQ de un ejemplo de bus de eventos de desarrollo/prueba es código estándar. Tiene que controlar la conexión con el servidor RabbitMQ y proporcionar código para publicar un evento de mensaje a las colas. También tiene que implementar un diccionario de colecciones de controladores de eventos de integración para cada tipo de evento; estos tipos de eventos pueden tener una instanciación diferente y suscripciones diferentes para cada microservicio receptor, como se muestra en la figura 6-21.

Implementación de un método de publicación simple con RabbitMQ

El código siguiente es una versión ***simplificada*** de una implementación de bus de eventos para RabbitMQ, para mostrar todo el escenario. Lo cierto es que este no es el modo de controlar la conexión. Para ver la implementación completa, consulte el código real en el repositorio dotnet-architecture/eShopOnContainers .

C#

```
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods ...
    // ...

    public void Publish(IntegrationEvent @event)
    {
        var eventName = @event.GetType().Name;
        var factory = new ConnectionFactory() { HostName = _connectionString };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: _brokerName,
                type: "direct");
            string message = JsonConvert.SerializeObject(@event);
            var body = Encoding.UTF8.GetBytes(message);
            channel.BasicPublish(exchange: _brokerName,
                routingKey: eventName,
                basicProperties: null,
                body: body);
       }
    }
}
```

El código real del método Publish en la aplicación eShopOnContainers se mejora mediante una directiva de reintento de Polly , que vuelve a intentar la tarea algunas veces en caso de que el contenedor RabbitMQ no esté listo. Este escenario puede producirse cuando docker-compose inicia los contenedores; Por ejemplo, el contenedor RabbitMQ podría iniciarse más lentamente que los demás contenedores.

Como se mencionó anteriormente, hay muchas configuraciones posibles en RabbitMQ, por lo que este código solo se debe usar para entornos de desarrollo y pruebas.

Implementación del código de suscripción con rabbitMQ API

Al igual que con el código de publicación, el código siguiente es una simplificación de parte de la implementación del bus de eventos para RabbitMQ. De nuevo, normalmente no es necesario cambiarlo a menos que quieras mejorarlo.

C#

```
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods ...
```

```
    // ...

    public void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIntegrationEventHandler<T>
    {
        var eventName = _subsManager.GetEventKey<T>();

        var containsKey = _subsManager.HasSubscriptionsForEvent(eventName);
        if (!containsKey)
        {
            if (!_persistentConnection.IsConnected)
            {
                _persistentConnection.TryConnect();
            }

            using (var channel = _persistentConnection.CreateModel())
            {
                channel.QueueBind(queue: _queueName,
                                  exchange: BROKER_NAME,
                                  routingKey: eventName);
            }
        }

        _subsManager.AddSubscription<T, TH>();
    }
}
```

Cada tipo de evento tiene un canal relacionado para obtener eventos de RabbitMQ. A continuación, puede tener tantos controladores de eventos por canal como tipo de evento según sea necesario.

El método Subscribe acepta un objeto IIntegrationEventHandler, que es como un método de devolución de llamada en el microservicio actual, además de su objeto IntegrationEvent relacionado. A continuación, el código agrega ese controlador de eventos a la lista de controladores de eventos que cada tipo de evento de integración puede tener por microservicio de cliente. Si el código de cliente aún no se ha suscrito al evento, el código crea un canal para el tipo de evento para que pueda recibir eventos en un estilo push de RabbitMQ cuando se publique el evento desde cualquier otro servicio.

Como se mencionó anteriormente, el bus de eventos implementado en eShopOnContainers solo tiene un propósito educativo, ya que solo controla los escenarios principales, por lo que no está listo para producción.

Para escenarios de producción, compruebe los recursos adicionales siguientes, específicos de RabbitMQ y la sección Implementación de la comunicación basada en eventos entre microservicios .

Recursos adicionales

Una solución lista para producción compatible con RabbitMQ.

- **Peregrine Connect** : simplifica la integración con un diseño, implementación y administración eficientes de aplicaciones, API y flujos de trabajo https://www.peregrineconnect.com/why-peregrine/rabbitmq-integration
- **NServiceBus** : bus de servicio comercial totalmente compatible con herramientas avanzadas de administración y supervisión para .NET https://particular.net/
- **EasyNetQ** : cliente de API de .NET de código abierto para RabbitMQ https://easynetq.com/
- **MassTransit** : marco de aplicaciones distribuidas de código abierto gratis para .NET https://masstransit-project.com/
- **Rebus** : Bus de servicio .NET de código abierto https://github.com/rebus-org/Rebus

Siguiente anterior

Suscribirse a eventos

19/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El primer paso para usar el bus de eventos es suscribir los microservicios a los eventos que quieren recibir. Esa funcionalidad debe realizarse en los microservicios receptores.

En el código simple siguiente se muestra lo que cada microservicio receptor necesita implementar al iniciar el servicio (es decir, en la `Startup` clase) para que se suscriba a los eventos que necesita. En este caso, el `basket-api` microservicio debe suscribirse a los mensajes de `ProductPriceChangedIntegrationEvent` y `OrderStartedIntegrationEvent`.

Por ejemplo, al suscribirse al `ProductPriceChangedIntegrationEvent` evento, que hace que el microservicio de cesta sea consciente de cualquier cambio en el precio del producto y le permite advertir al usuario sobre el cambio si ese producto está en la cesta del usuario.

C#

```
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();

eventBus.Subscribe<ProductPriceChangedIntegrationEvent,
                   ProductPriceChangedIntegrationEventHandler>();

eventBus.Subscribe<OrderStartedIntegrationEvent,
                   OrderStartedIntegrationEventHandler>();
```

Después de ejecutar este código, el microservicio de suscriptor escuchará a través de los canales de RabbitMQ. Cuando llega cualquier mensaje de tipo ProductPriceChangedIntegrationEvent, el código invoca al controlador de eventos que se pasa a él y procesa el evento.

Publicación de eventos a través del bus de eventos

Por último, el remitente del mensaje (microservicio de origen) publica los eventos de integración con código similar al ejemplo siguiente. (Este enfoque es un ejemplo simplificado que no tiene en cuenta la atomicidad). Implementaría código similar cada vez que se debe propagar un evento entre varios microservicios, normalmente justo después de confirmar datos o transacciones desde el microservicio de origen.

En primer lugar, el objeto de implementación del bus de eventos (basado en RabbitMQ o basado en un bus de servicio) se insertaría en el constructor del controlador, como en el código siguiente:

C#

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _context;
    private readonly IOptionsSnapshot<Settings> _settings;
    private readonly IEventBus _eventBus;

    public CatalogController(CatalogContext context,
        IOptionsSnapshot<Settings> settings,
        IEventBus eventBus)
    {
        _context = context;
        _settings = settings;
        _eventBus = eventBus;
    }
    // ...
}
```

Luego, lo utilizas desde los métodos de tu controlador, como en el método UpdateProduct.

C#

```
[Route("items")]
[HttpPost]
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem product)
{
    var item = await _context.CatalogItems.SingleOrDefaultAsync(
        i => i.Id == product.Id);
```

```
    // ...
    if (item.Price != product.Price)
    {
        var oldPrice = item.Price;
        item.Price = product.Price;
        _context.CatalogItems.Update(item);
        var @event = new ProductPriceChangedIntegrationEvent(item.Id,
            item.Price,
            oldPrice);
        // Commit changes in original transaction
        await _context.SaveChangesAsync();
        // Publish integration event to the event bus
        // (RabbitMQ or a service bus underneath)
        _eventBus.Publish(@event);
        // ...
    }
    // ...
}
```

En este caso, dado que el microservicio de origen es un microservicio CRUD simple, ese código se coloca directamente en un controlador de API web.

En microservicios más avanzados, como cuando se usan enfoques de CQRS, se puede implementar en la `CommandHandler` clase , dentro del `Handle()` método .

Diseño de la atomicidad y la resistencia al publicar en el bus de eventos

Al publicar eventos de integración a través de un sistema de mensajería distribuido como el bus de eventos, tiene el problema de actualizar la base de datos original de forma atómica y de publicar un evento (es decir, se completan las dos operaciones o ninguna de ellas). Por ejemplo, en el ejemplo simplificado mostrado anteriormente, el código confirma los datos en la base de datos cuando se cambia el precio del producto y, a continuación, publica un mensaje ProductPriceChangedIntegrationEvent. Inicialmente, podría parecer esencial que estas dos operaciones se realicen de forma atómica. Sin embargo, si usa una transacción distribuida que implica la base de datos y el agente de mensajes, como lo hace en sistemas antiguos como Microsoft Message Queuing (MSMQ), este enfoque no se recomienda por las razones descritas por el teorema CAP .

Básicamente, se usan microservicios para crear sistemas escalables y de alta disponibilidad. Al simplificar un poco, el teorema CAP dice que no se puede crear una base de datos (distribuida) (o un microservicio que posee su modelo) que esté continuamente disponible, fuertemente coherente *y* tolerante a cualquier partición. Debería elegir dos de estas tres propiedades.

En las arquitecturas basadas en microservicios, debe elegir la disponibilidad y la tolerancia, y quitar énfasis a la coherencia fuerte. Por lo tanto, en la mayoría de las aplicaciones modernas

basadas en microservicios, normalmente no desea usar transacciones distribuidas en mensajería, como sucede al implementar transacciones distribuidas basadas en el Coordinador de transacciones distribuidas (DTC) de Windows con MSMQ.

Volvamos al problema inicial y a su ejemplo. Si el servicio se bloquea después de actualizar la base de datos (en este caso, justo después de la línea de código con `_context.SaveChangesAsync()`), pero antes de que se publique el evento de integración, el sistema general podría ser incoherente. Este enfoque puede ser crítico para la empresa, en función de la operación empresarial específica con la que se trabaja.

Como se mencionó anteriormente en la sección de arquitectura, puede tener varios enfoques para tratar este problema:

- Uso del patrón de orígenes de eventos completo.
- Uso de la minería de datos del registro de transacciones.
- Uso del patrón de bandeja de salida. Se trata de una tabla transaccional para almacenar los eventos de integración (extensión de la transacción local).

En este escenario, el uso del patrón completo de Event Sourcing (ES) es uno de los mejores enfoques, si no es el mejor. Sin embargo, en muchos escenarios de aplicación, es posible que no pueda implementar un sistema ES completo. ES significa almacenar solo eventos de dominio en la base de datos transaccional, en lugar de almacenar los datos de estado actuales. Almacenar solo los eventos de dominio puede tener grandes ventajas, como tener el historial del sistema disponible y poder determinar el estado del sistema en cualquier momento del pasado. Sin embargo, la implementación de un sistema ES completo requiere rediseñar la mayor parte del sistema e introduce muchas otras complejidades y requisitos. Por ejemplo, le gustaría usar una base de datos hecha específicamente para el aprovisionamiento de eventos, como event Store o una base de datos orientada a documentos, como Azure Cosmos DB, MongoDB, Cassandra, CouchDB o RavenDB. Los orígenes de evento son un buen enfoque para este problema, pero no es la solución más sencilla a menos que ya esté familiarizado con los orígenes de eventos.

La opción para usar la minería de datos del registro de transacciones parece inicialmente transparente. Sin embargo, para usar este enfoque, el microservicio debe acoplarse al registro de transacciones de RDBMS, como el registro de transacciones de SQL Server. Este enfoque probablemente no es deseable. Otro inconveniente es que las actualizaciones de bajo nivel registradas en el registro de transacciones podrían no estar en el mismo nivel que los eventos de integración de alto nivel. Si es así, el proceso de ingeniería inversa de esas operaciones de registro de transacciones puede ser difícil.

Un enfoque equilibrado es una combinación de una tabla de base de datos transaccional y un patrón ES simplificado. Puede usar un estado como "listo para publicar el evento", que estableció en el evento original al confirmarlo en la tabla de eventos de integración. Después, intente publicar el evento en el bus de eventos. Si la acción publish-event se realiza correctamente, inicie otra transacción en el servicio de origen y mueva el estado de "listo para publicar el evento" a "evento ya publicado".

Si se produce un error en la acción de publicación de eventos en el bus de eventos, los datos seguirán siendo coherentes dentro del microservicio de origen, están todavía marcados como "listos para publicar el evento" y, con respecto al resto de los servicios, eventualmente serán coherentes. Siempre puede tener trabajos en segundo plano que comprueben el estado de las transacciones o los eventos de integración. Si el trabajo encuentra un evento en el estado "listo para publicar el evento", puede intentar volver a publicarlo en el bus de eventos.

Tenga en cuenta que, con este enfoque, solo se conservan los eventos de integración de cada microservicio de origen y solo los eventos que desea comunicar con otros microservicios o sistemas externos. En cambio, en un sistema ES completo, también se almacenan todos los eventos de dominio.

Por lo tanto, este enfoque equilibrado es un sistema ES simplificado. Necesita una lista de eventos de integración con su estado actual ("listo para publicar" frente a "publicado"). Pero solo tiene que implementar estos estados para los eventos de integración. Y en este enfoque, no es necesario almacenar todos los datos de dominio como eventos en la base de datos transaccional, como lo haría en un sistema ES completo.

Si ya usa una base de datos relacional, puede usar una tabla transaccional para almacenar eventos de integración. Para lograr la atomicidad en la aplicación, se usa un proceso de dos pasos basado en transacciones locales. Básicamente, tiene una tabla IntegrationEvent en la misma base de datos donde tiene las entidades de dominio. Esa tabla funciona como un seguro para lograr la atomicidad, de modo que los eventos de integración guardados se incluyan en las mismas transacciones con las que se confirman los datos de dominio.

Paso a paso, el proceso es similar al siguiente:

1. La aplicación inicia una transacción de base de datos local.
2. A continuación, actualiza el estado de las entidades de dominio e inserta un evento en la tabla de eventos de integración.
3. Por último, confirma la transacción, por lo que obtendrá la atomicidad deseada y, a continuación,
4. Publique el evento de alguna manera (próximo paso).

Al implementar los pasos para publicar los eventos, tiene estas opciones:

- Publique el evento de integración justo después de confirmar la transacción y use otra transacción local para marcar los eventos de la tabla como publicados. A continuación, use la tabla igual que un artefacto para realizar un seguimiento de los eventos de integración en caso de problemas en los microservicios remotos y realizar acciones compensatorias basadas en los eventos de integración almacenados.

- Usar la tabla como una especie de cola. Un subproceso de aplicación independiente o un proceso consulta la tabla de eventos de integración, publica los eventos en el bus de eventos y, a continuación, usa una transacción local para marcar los eventos como publicados.

En la figura 6-22 se muestra la arquitectura para el primero de estos enfoques.

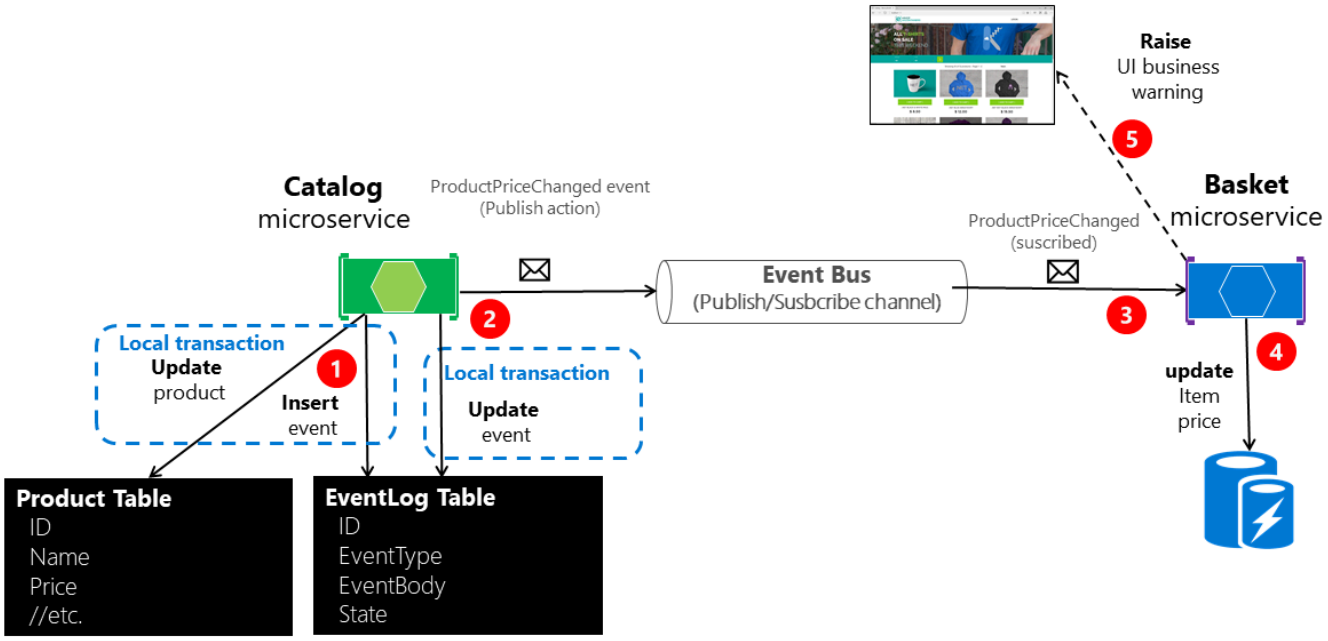


Figura 6-22. Atomicidad al publicar eventos en el bus de eventos

En el enfoque que se muestra en la figura 6-22 falta un microservicio de trabajo adicional que se encarga de comprobar y confirmar que los eventos de integración se han publicado correctamente. En caso de error, ese microservicio de trabajo de comprobación adicional puede leer los eventos de la tabla y volver a publicarlos, es decir, repetir el paso 2.

Acerca del segundo enfoque: se utiliza la tabla EventLog como una cola y siempre se emplea un microservicio de trabajo para publicar los mensajes. En ese caso, el proceso es similar al que se muestra en la figura 6-23. Esto muestra un microservicio adicional y la tabla es el único origen al publicar eventos.

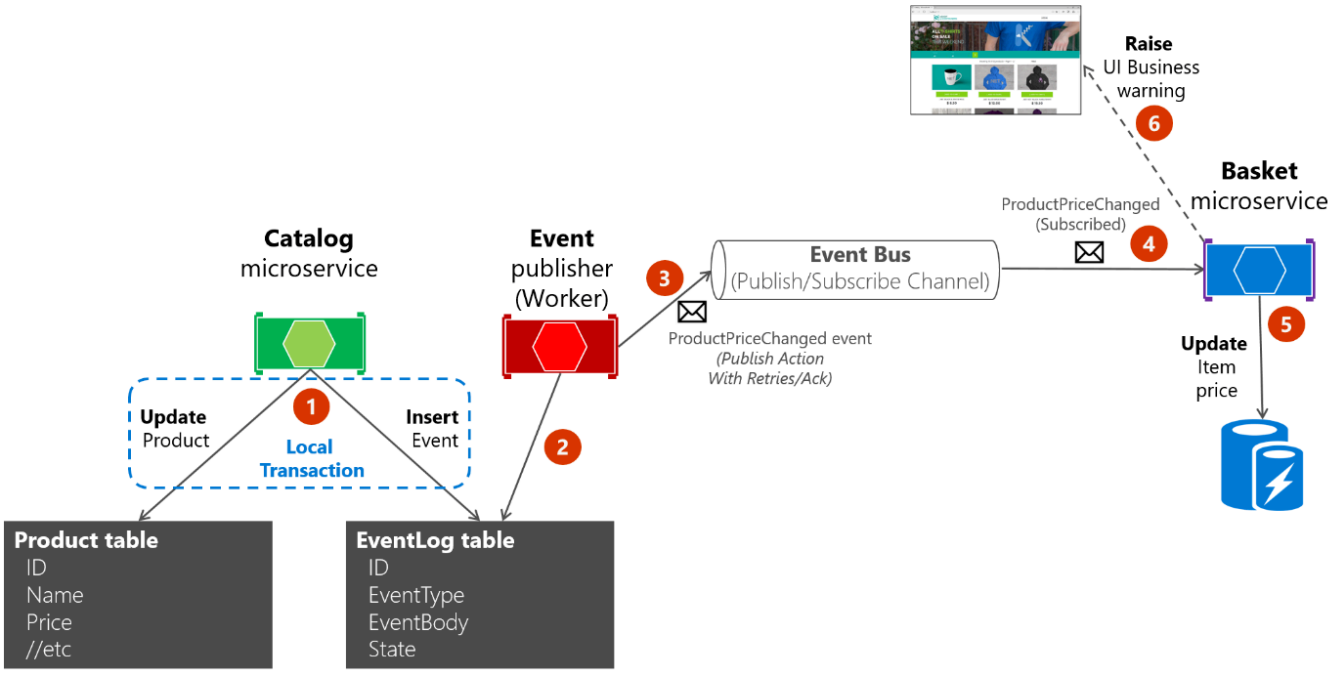


Figura 6-23. Atomicidad al publicar eventos en el bus de eventos con un microservicio de trabajo

Por motivos de simplicidad, el ejemplo eShopOnContainers usa el primer enfoque (sin procesos adicionales ni microservicios de comprobador) más el bus de eventos. Sin embargo, el ejemplo de eShopOnContainers no controla todos los posibles casos de error. En una aplicación real implementada en la nube, debe adoptar el hecho de que surjan problemas finalmente y debe implementar esa lógica de comprobación y reenvío. El uso de la tabla como una cola puede ser más eficaz que el primer enfoque si tiene esa tabla como un único origen de eventos cuando los publica (con el trabajo) a través del bus de eventos.

Implementación de la atomicidad al publicar eventos de integración a través del bus de eventos

En el código siguiente se muestra cómo se puede crear una sola transacción que implique varios objetos DbContext: un contexto relacionado con los datos originales que se actualizan y el segundo contexto relacionado con la tabla IntegrationEventLog.

La transacción del código de ejemplo siguiente no será resistente si las conexiones a la base de datos tienen algún problema en el momento en que se ejecuta el código. Esto puede ocurrir en sistemas basados en la nube, como Azure SQL DB, que podrían mover bases de datos entre servidores. Para implementar transacciones resistentes en varios contextos, consulte la sección Implementación de conexiones SQL de Entity Framework Core resistentes más adelante en esta guía.

Para mayor claridad, en el ejemplo siguiente se muestra todo el proceso en un solo fragmento de código. Sin embargo, la implementación de eShopOnContainers se refactoriza y divide esta

lógica en varias clases para que sea más fácil de mantener.

C#

```
// Update Product from the Catalog microservice
//
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem
productToUpdate)
{
  var catalogItem =
       await _catalogContext.CatalogItems.SingleOrDefaultAsync(i => i.Id ==

productToUpdate.Id);
  if (catalogItem == null) return NotFound();

  bool raiseProductPriceChangedEvent = false;
  IntegrationEvent priceChangedEvent = null;

  if (catalogItem.Price != productToUpdate.Price)
          raiseProductPriceChangedEvent = true;

  if (raiseProductPriceChangedEvent) // Create event if price has changed
  {
      var oldPrice = catalogItem.Price;
      priceChangedEvent = new ProductPriceChangedIntegrationEvent(catalogItem.Id,

productToUpdate.Price,
                                                                    oldPrice);
  }
  // Update current product
  catalogItem = productToUpdate;

  // Just save the updated product if the Product's Price hasn't changed.
  if (!raiseProductPriceChangedEvent)
  {
      await _catalogContext.SaveChangesAsync();
  }
  else  // Publish to event bus only if product price changed
  {
        // Achieving atomicity between original DB and the IntegrationEventLog
        // with a local transaction
        using (var transaction = _catalogContext.Database.BeginTransaction())
        {
           _catalogContext.CatalogItems.Update(catalogItem);
           await _catalogContext.SaveChangesAsync();

           await _integrationEventLogService.SaveEventAsync(priceChangedEvent);

           transaction.Commit();
        }

      // Publish the integration event through the event bus
      _eventBus.Publish(priceChangedEvent);
```

```
        _integrationEventLogService.MarkEventAsPublishedAsync(
                                                    priceChangedEvent);
    }

    return Ok();
  }
```

Después de crear el evento de integración ProductPriceChangedIntegrationEvent, la transacción que almacena la operación de dominio original (actualizar el elemento de catálogo) también incluye la persistencia del evento en la tabla EventLog. Esto lo convierte en una sola transacción y siempre podrá comprobar si se enviaron mensajes de evento.

La tabla del registro de eventos se actualiza de forma atómica con la operación de base de datos original, mediante una transacción local en la misma base de datos. Si se produce un error en alguna de las operaciones, se produce una excepción y la transacción revierte cualquier operación completada, manteniendo así la coherencia entre las operaciones de dominio y los mensajes de evento guardados en la tabla.

Recepción de mensajes de suscripciones: controladores de eventos en microservicios receptores

Además de la lógica de suscripción de eventos, debe implementar el código interno para los manejadores de eventos de integración (como un método callback). El controlador de eventos es donde se especifica dónde se recibirán y procesarán los mensajes de evento de un tipo determinado.

Un controlador de eventos recibe primero una instancia de evento del bus de eventos. A continuación, busca el componente que se va a procesar relacionado con ese evento de integración, propagando y conservando el evento como un cambio en el estado en el microservicio receptor. Por ejemplo, si un evento ProductPriceChanged se origina en el microservicio de catálogo, se controla en el microservicio de cesta y cambia el estado de este microservicio de cesta del receptor, como se muestra en el código siguiente.

C#

```
namespace
Microsoft.eShopOnContainers.Services.Basket.API.IntegrationEvents.EventHandling
{
    public class ProductPriceChangedIntegrationEventHandler :
        IIntegrationEventHandler<ProductPriceChangedIntegrationEvent>
    {
        private readonly IBasketRepository _repository;

        public ProductPriceChangedIntegrationEventHandler(
            IBasketRepository repository)
        {
```

```
            _repository = repository;
        }

        public async Task Handle(ProductPriceChangedIntegrationEvent @event)
        {
            var userIds = await _repository.GetUsers();
            foreach (var id in userIds)
            {
                var basket = await _repository.GetBasket(id);
                await UpdatePriceInBasketItems(@event.ProductId, @event.NewPrice,
basket);
            }
        }

        private async Task UpdatePriceInBasketItems(int productId, decimal
newPrice,
            CustomerBasket basket)
        {
            var itemsToUpdate = basket?.Items?.Where(x => int.Parse(x.ProductId)
==
                productId).ToList();
            if (itemsToUpdate != null)
            {
                foreach (var item in itemsToUpdate)
                {
                    if(item.UnitPrice != newPrice)
                    {
                        var originalPrice = item.UnitPrice;
                        item.UnitPrice = newPrice;
                        item.OldUnitPrice = originalPrice;
                    }
                }
                await _repository.UpdateBasket(basket);
            }
        }
    }
}
```

El controlador de eventos debe comprobar si el producto existe en cualquiera de las instancias de cesta. También actualiza el precio del artículo para cada artículo de línea de cesta relacionado. Por último, crea una alerta que se mostrará al usuario sobre el cambio de precio, como se muestra en la figura 6-24.

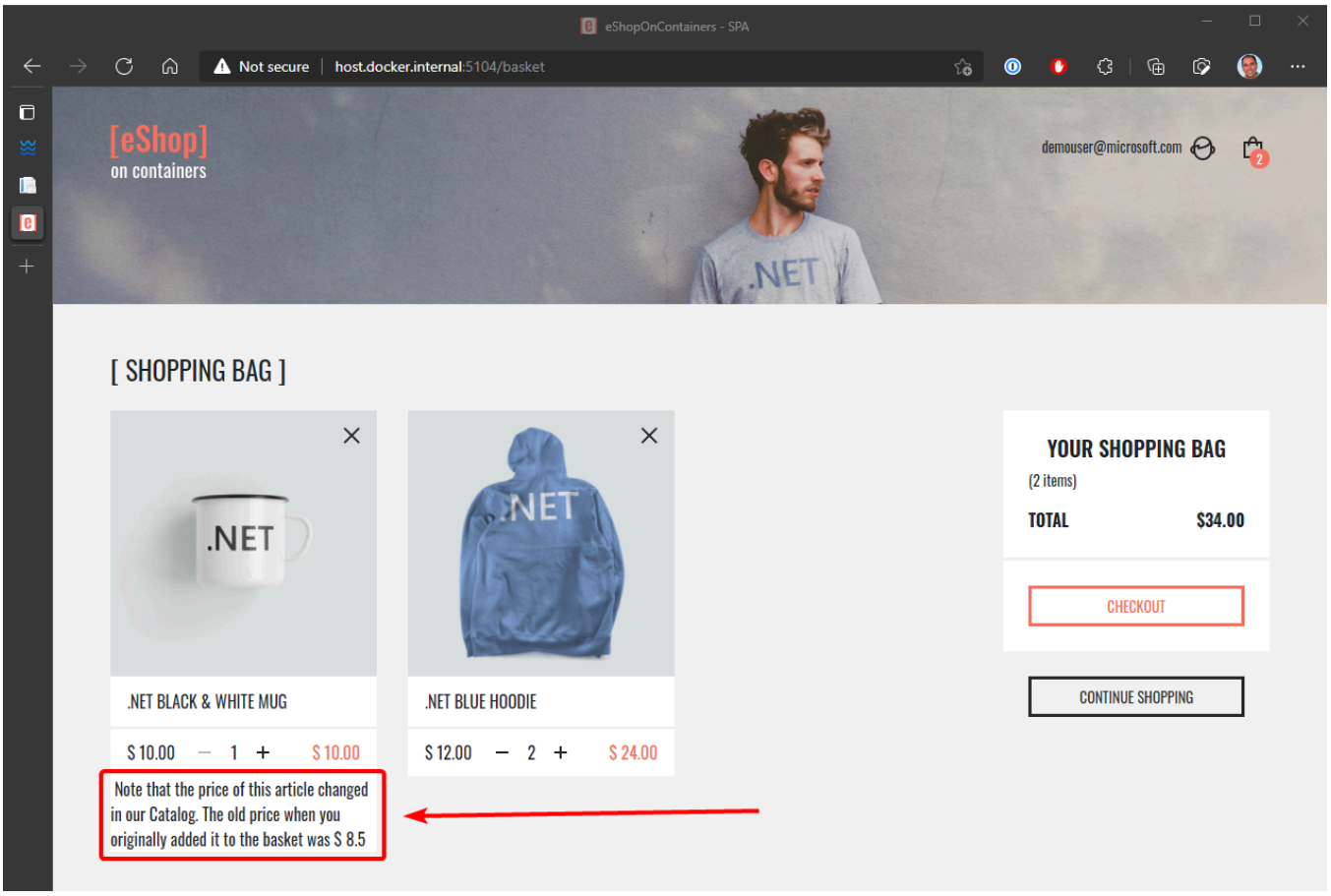


Figura 6-24. Representación de un cambio del precio de un artículo en una cesta, comunicado por eventos de integración

Idempotencia en los eventos de mensajes de actualización

Un aspecto importante de los eventos de mensaje de actualización es que un error en cualquier punto de la comunicación debe hacer que se vuelva a intentar el mensaje. En caso contrario, es posible que una tarea en segundo plano intente publicar un evento que ya se ha publicado, lo que genera una condición de carrera. Asegúrese de que las actualizaciones sean idempotentes o que proporcionen suficiente información para asegurarse de que puede detectar un duplicado, descartarlo y devolver solo una respuesta.

Como se indicó anteriormente, la idempoencia significa que una operación se puede realizar varias veces sin cambiar el resultado. En un entorno de mensajería, como al comunicar eventos, un evento es idempotente si se puede entregar varias veces sin cambiar el resultado del microservicio receptor. Esto puede ser necesario debido a la naturaleza del propio evento o a la forma en que el sistema controla el evento. La idempotencia del mensaje es importante en cualquier aplicación que use mensajería, y no solo en las que implementan el patrón del bus de eventos.

Un ejemplo de una operación idempotent es una instrucción SQL que inserta datos en una tabla solo si esos datos aún no están en la tabla. No importa cuántas veces ejecute esa instrucción SQL; el resultado será el mismo: la tabla contendrá esos datos. La idempotencia de este tipo también puede ser necesaria cuando se trata de mensajes si estos podrían enviarse y, por tanto, procesarse más de una vez. Por ejemplo, si la lógica de reintento hace que un remitente envíe exactamente el mismo mensaje más de una vez, debe asegurarse de que es idempotente.

Es posible diseñar mensajes idempotentes. Por ejemplo, puede crear un evento que diga "establecer el precio del producto en $25''enlugarde''agregar5$ al precio del producto". Puede procesar de forma segura el primer mensaje cualquier número de veces y el resultado será el mismo. Esto no es cierto para el segundo mensaje. Pero incluso en el primer caso, es posible que no desee procesar el primer evento, ya que el sistema también podría haber enviado un evento de cambio de precio más reciente y esto sobrescribiría el nuevo precio.

Otro ejemplo podría ser un evento de pedido completado que se propaga a varios suscriptores. La aplicación tiene que asegurarse de que la información del pedido se actualiza en otros sistemas solo una vez, incluso si hay eventos de mensaje duplicados para el mismo evento de pedido completado.

Es conveniente tener algún tipo de identidad por evento para que pueda crear lógica que exija que cada evento se procese solo una vez por receptor.

Algunos procesamientos de mensajes son intrínsecamente idempotentes. Por ejemplo, si un sistema genera miniaturas de imagen, es posible que no importa cuántas veces se procese el mensaje sobre la miniatura generada; el resultado es que se generan las miniaturas y son las mismas cada vez. Por otro lado, operaciones como la llamada a una pasarela de pago para cobrar una tarjeta de crédito pueden no ser idempotentes en absoluto. En estos casos, debe asegurarse de que el procesamiento de un mensaje varias veces tiene el efecto esperado.

Recursos adicionales

- **Respetar la idempotencia del mensaje**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591565(v=pandp.10)#honoring-message-idempotency

Eliminación de duplicados en mensajes de eventos de integración

Puede asegurarse de que los eventos de mensaje se envían y procesan solo una vez por suscriptor en distintos niveles. Una manera es usar una característica de desduplicación que

ofrece la infraestructura de mensajería que usa. Otro es implementar lógica personalizada en el microservicio de destino. Tener validaciones tanto en el nivel de transporte como en el nivel de aplicación es la mejor opción.

Eliminación de duplicados de eventos de mensaje al nivel de EventHandler

Una manera de asegurarse de que un evento solo se procesa una vez por cualquier receptor es implementando cierta lógica al procesar los eventos de mensaje en controladores de eventos. Por ejemplo, es el enfoque usado en la aplicación eShopOnContainers, como puede ver en el código fuente de la clase UserCheckoutAcceptedIntegrationEventHandler cuando recibe un `UserCheckoutAcceptedIntegrationEvent` evento de integración. (En este caso, `CreateOrderCommand` se envuelve con un `IdentifiedCommand`, usando `eventMsg.RequestId` como identificador, antes de enviarlo al controlador de comandos).

Desduplicación de mensajes al usar RabbitMQ

Cuando se producen errores de red intermitentes, los mensajes se pueden duplicar y el receptor de mensajes debe estar listo para controlar estos mensajes duplicados. Si es posible, los receptores deben controlar los mensajes de una manera idempotente, lo que es mejor que controlarlos de forma explícita mediante desduplicación.

Según la documentación de RabbitMQ, "si un mensaje se entrega a un consumidor y después se vuelve a poner en la cola (porque no se confirmó antes de desconectar la conexión del consumidor, por ejemplo), RabbitMQ establecerá la marca 'entregado de nuevo' cuando se vuelva a entregar (con independencia de que sea al mismo consumidor o a otro)".

Si se establece la marca "redelivered", el receptor debe tenerlo en cuenta, ya que es posible que el mensaje ya se haya procesado. Pero eso no está garantizado; es posible que el mensaje nunca haya llegado al receptor después de dejar el agente de mensajes, quizás debido a problemas de red. Por otro lado, si no se establece la marca "redelivered", se garantiza que el mensaje no se haya enviado más de una vez. Por tanto, el receptor debe desduplicar o procesar los mensajes de una manera idempotente solo si se establece la marca "entregado de nuevo" en el mensaje.

Recursos adicionales

- **Bifurcación de eShopOnContainers mediante NServiceBus [software particular]** https://go.particular.net/eShopOnContainers

- **Mensajería controlada por eventos**
  https://patterns.arcitura.com/soa-patterns/design_patterns/event_driven_messaging
- **Jimmy Bogard. Refactorización hacia la resiliencia: evaluación del acoplamiento**
  https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/
- **canal dePublish-Subscribe**
  https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
- **Comunicación entre contextos enlazados**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591572(v=pandp.10)
- **Coherencia eventual**
  https://en.wikipedia.org/wiki/Eventual_consistency
- **Felipe Brown. Estrategias para integrar contextos limitados**
  https://www.culttt.com/2014/11/26/strategies-integrating-bounded-contexts/
- **Chris Kemp. Desarrollo de microservicios transaccionales mediante agregados, orígenes de eventos y CQRS: parte 2**
  https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson
- **Chris Kemp. Patrón Event Sourcing**
  https://microservices.io/patterns/data/event-sourcing.html
- **Introducción a los orígenes de eventos**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591559(v=pandp.10)
- **Base de datos del Almacén de eventos**. Sitio oficial.
  https://geteventstore.com/
- **Patrick Nommensen. Event-Driven Administración de Datos para Microservicios**
  https://dzone.com/articles/event-driven-data-management-for-microservices-1
- **Teorema CAP**
  https://en.wikipedia.org/wiki/CAP_theorem
- **¿Qué es el teorema CAP?**
  https://www.quora.com/What-Is-CAP-Theorem-1
- **Información básica sobre la consistencia de datos**
  https://learn.microsoft.com/previous-versions/msp-n-p/dn589800(v=pandp.10)
- **Rick Saling. El teorema CAP: Por qué "todo es diferente" con la nube e Internet**
  https://learn.microsoft.com/archive/blogs/rickatmicrosoft/the-cap-theorem-why-

everything-is-different-with-the-cloud-and-internet/

- **Eric Brewer. CAP doce años más tarde: Cómo han cambiado las "reglas"**
  https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
- **CAP, PACELC y microservicios**
  https://ardalis.com/cap-pacelc-and-microservices/
- **Azure Service Bus. Mensajería intermediada: detección de duplicados**
  https://github.com/microsoftarchive/msdn-code-gallery-microsoft/tree/master/Windows%20Azure%20Product%20Team/Brokered%20Messaging%20Duplicate%20Detection
- **Guía de confiabilidad** (documentación de RabbitMQ)
  https://www.rabbitmq.com/reliability.html#consumer

Siguiente anterior

Probar aplicaciones web y servicios ASP.NET Core

Artículo • 15/10/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Los controladores son una parte fundamental de cualquier servicio de la API de ASP.NET Core y de la aplicación web de ASP.NET MVC. Por lo tanto, debe tener la seguridad de que se comportan según lo previsto en la aplicación. Las pruebas automatizadas pueden darle esta seguridad, así como detectar errores antes de que lleguen a la fase producción.

Debe probar cómo se comporta el controlador según las entradas válidas o no válidas y probar las respuestas del controlador en función del resultado de la operación comercial que lleve a cabo. Pero debe realizar estos tipos de pruebas en los microservicios:

- Pruebas unitarias. Estas pruebas garantizan que los componentes individuales de la aplicación funcionen según lo previsto. Las aserciones prueban la API del componente.

- Pruebas de integración. Estas pruebas garantizan que las interacciones de los componentes funcionen según lo previsto con los artefactos externos como bases de datos. Las aserciones pueden poner a prueba la API del componente, la interfaz de usuario o los efectos secundarios de acciones como la E/S de la base de datos, el registro, etc.

- Pruebas funcionales para cada microservicio. Estas pruebas garantizan que la aplicación funcione según lo esperado desde la perspectiva del usuario.
- Pruebas de servicio. Estas pruebas garantizan que se pongan a prueba todos los casos de uso de servicio de un extremo a otro, incluidas pruebas de servicios múltiples al mismo tiempo. Para este tipo de prueba, primero debe preparar el entorno. En este caso, esto significa iniciar los servicios (por ejemplo, mediante el uso de Docker Compose).

Implementación de pruebas unitarias para las API web de ASP.NET Core

Las pruebas unitarias conllevan probar una parte de una aplicación de forma aislada con respecto a su infraestructura y dependencias. Cuando se realizan pruebas unitarias de la lógica de controlador, solo se comprueba el método o el contenido de una única acción, no el comportamiento de sus dependencias o del marco en sí. Con todo, las pruebas unitarias no detectan problemas de interacción entre componentes; este es el propósito de las pruebas de integración.

Cuando realice pruebas unitarias de sus acciones de controlador, asegúrese de centrarse solamente en su comportamiento. Una prueba unitaria de controlador evita elementos como filtros, el enrutamiento o enlaces de modelos (la asignación de datos de solicitud a un ViewModel o DTO). Como se centran en comprobar solo una cosa, las pruebas unitarias suelen ser fáciles de escribir y rápidas de ejecutar. Un conjunto de pruebas unitarias bien escrito se puede ejecutar con frecuencia sin demasiada sobrecarga.

Las pruebas unitarias se implementan en función de los marcos de pruebas como xUnit.net, MSTest, Moq o NUnit. En la aplicación de ejemplo eShopOnContainers, se usa XUnit.

Al escribir una prueba unitaria para un controlador de API web, puede ejemplificar directamente la clase de controlador mediante la nueva palabra clave en C#, para que la prueba se ejecute tan rápido como sea posible. En el ejemplo siguiente se muestra cómo hacerlo con XUnit como marco de pruebas.

C#

```
[Fact]
public async Task Get_order_detail_success()
{
    //Arrange
    var fakeOrderId = "12";
    var fakeOrder = GetFakeOrder();
```

```
    //...

    //Act
    var orderController = new OrderController(
        _orderServiceMock.Object,
        _basketServiceMock.Object,
        _identityParserMock.Object);

    orderController.ControllerContext.HttpContext = _contextMock.Object;
    var actionResult = await orderController.Detail(fakeOrderId);

    //Assert
    var viewResult = Assert.IsType<ViewResult>(actionResult);
    Assert.IsAssignableFrom<Order>(viewResult.ViewData.Model);
}
```

Implementación de pruebas funcionales y de integración para cada microservicio

Como se ha indicado, las pruebas funcionales y de integración tienen objetivos y propósitos diferentes. Pero la forma de implementarlas para probar los controladores de ASP.NET Core es similar, por lo que en esta sección nos centraremos en las pruebas de integración.

Las pruebas de integración garantizan que los componentes de una aplicación funcionen correctamente durante el ensamblaje. ASP.NET Core admite las pruebas de integración que usan marcos de pruebas unitarias y un host de web de prueba integrado que puede usarse para controlar las solicitudes sin sobrecargar la red.

A diferencia de las pruebas unitarias, las pruebas de integración suelen incluir problemas de infraestructura de la aplicación, como base de datos, sistema de archivos, recursos de red o solicitudes web, y sus respuestas. Para las pruebas unitarias se usan emulaciones u objetos ficticios en lugar de estos problemas. Pero el propósito de las pruebas de integración es confirmar que el sistema funciona según lo previsto con estos sistemas, por lo que para las pruebas de integración no se usan simulaciones ni objetos ficticios. En cambio, se incluye la infraestructura, como el acceso a la base de datos o la invocación del servicio desde otros servicios.

Como las pruebas de integración usan segmentos de código más grandes que las pruebas unitarias y dependen de los elementos de infraestructura, tienden a ser órdenes de envergadura, más lentas que las pruebas unitarias. Por lo tanto, es conveniente limitar el número de pruebas de integración que va a escribir y a ejecutar.

ASP.NET Core incluye un host web de prueba integrado que puede usarse para controlar las solicitudes HTTP sin causar una sobrecarga en la red, lo que significa que

puede ejecutar dichas pruebas más rápidamente si usa un host de web real. El host web de prueba (TestServer) está disponible en un componente NuGet como Microsoft.AspNetCore.TestHost. Se puede agregar a proyectos de prueba de integración y utilizarlo para hospedar aplicaciones de ASP.NET Core.

Como puede ver en el código siguiente, al crear pruebas de integración para controladores de ASP.NET Core, los controladores se ejemplifican a través del host de prueba. Esta funcionalidad es comparable a una solicitud HTTP, pero se ejecuta con mayor rapidez.

C#

```
public class PrimeWebDefaultRequestShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;

    public PrimeWebDefaultRequestShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
           .UseStartup<Startup>());
        _client = _server.CreateClient();
    }

    [Fact]
    public async Task ReturnHelloWorld()
    {
        // Act
        var response = await _client.GetAsync("/");
        response.EnsureSuccessStatusCode();
        var responseString = await response.Content.ReadAsStringAsync();
        // Assert
        Assert.Equal("Hello World!", responseString);
    }
}
```

Recursos adicionales

- **Steve Smith. Probar los controladores** (ASP.NET Core)
  https://learn.microsoft.com/aspnet/core/mvc/controllers/testing

- **Steve Smith. Pruebas de integración** (ASP.NET Core)
  https://learn.microsoft.com/aspnet/core/test/integration-tests

- **Pruebas unitarias en .NET con dotnet test**
  https://learn.microsoft.com/dotnet/core/testing/unit-testing-with-dotnet-test

- **xUnit.net**. Sitio oficial.
  https://xunit.net/

- **Unit Test Basics** (Conceptos básicos de prueba unitaria).
  https://learn.microsoft.com/visualstudio/test/unit-test-basics

- **Moq**. Repositorio de GitHub.
  https://github.com/moq/moq

- **NUnit**. Sitio oficial.
  https://nunit.org/

Implementación de pruebas de servicio en una aplicación con varios contenedores

Como se indicó anteriormente, al probar aplicaciones con varios contenedores, todos los microservicios deben ejecutarse en el host de Docker o en un clúster de contenedor. Las pruebas de servicio de un extremo a otro que incluyen varias operaciones que implican varios microservicios requieren que implemente e inicie la aplicación en el host de Docker mediante la ejecución de Docker Compose (o un mecanismo comparable si usa un orquestador). Cuando la aplicación y todos sus servicios se estén ejecutando, podrá ejecutar pruebas funcionales y de integración de un extremo a otro.

Puede usar diferentes enfoques. En el archivo docker-compose.yml que se usa para implementar la aplicación en el nivel de solución puede expandir el punto de entrada para usar dotnet test. También puede usar otro archivo de composición que ejecute las pruebas en la imagen de destino. Si utiliza otro archivo de composición para las pruebas de integración, que incluyan sus microservicios y bases de datos en contenedores, puede comprobar que los datos relacionados siempre se restablecen a su estado original antes de ejecutar las pruebas.

Si ejecuta Visual Studio, cuando la aplicación de redacción esté en funcionamiento, podrá aprovechar los puntos de interrupción y las excepciones. También podrá ejecutar las pruebas de integración automáticamente en la canalización de integración continua en Azure DevOps Services o en cualquier otro sistema de integración continua o de entrega continua que admita los contenedores de Docker.

Realización de pruebas en eShopOnContainers

Recientemente se han reestructurado las pruebas de referencia de la aplicación (eShopOnContainers) y ahora hay cuatro categorías:

1. **Pruebas unitarias**, simples pruebas unitarias normales, incluidas en los proyectos **{MicroserviceName}.UnitTests**

2. **Pruebas de integración o funcionales de microservicio**, con casos de prueba que implican la infraestructura para cada microservicio, pero aisladas de los demás, y están incluidas en los proyectos **{MicroserviceName}. FunctionalTests**.

3. **Pruebas funcionales o de integración de aplicación**, que se centran en la integración de microservicios, con casos de prueba para ejercer varios microservicios. Estas pruebas se encuentran en el proyecto **Application.FunctionalTests**.

Mientras que las pruebas unitarias y de integración se organizan en una carpeta de prueba en el proyecto de microservicio, las pruebas de carga y aplicación se administran por separado en la carpeta raíz, como se muestra en la figura 6-25.

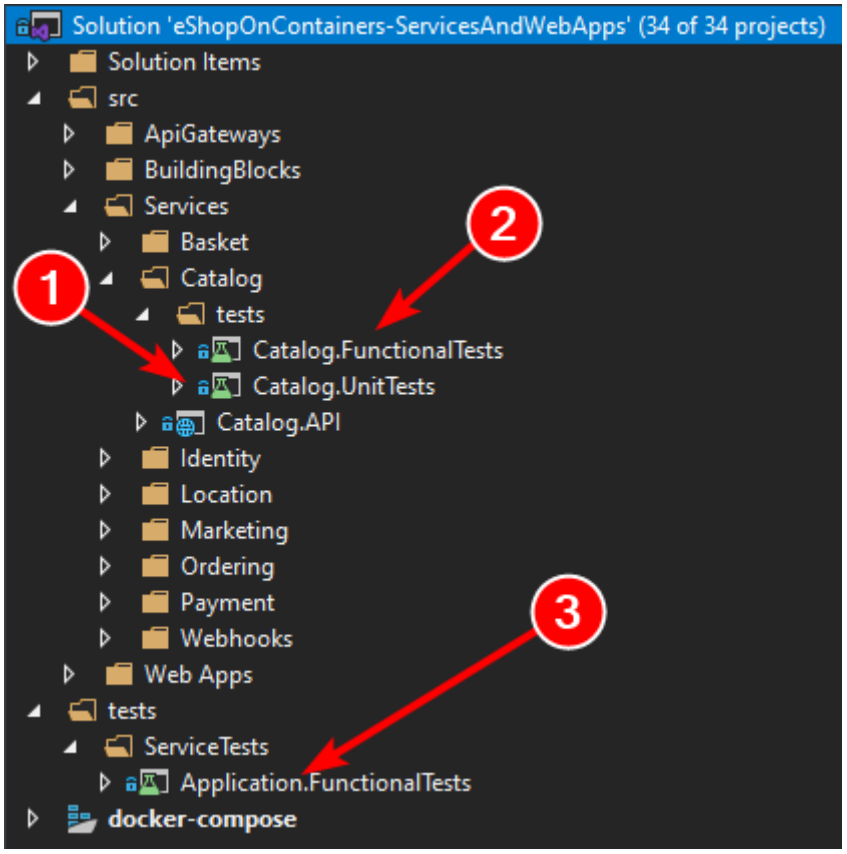


Figura 6-25. Estructura de carpetas de prueba en eShopOnContainers

Las pruebas de integración y funcionales de microservicios y aplicaciones se ejecutan desde Visual Studio, mediante el ejecutor de pruebas periódicas, pero primero debe iniciar los servicios de infraestructura necesarios, con un conjunto de archivos docker-compose incluidos en la carpeta de prueba de la solución:

docker-compose-test.yml

```
yml
```

```yaml
version: '3.4'

services:
  redis.data:
    image: redis:alpine
  rabbitmq:
    image: rabbitmq:3-management-alpine
  sqldata:
    image: mcr.microsoft.com/mssql/server:2017-latest
  nosqldata:
    image: mongo
```

docker-compose-test.override.yml

yml

```yaml
version: '3.4'

services:
  redis.data:
    ports:
      - "6379:6379"
  rabbitmq:
    ports:
      - "15672:15672"
      - "5672:5672"
  sqldata:
    environment:
      - SA_PASSWORD=[PLACEHOLDER]
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
  nosqldata:
    ports:
      - "27017:27017"
```

> ⓘ **Importante**
>
> Microsoft recomienda usar el flujo de autenticación más seguro disponible. Si se conecta a Azure SQL, el método de autenticación recomendado es [Identidades administradas para recursos de Azure](#).

Por tanto, para ejecutar las pruebas de integración y funcionales primero debe ejecutar este comando, desde la carpeta de prueba de la solución:

Consola

```
docker-compose -f docker-compose-test.yml -f docker-compose-
test.override.yml up
```

Como puede ver, estos archivos docker-compose solo inician los microservicios Redis, RabbitMQ, SQL Server y MongoDB.

Recursos adicionales

- **Pruebas unitarias y de integración** en eShopOnContainers https://github.com/dotnet-architecture/eShopOnContainers/wiki/Unit-and-integration-testing
- **Pruebas de carga** en eShopOnContainers https://github.com/dotnet-architecture/eShopOnContainers/wiki/Load-testing

Anterior Siguiente

Implementación de tareas en segundo plano en microservicios con IHostedService y la clase BackgroundService

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Las tareas en segundo plano y los trabajos programados son algo que podría necesitar usar en cualquier aplicación, independientemente de si sigue o no el patrón de arquitectura de microservicios. La diferencia al usar una arquitectura de microservicios es que puede implementar la tarea en segundo plano en un proceso o contenedor independiente para hospedarlo y poder escalarlo hacia abajo o hacia arriba según sus necesidades.

Desde un punto de vista genérico, en .NET llamamos a este tipo de tareas *Servicios hospedados*, ya que son servicios o lógicas que hospeda dentro del host, la aplicación o el microservicio. Tenga en cuenta que, en este caso, el servicio hospedado simplemente significa una clase con la lógica de tareas en segundo plano.

Desde .NET Core 2.0, el marco proporciona una nueva interfaz denominada IHostedService que le ayuda a implementar fácilmente los servicios hospedados. La idea básica es que puede registrar varias tareas de fondo (servicios hospedados) que se ejecutan mientras su servidor web o host está en funcionamiento, según se muestra en la imagen 6-26.

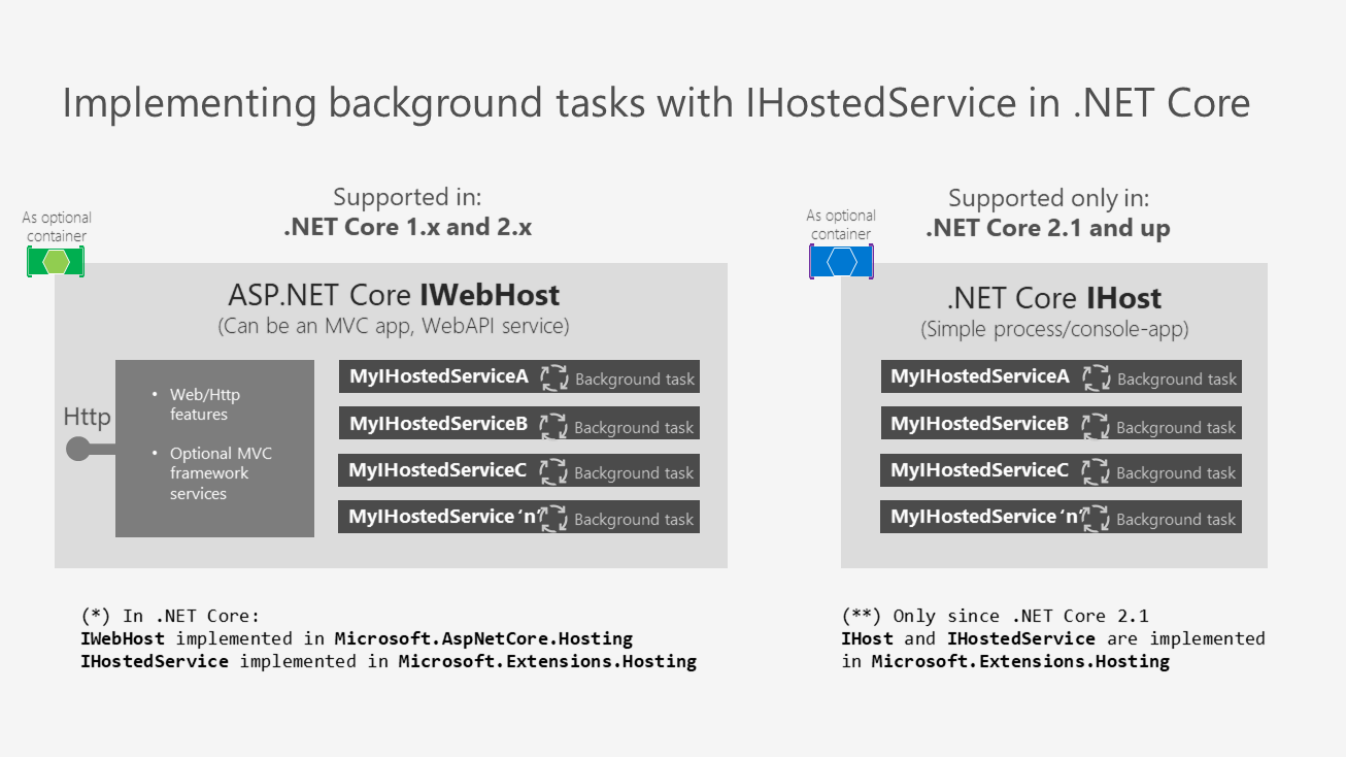


Figura 6-26. Uso de IHostedService en un WebHost frente a un host

ASP.NET Core 1.x y 2.x ofrecen compatibilidad `IWebHost` para procesos en segundo plano en aplicaciones web. .NET Core 2.1 y versiones posteriores admiten `IHost` para procesos en segundo plano con aplicaciones de consola simples. Observe la diferencia entre `WebHost` y `Host`.

Una clase base `WebHost` (que implementa `IWebHost`) en ASP.NET Core 2.0 es el artefacto de infraestructura que se usa para proporcionar características del servidor HTTP al proceso, como al implementar una aplicación web MVC o un servicio de API web. Proporciona todas las ventajas de la nueva infraestructura de ASP.NET Core, lo que le permite usar la inserción de dependencias e insertar middleware en la canalización de solicitudes, así como actividades similares. El `WebHost` usa estos mismos `IHostedServices` para las tareas en segundo plano.

Se introdujo una `Host` clase base que implementa `IHost` en .NET Core 2.1. Básicamente, `Host` permite tener una infraestructura similar a la que tiene con `WebHost` (inyección de dependencias, servicios hospedados, etc.), pero en este caso, solo quiere tener un proceso simple y más ligero como host, sin nada relacionado con las características de MVC, Web API o servidor HTTP.

Por lo tanto, puede optar por crear un proceso de host especializado con `IHost` para controlar los servicios hospedados y nada más, como un microservicio dedicado a hospedar exclusivamente `IHostedServices`. Alternativamente, puede ampliar una aplicación ASP.NET Core `WebHost` existente, tal como una aplicación Web API o MVC de ASP.NET Core existente.

Cada enfoque tiene ventajas y desventajas en función de sus necesidades empresariales y de escalabilidad. Lo fundamental es básicamente que si las tareas en segundo plano no tienen nada que ver con HTTP (`IWebHost`), usted debe usar `IHost`.

Registro de servicios hospedados en Host o WebHost

Vamos a profundizar más en la `IHostedService` interfaz, ya que su uso es bastante similar en un `WebHost` o en un `Host`.

SignalR es un ejemplo de un artefacto con servicios hospedados, pero también puede utilizarlo para cosas mucho más sencillas como las siguientes:

- Tarea en segundo plano que sondea una base de datos que busca cambios.
- Una tarea programada que actualiza algunas memorias caché periódicamente.
- Implementación de QueueBackgroundWorkItem que permite ejecutar una tarea en un subproceso en segundo plano.
- Procesar los mensajes de una cola de mensajes en el segundo plano de una aplicación web mientras se comparten servicios comunes como `ILogger`.
- Una tarea en segundo plano iniciada con `Task.Run()`.

Básicamente, puede trasladar cualquiera de esas acciones a una tarea en segundo plano que implemente `IHostedService`.

La manera de añadir uno o varios `IHostedServices` dentro de tus `WebHost` o `Host` es registrándolos mediante el método de extensión AddHostedService en un `WebHost` de ASP.NET Core (o en un `Host` en .NET Core 2.1 y versiones posteriores). Básicamente, tiene que registrar los servicios hospedados en el inicio de la aplicación en *Program.cs*.

C#

```
//Other DI registrations;

// Register Hosted Services
builder.Services.AddHostedService<GracePeriodManagerService>();
builder.Services.AddHostedService<MyHostedServiceB>();
builder.Services.AddHostedService<MyHostedServiceC>();
//...
```

En ese código, el `GracePeriodManagerService` servicio hospedado es código real del microservicio empresarial de pedidos en eShopOnContainers, mientras que los otros dos son solo dos ejemplos adicionales.

La `IHostedService` ejecución de la tarea en segundo plano se coordina con el ciclo de vida de la aplicación, tanto en el host como en el microservicio. Registra tareas cuando se inicia la aplicación y tiene la oportunidad de realizar alguna acción correcta o limpiar cuando la aplicación se apaga.

Sin usar `IHostedService`, siempre podría iniciar un subproceso en segundo plano para ejecutar cualquier tarea. La diferencia está precisamente en el momento de cierre de la aplicación, cuando ese subproceso simplemente terminaría sin tener ocasión de ejecutar las acciones de limpieza correcta.

Interfaz IHostedService

Al registrar un `IHostedService`, .NET llama a los métodos `StartAsync()` y `StopAsync()` de su tipo `IHostedService` durante el inicio y la detención de la aplicación, respectivamente. Para obtener más información, consulte Interfaz IHostedService.

Como puede imaginar, puede crear varias implementaciones de IHostedService y registrar cada una de ellas en *Program.cs*, como se mostró anteriormente. Todos esos servicios hospedados se iniciarán y detendrán junto con el microservicio o la aplicación.

Como desarrollador, usted es responsable de manejar la acción de detención de sus servicios cuando el método `StopAsync()` es desencadenado por el host.

Implementación de IHostedService con una clase de servicio hospedada personalizada derivada de la clase base BackgroundService

Puede seguir adelante y crear su clase personalizada de servicio alojado desde cero e implementar `IHostedService`, tal como debe hacer al usar .NET Core 2.0 y versiones posteriores.

Sin embargo, dado que la mayoría de las tareas en segundo plano tendrán necesidades similares con respecto a la administración de tokens de cancelación y otras operaciones típicas, hay una clase base abstracta conveniente a la que se puede derivar, denominada `BackgroundService` (disponible desde .NET Core 2.1).

Esta clase proporciona el trabajo principal necesario para configurar la tarea en segundo plano.

El código siguiente es la clase base abstracta BackgroundService tal como se implementa en .NET.

C#

```
// Copyright (c) .NET Foundation. Licensed under the Apache License, Version 2.0.
/// <summary>
/// Base class for implementing a long running <see cref="IHostedService"/>.
/// </summary>
public abstract class BackgroundService : IHostedService, IDisposable
{
    private Task _executingTask;
    private readonly CancellationTokenSource _stoppingCts =
                                                    new CancellationTokenSource();

    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);

    public virtual Task StartAsync(CancellationToken cancellationToken)
    {
        // Store the task we're executing
        _executingTask = ExecuteAsync(_stoppingCts.Token);

        // If the task is completed then return it,
        // this will bubble cancellation and failure to the caller
        if (_executingTask.IsCompleted)
        {
            return _executingTask;
        }

        // Otherwise it's running
        return Task.CompletedTask;
    }

    public virtual async Task StopAsync(CancellationToken cancellationToken)
    {
        // Stop called without start
        if (_executingTask == null)
        {
            return;
        }

        try
        {
            // Signal cancellation to the executing method
            _stoppingCts.Cancel();
        }
        finally
        {
            // Wait until the task completes or the stop token triggers
            await Task.WhenAny(_executingTask, Task.Delay(Timeout.Infinite,
                                                          cancellationToken));
        }

    }

    public virtual void Dispose()
    {
```

```
        _stoppingCts.Cancel();
    }
}
```

Al derivar de la clase base abstracta anterior, gracias a esa implementación heredada, solo tiene que implementar el `ExecuteAsync()` método en su propia clase de servicio hospedada personalizada, como en el siguiente código simplificado de eShopOnContainers que sondea una base de datos y publica eventos de integración en event Bus cuando sea necesario.

C#

```
public class GracePeriodManagerService : BackgroundService
{
    private readonly ILogger<GracePeriodManagerService> _logger;
    private readonly OrderingBackgroundSettings _settings;

    private readonly IEventBus _eventBus;

    public GracePeriodManagerService(IOptions<OrderingBackgroundSettings>
settings,
                                     IEventBus eventBus,
                                     ILogger<GracePeriodManagerService> logger)
    {
        // Constructor's parameters validations...
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogDebug($"GracePeriodManagerService is starting.");

        stoppingToken.Register(() =>
            _logger.LogDebug($" GracePeriod background task is stopping."));

        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogDebug($"GracePeriod task doing background work.");

            // This eShopOnContainers method is querying a database table
            // and publishing events into the Event Bus (RabbitMQ / ServiceBus)
            CheckConfirmedGracePeriodOrders();

            try {
                    await Task.Delay(_settings.CheckUpdateTime, stoppingToken);
                }
            catch (TaskCanceledException exception) {
                    _logger.LogCritical(exception, "TaskCanceledException Error",
exception.Message);
                }
        }

        _logger.LogDebug($"GracePeriod background task is stopping.");
    }
```

```
    .../...
}
```

En este caso específico para eShopOnContainers, ejecuta un método de aplicación que consulta una tabla de base de datos que busca pedidos con un estado específico y, al aplicar cambios, está publicando eventos de integración a través del bus de eventos (debajo puede usar RabbitMQ o Azure Service Bus).

Por supuesto, podría ejecutar cualquier otra tarea en segundo plano empresarial, en su lugar.

De forma predeterminada, el token de cancelación se establece con un tiempo de espera de 5 segundos, aunque puede cambiar ese valor cuando construye `WebHost` utilizando la extensión `UseShutdownTimeout` de `IWebHostBuilder`. Esto significa que se espera que nuestro servicio se cancele en un plazo de 5 segundos; de lo contrario, se eliminará más abruptamente.

El código siguiente cambiaría ese tiempo a 10 segundos.

C#

```
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...
```

Diagrama resumen de clases

En la imagen siguiente se muestra un resumen visual de las clases e interfaces implicadas al implementar IHostedServices.

Class diagram with a custom IHostedService and related classes and interfaces

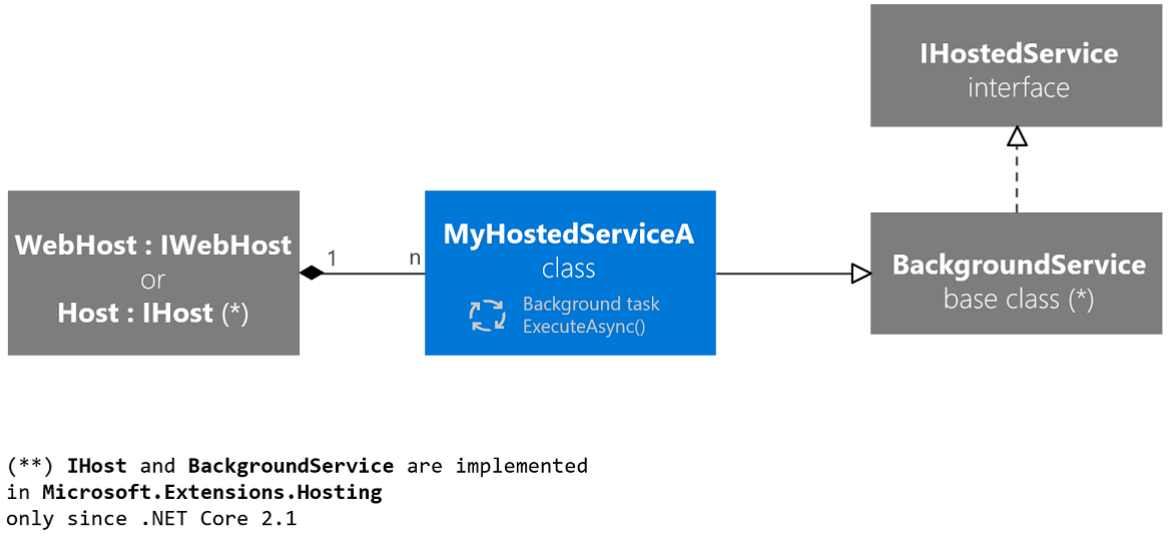


Figura 6-27. Diagrama de clases que muestra las varias clases e interfaces relacionadas con IHostedService

Diagrama de clases: IWebHost e IHost pueden hospedar muchos servicios, que heredan de BackgroundService, que implementa IHostedService.

Consideraciones y conclusiones de la implementación

Es importante tener en cuenta que la forma en que implemente el ASP.NET Core `WebHost` o .NET `Host` podría afectar a la solución final. Por ejemplo, si implementa su `WebHost` en IIS o en un servicio de Azure App Service normal, el host se puede cerrar debido a reciclajes del grupo de aplicaciones. Pero si va a desplegar su servidor como un contenedor en un orquestador como Kubernetes, puede controlar el número asegurado de instancias activas de su servidor. Además, podría considerar otros enfoques en la nube especialmente realizados para estos escenarios, como Azure Functions. Por último, si necesita que el servicio se ejecute todo el tiempo y esté implementando en un servidor de Windows Server, podría usar un servicio de Windows.

Pero incluso para un elemento `WebHost` implementado en un grupo de aplicaciones, hay escenarios, como el relleno o el vaciado de la memoria caché de la aplicación, en los que sería también aplicable.

La `IHostedService` interfaz proporciona una manera cómoda de iniciar tareas en segundo plano en una aplicación web de ASP.NET Core (en .NET Core 2.0 y versiones posteriores) o en cualquier proceso o host (a partir de .NET Core 2.1 con `IHost`). La principal ventaja es la oportunidad de obtener con la cancelación correcta un código de limpieza de sus tareas en segundo plano cuando se está cerrando el propio host.

Recursos adicionales

- **Creación de una tarea programada en ASP.NET Core/Standard 2.0**
  https://blog.maartenballiauw.be/post/2017/08/01/building-a-scheduled-cache-updater-in-aspnet-core-2.html

- **Implementación de IHostedService en ASP.NET Core 2.0**
  https://www.stevejgordon.co.uk/asp-net-core-2-ihostedservice

- **Ejemplo de GenericHost con ASP.NET Core 2.1**
  https://github.com/aspnet/Hosting/tree/release/2.1/samples/GenericHostSample

Siguiente anterior

Implementación de puertas de enlace de API con Ocelot

Artículo • 08/05/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Importante

La aplicación de microservicio de referencia eShopOnContainers usa actualmente las características proporcionadas por Envoy para implementar la puerta de enlace de API en lugar de la Ocelot a la que se hace referencia anteriormente. Hemos elegido este diseño debido a la compatibilidad integrada de Envoy con el protocolo WebSocket, requerido por las nuevas comunicaciones entre servicios gRPC implementadas en eShopOnContainers. Sin embargo, hemos conservado esta sección en la guía para que pueda considerar Ocelot como una puerta de enlace de API sencilla, capaz y ligera adecuada para escenarios de nivel de producción. Además, la versión más reciente de Ocelot contiene un cambio importante en su esquema json. Considere la posibilidad de usar Ocelot < v16.0.0 o use las rutas clave en lugar de ReRoutes.

Diseño y diseño de las puertas de enlace de API

En el diagrama de arquitectura siguiente se muestra cómo se implementaron las puertas de enlace de API con Ocelot en eShopOnContainers.

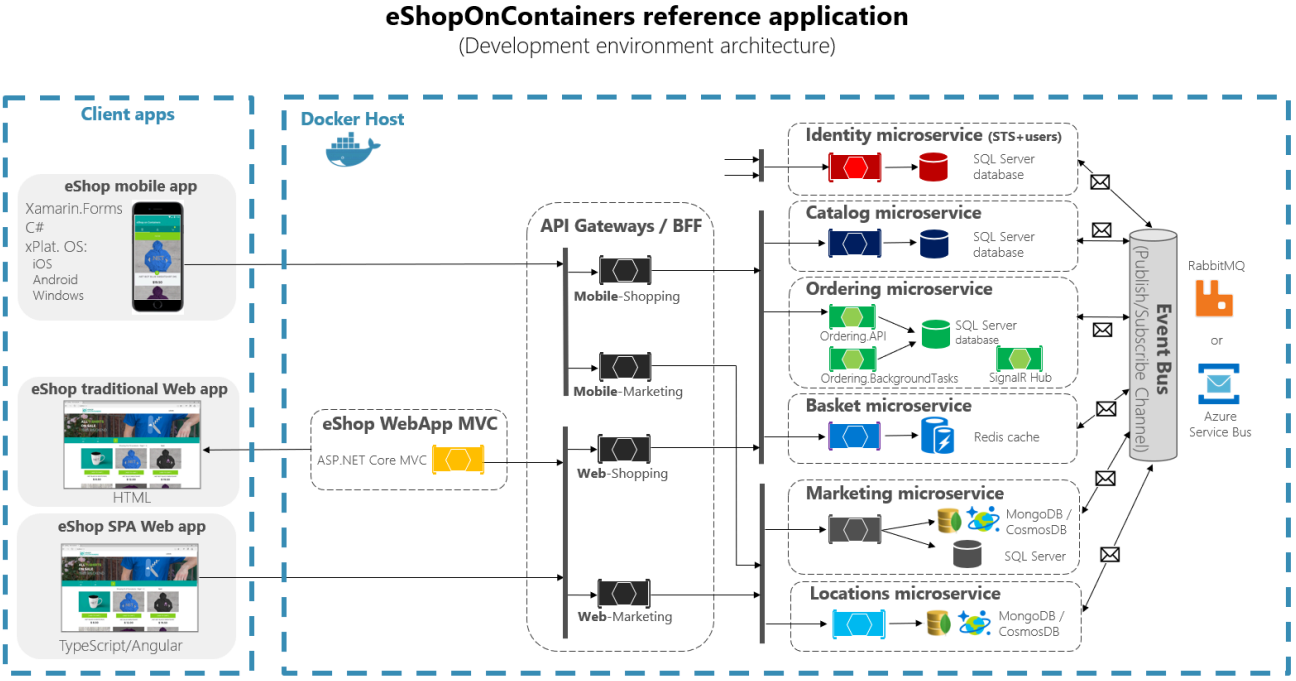


Figura 6-28. Arquitectura de eShopOnContainers con puertas de enlace de API

En ese diagrama se muestra cómo se implementa toda la aplicación en un único host de Docker o en un equipo de desarrollo con "Docker para Windows" o "Docker para Mac". Sin embargo, la implementación en cualquier orquestador sería similar, pero cualquier contenedor del diagrama se podría escalar horizontalmente en el orquestador.

Además, los recursos de infraestructura, como bases de datos, caché y agentes de mensajes, deben descargarse del orquestador e implementarse en sistemas de alta disponibilidad para la infraestructura, como Azure SQL Database, Azure Cosmos DB, Azure Redis, Azure Service Bus o cualquier solución de agrupación en clústeres de alta disponibilidad local.

Como también puede observar en el diagrama, tener varias puertas de enlace de API permite que varios equipos de desarrollo sean autónomos (en este caso, características de marketing frente a características de compras) al desarrollar e implementar sus microservicios más sus propias puertas de enlace de API relacionadas.

Si tuviera una única puerta de enlace de API monolítica que significaría que varios equipos de desarrollo actualizarían un único punto, lo que podría acoplar todos los microservicios con una sola parte de la aplicación.

En ocasiones, ir mucho más lejos en el diseño, una puerta de enlace de API específica también puede limitarse a un único microservicio empresarial en función de la arquitectura elegida. Tener los límites de la puerta de enlace de API dictados por la empresa o el dominio le ayudará a obtener un mejor diseño.

Por ejemplo, la granularidad fina en el nivel de puerta de enlace de API puede ser especialmente útil para aplicaciones de interfaz de usuario compuestas más avanzadas basadas

en microservicios, ya que el concepto de una puerta de enlace de API específica es similar a un servicio de composición de la interfaz de usuario.

Profundizamos en más detalles en la sección anterior Creación de una interfaz de usuario compuesta basada en microservicios.

Como resultado clave, para muchas aplicaciones de tamaño mediano y grande, el uso de un producto de puerta de enlace de API personalizada suele ser un buen enfoque, pero no como un único agregador monolítico o una puerta de enlace de API personalizada única central, a menos que esa puerta de enlace de API permita varias áreas de configuración independientes para los distintos equipos de desarrollo que crean microservicios autónomos.

Microservicios o contenedores de ejemplo para volver a enrutar a través de las puertas de enlace de API

Por ejemplo, eShopOnContainers tiene alrededor de seis tipos de microservicio internos que deben publicarse a través de las puertas de enlace de API, como se muestra en la siguiente imagen.

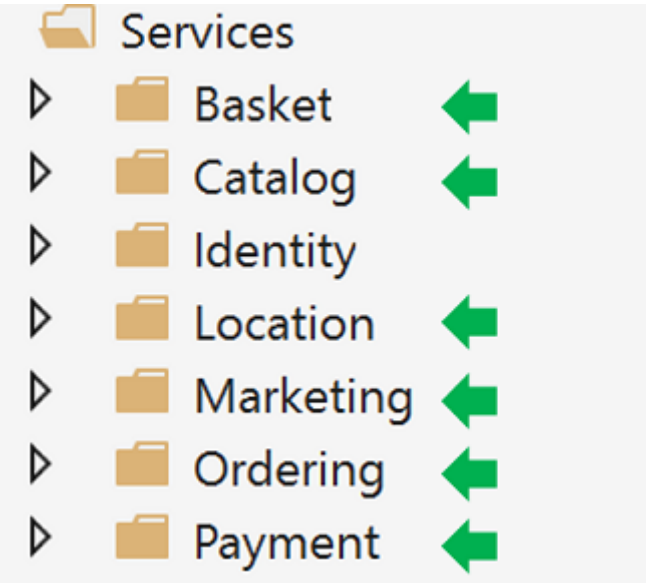


Figura 6-29. Carpetas de microservicios en la solución eShopOnContainers en Visual Studio

Acerca del servicio de identidad, en el diseño se deja fuera del enrutamiento de la puerta de enlace de API porque es la única preocupación transversal en el sistema, aunque con Ocelot también es posible incluirlo como parte de las listas de reenrutamiento.

Todos esos servicios se implementan actualmente como ASP.NET servicios core web API, como puede indicar desde el código. Vamos a centrarnos en uno de los microservicios, como el código de microservicio de catálogo.

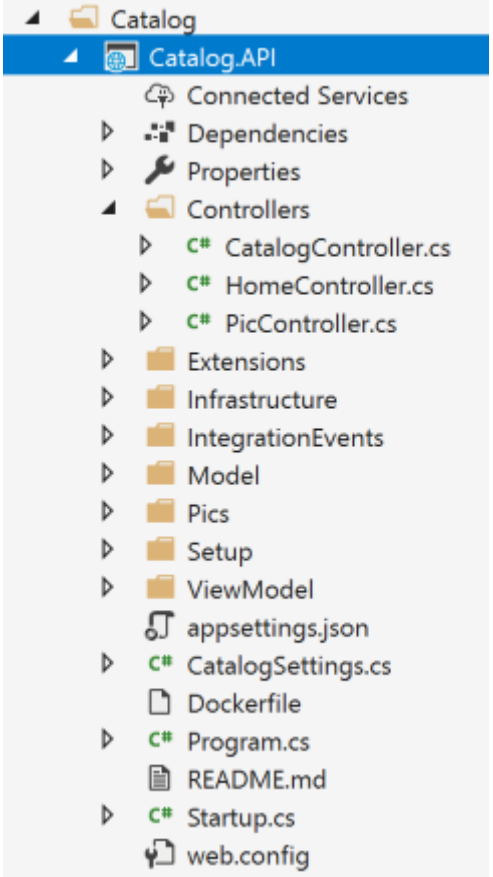


Figura 6-30. Microservicio de API web de ejemplo (microservicio de catálogo)

Puede ver que el microservicio Catalog es un proyecto típico de API web de ASP.NET Core con varios controladores y métodos como en el código siguiente.

C#

```
[HttpGet]
[Route("items/{id:int}")]
[ProducesResponseType((int)HttpStatusCode.BadRequest)]
[ProducesResponseType((int)HttpStatusCode.NotFound)]
[ProducesResponseType(typeof(CatalogItem),(int)HttpStatusCode.OK)]
public async Task<IActionResult> GetItemById(int id)
{
    if (id <= 0)
    {
        return BadRequest();
    }
    var item = await _catalogContext.CatalogItems.
                                        SingleOrDefaultAsync(ci => ci.Id == id);
    //…

    if (item != null)
    {
        return Ok(item);
    }
    return NotFound();
}
```

La solicitud HTTP terminará ejecutando ese tipo de código de C# que accede a la base de datos de microservicios y cualquier acción necesaria adicional.

Con respecto a la dirección URL del microservicio, cuando los contenedores se implementan en el equipo de desarrollo local (host de Docker local), cada contenedor de microservicios siempre tiene un puerto interno (normalmente el puerto 80) especificado en su dockerfile, como en el siguiente dockerfile:

Dockerfile

```
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS base
WORKDIR /app
EXPOSE 80
```

El puerto 80 que se muestra en el código es interno dentro del host de Docker, por lo que las aplicaciones cliente no pueden acceder a él.

Las aplicaciones cliente solo pueden acceder a los puertos externos (si los hay) publicados al implementar con `docker-compose`.

Esos puertos externos no se deben publicar al implementar en un entorno de producción. Por este motivo específico, por qué desea usar la puerta de enlace de API para evitar la comunicación directa entre las aplicaciones cliente y los microservicios.

Sin embargo, al desarrollar, quiere acceder directamente al microservicio o contenedor y ejecutarlo a través de Swagger. Por eso, en eShopOnContainers, los puertos externos se siguen especificando incluso cuando la puerta de enlace de API o las aplicaciones cliente no las usarán.

Este es un ejemplo del `docker-compose.override.yml` archivo para el microservicio Catalog:

yml

```yml
catalog-api:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://0.0.0.0:80
    - ConnectionString=YOUR_VALUE
    - ... Other Environment Variables
  ports:
    - "5101:80"   # Important: In a production environment you should remove the
external port (5101) kept here for microservice debugging purposes.
                  # The API Gateway redirects and access through the internal port
(80).
```

Puede ver cómo en la configuración de docker-compose.override.yml el puerto interno del contenedor catálogo es el puerto 80, pero el puerto para el acceso externo es 5101. Pero la aplicación no debe usar este puerto cuando se usa una puerta de enlace de API, solo para depurar, ejecutar y probar solo el microservicio catalog.

Normalmente, no se implementará con docker-compose en un entorno de producción porque el entorno de implementación de producción adecuado para microservicios es un orquestador como Kubernetes o Service Fabric. Al implementar en esos entornos, use archivos de configuración diferentes en los que no publique directamente ningún puerto externo para los microservicios, pero siempre usará el proxy inverso desde la puerta de enlace de API.

Ejecute el microservicio de catálogo en el host de Docker local. Ejecute la solución completa eShopOnContainers desde Visual Studio (ejecuta todos los servicios de los archivos docker-compose) o inicie el microservicio Catalog con el siguiente comando docker-compose en CMD o PowerShell situado en la carpeta donde se colocan y `docker-compose.yml` `docker-compose.override.yml` .

Consola

```
docker-compose run --service-ports catalog-api
```

Este comando solo ejecuta el contenedor del servicio catalog-api más las dependencias especificadas en el docker-compose.yml. En este caso, el contenedor de SQL Server y el contenedor RabbitMQ.

A continuación, puede acceder directamente al microservicio Catalog y ver sus métodos a través de la interfaz de usuario de Swagger que accede directamente a través de ese puerto "externo", en este caso `http://host.docker.internal:5101/swagger`:

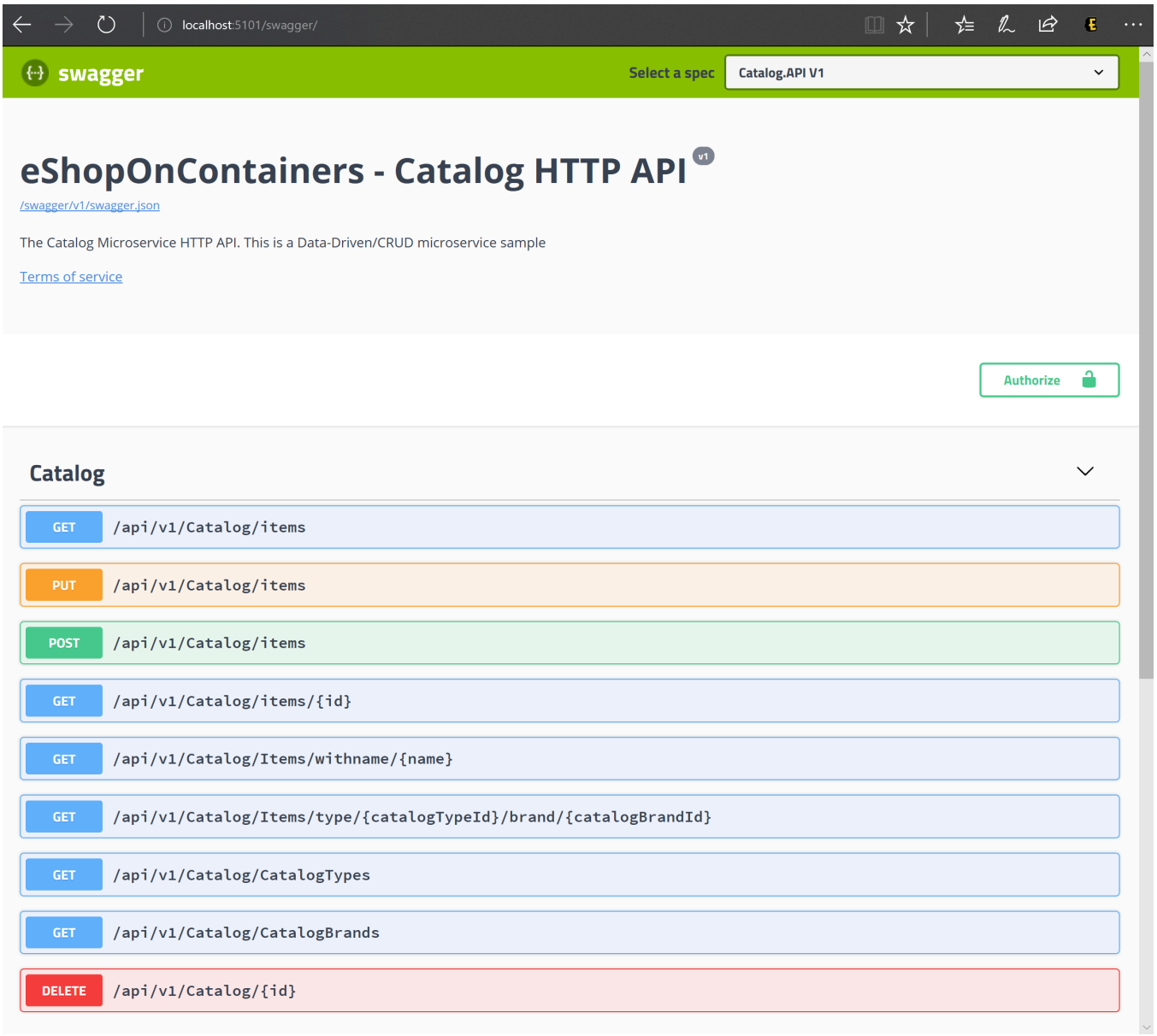


Figura 6-31. Probar el microservicio de catálogo con su interfaz de usuario de Swagger

En este punto, podría establecer un punto de interrupción en el código de C# en Visual Studio, probar el microservicio con los métodos expuestos en la interfaz de usuario de Swagger y, por último, limpiar todo con el `docker-compose down` comando .

Sin embargo, la comunicación de acceso directo al microservicio, en este caso a través del puerto externo 5101, es precisamente lo que desea evitar en la aplicación. Y puede evitarlo estableciendo el nivel adicional de direccionamiento indirecto de la puerta de enlace de API (Ocelot, en este caso). De este modo, la aplicación cliente no accederá directamente al microservicio.

Implementación de las puertas de enlace de API con Ocelot

Ocelot es básicamente un conjunto de middleware que se puede aplicar en un orden específico.

Ocelot está diseñado para trabajar solo con ASP.NET Core. La versión más reciente del paquete es 18.0, que tiene como destino .NET 6 y, por tanto, no es adecuada para las aplicaciones de .NET Framework.

Puede instalar Ocelot y sus dependencias en el proyecto de ASP.NET Core con el paquete NuGet de Ocelot, desde Visual Studio.

PowerShell

```
Install-Package Ocelot
```

En eShopOnContainers, su implementación de PUERTA de enlace de API es un proyecto sencillo ASP.NET Core WebHost y el middleware de Ocelot controla todas las características de la puerta de enlace de API, como se muestra en la siguiente imagen:

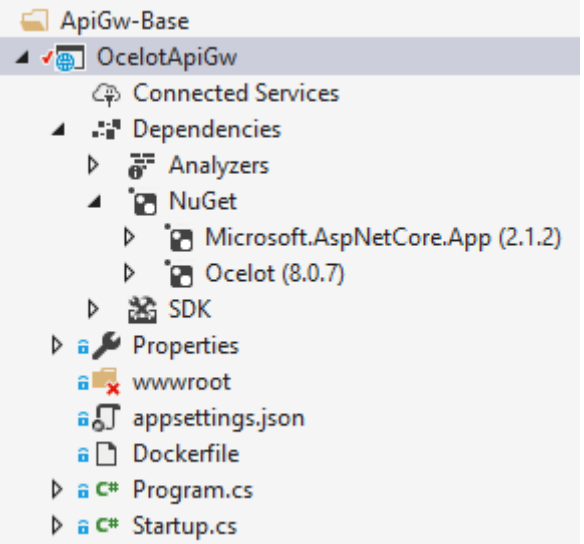


Figura 6-32. El proyecto base OcelotApiGw en eShopOnContainers

Este proyecto ASP.NET Core WebHost se compila con dos archivos simples: `Program.cs` y `Startup.cs`.

El Program.cs solo tiene que crear y configurar el ASP.NET Core BuildWebHost típico.

C#

```
namespace OcelotApiGw
{
    public class Program
    {
        public static void Main(string[] args)
        {
```

```
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args)
        {
            var builder = WebHost.CreateDefaultBuilder(args);

            builder.ConfigureServices(s => s.AddSingleton(builder))
                    .ConfigureAppConfiguration(
                          ic => ic.AddJsonFile(Path.Combine("configuration",

"configuration.json")))
                    .UseStartup<Startup>();
            var host = builder.Build();
            return host;
        }
    }
}
```

El punto importante aquí para Ocelot es el `configuration.json` archivo que debe proporcionar al generador a través del `AddJsonFile()` método . Aquí `configuration.json` es donde se especifican todos los reroutes de puerta de enlace de API, lo que significa que los puntos de conexión externos con puertos específicos y los puntos de conexión internos correlacionados suelen usar puertos diferentes.

JSON

```
{
    "ReRoutes": [],
    "GlobalConfiguration": {}
}
```

Hay dos secciones para la configuración. Matriz de ReRoutes y GlobalConfiguration. ReRoutes son los objetos que indican a Ocelot cómo tratar una solicitud ascendente. La configuración global permite invalidaciones de la configuración específica de ReRoute. Resulta útil si no quiere administrar una gran cantidad de configuraciones específicas de ReRoute.

Este es un ejemplo simplificado del archivo de configuración de ReRoute de una de las puertas de enlace de API de eShopOnContainers.

JSON

```
{
  "ReRoutes": [
    {
      "DownstreamPathTemplate": "/api/{version}/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
```

```
        {
          "Host": "catalog-api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/c/{everything}",
      "UpstreamHttpMethod": [ "POST", "PUT", "GET" ]
    },
    {
      "DownstreamPathTemplate": "/api/{version}/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "basket-api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/b/{everything}",
      "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "AllowedScopes": []
      }
    }

  ],
    "GlobalConfiguration": {
      "RequestIdKey": "OcRequestId",
      "AdministrationPath": "/administration"
    }
  }
```

La funcionalidad principal de una puerta de enlace de API de Ocelot es tomar solicitudes HTTP entrantes y reenviarlas a un servicio de bajada, actualmente como otra solicitud HTTP. Ocelot describe el enrutamiento de una solicitud a otra como ReRoute.

Por ejemplo, vamos a centrarnos en uno de los reroutes de la configuration.json anterior, la configuración del microservicio Basket.

JSON

```
{
      "DownstreamPathTemplate": "/api/{version}/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "basket-api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/b/{everything}",
      "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
```

```
        "AuthenticationOptions": {
          "AuthenticationProviderKey": "IdentityApiKey",
          "AllowedScopes": []
        }
  }
```

Las propiedades DownstreamPathTemplate, Scheme y DownstreamHostAndPorts realizan la dirección URL interna del microservicio a la que se reenviará esta solicitud.

El puerto es el puerto interno que usa el servicio. Al usar contenedores, el puerto especificado en su dockerfile.

`Host` es un nombre de servicio que depende de la resolución de nombres de servicio que está usando. Al usar docker-compose, el host de Docker proporciona los nombres de los servicios, que usa los nombres de servicio proporcionados en los archivos docker-compose. Si usa un orquestador como Kubernetes o Service Fabric, ese nombre se debe resolver mediante la resolución dns o nombre proporcionada por cada orquestador.

DownstreamHostAndPorts es una matriz que contiene el host y el puerto de los servicios de bajada a los que desea reenviar las solicitudes. Normalmente, esta configuración solo contendrá una entrada, pero a veces es posible que quiera equilibrar la carga de las solicitudes a los servicios de bajada y Ocelot le permite agregar más de una entrada y, a continuación, seleccionar un equilibrador de carga. Pero si usa Azure y cualquier orquestador, probablemente sea una mejor idea equilibrar la carga con la infraestructura de la nube y del orquestador.

UpstreamPathTemplate es la dirección URL que Ocelot usará para identificar qué DownstreamPathTemplate se usará para una solicitud determinada del cliente. Por último, se usa UpstreamHttpMethod para que Ocelot pueda distinguir entre diferentes solicitudes (GET, POST, PUT) a la misma dirección URL.

En este momento, podría tener una única puerta de enlace de API de Ocelot (ASP.NET Core WebHost) mediante uno o varios archivos de configuration.json combinados o también puede almacenar la configuración en un almacén de KV de Consul.

Pero como se introdujo en las secciones de arquitectura y diseño, si realmente desea tener microservicios autónomos, podría ser mejor dividir esa puerta de enlace de API monolítica única en varias puertas de enlace de API o BFF (back-end para front-end). Para ello, veamos cómo implementar ese enfoque con contenedores de Docker.

Uso de una sola imagen de contenedor de Docker para ejecutar varios tipos de contenedor de API Gateway o BFF diferentes

En eShopOnContainers, usamos una sola imagen de contenedor de Docker con la puerta de enlace de API de Ocelot, pero, en tiempo de ejecución, creamos diferentes servicios o contenedores para cada tipo de API-Gateway/BFF proporcionando un archivo de configuration.json diferente, mediante un volumen de Docker para acceder a una carpeta de PC diferente para cada servicio.

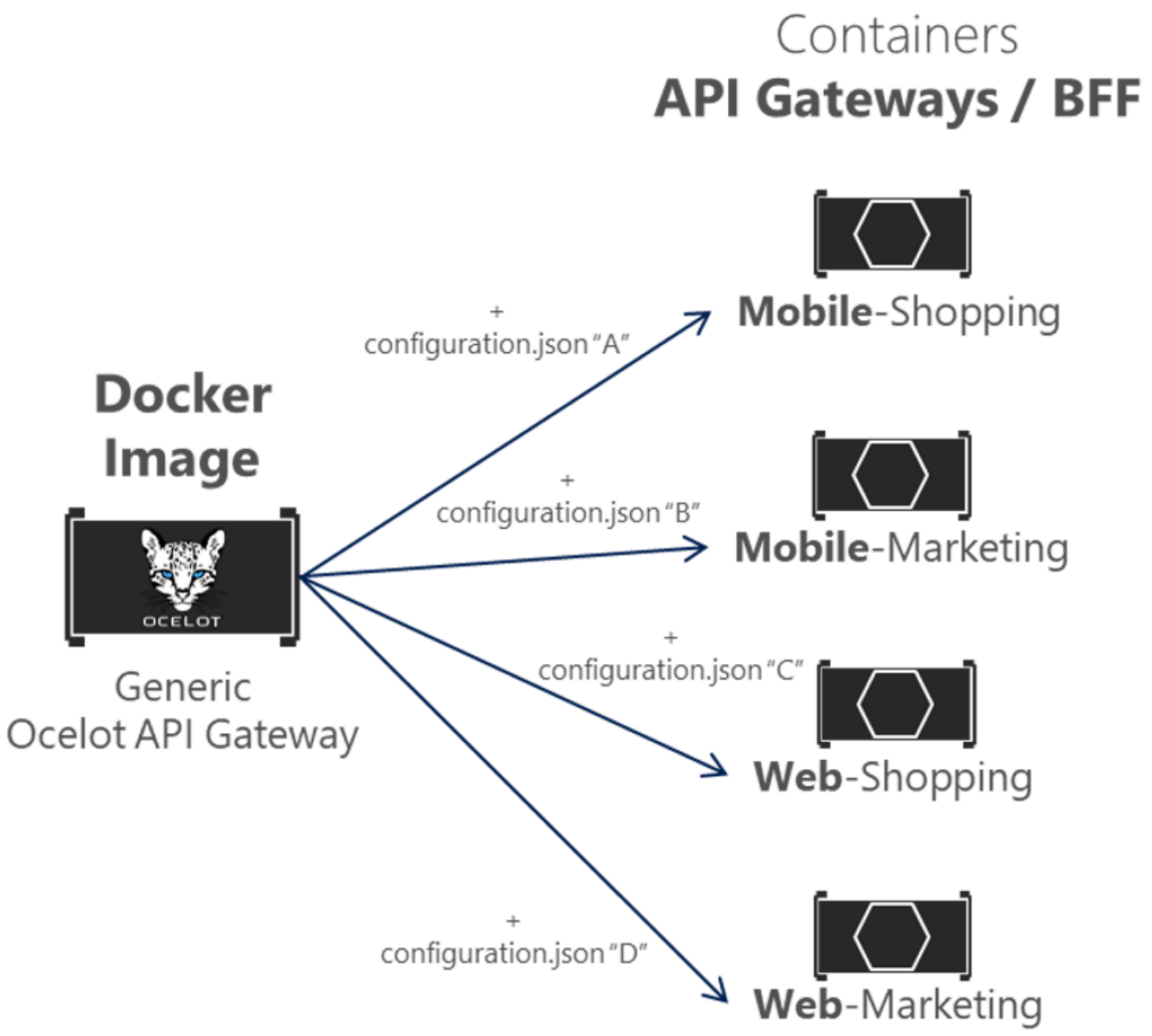


Figura 6-33. Reutilización de una sola imagen de Docker de Ocelot en varios tipos de puerta de enlace de API

En eShopOnContainers, se crea la "imagen genérica de Docker de puerta de enlace de API de Ocelot" con el proyecto denominado "OcelotApiGw" y el nombre de imagen "eshop/ocelotapigw" especificado en el archivo docker-compose.yml. A continuación, al implementar en Docker, habrá cuatro contenedores API-Gateway creados a partir de esa misma imagen de Docker, como se muestra en la siguiente extracción del archivo docker-compose.yml.

```yml
  mobileshoppingapigw:
    image: eshop/ocelotapigw:${TAG:-latest}
    build:
      context: .
      dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

  mobilemarketingapigw:
    image: eshop/ocelotapigw:${TAG:-latest}
    build:
      context: .
      dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

  webshoppingapigw:
    image: eshop/ocelotapigw:${TAG:-latest}
    build:
      context: .
      dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

  webmarketingapigw:
    image: eshop/ocelotapigw:${TAG:-latest}
    build:
      context: .
      dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile
```

Además, como puede ver en el siguiente archivo de docker-compose.override.yml, la única diferencia entre esos contenedores de puerta de enlace de API es el archivo de configuración de Ocelot, que es diferente para cada contenedor de servicios y se especifica en tiempo de ejecución a través de un volumen de Docker.

```yml
mobileshoppingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5200:80"
  volumes:
    - ./src/ApiGateways/Mobile.Bff.Shopping/apigw:/app/configuration

mobilemarketingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5201:80"
  volumes:
    - ./src/ApiGateways/Mobile.Bff.Marketing/apigw:/app/configuration

webshoppingapigw:
```

```
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - IdentityUrl=http://identity-api
    ports:
      - "5202:80"
    volumes:
      - ./src/ApiGateways/Web.Bff.Shopping/apigw:/app/configuration

  webmarketingapigw:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - IdentityUrl=http://identity-api
    ports:
      - "5203:80"
    volumes:
      - ./src/ApiGateways/Web.Bff.Marketing/apigw:/app/configuration
```

Debido a ese código anterior y, como se muestra en el Explorador de Visual Studio siguiente, el único archivo necesario para definir cada puerta de enlace de API o negocio específica es solo un archivo configuration.json, ya que las cuatro puertas de enlace de API se basan en la misma imagen de Docker.

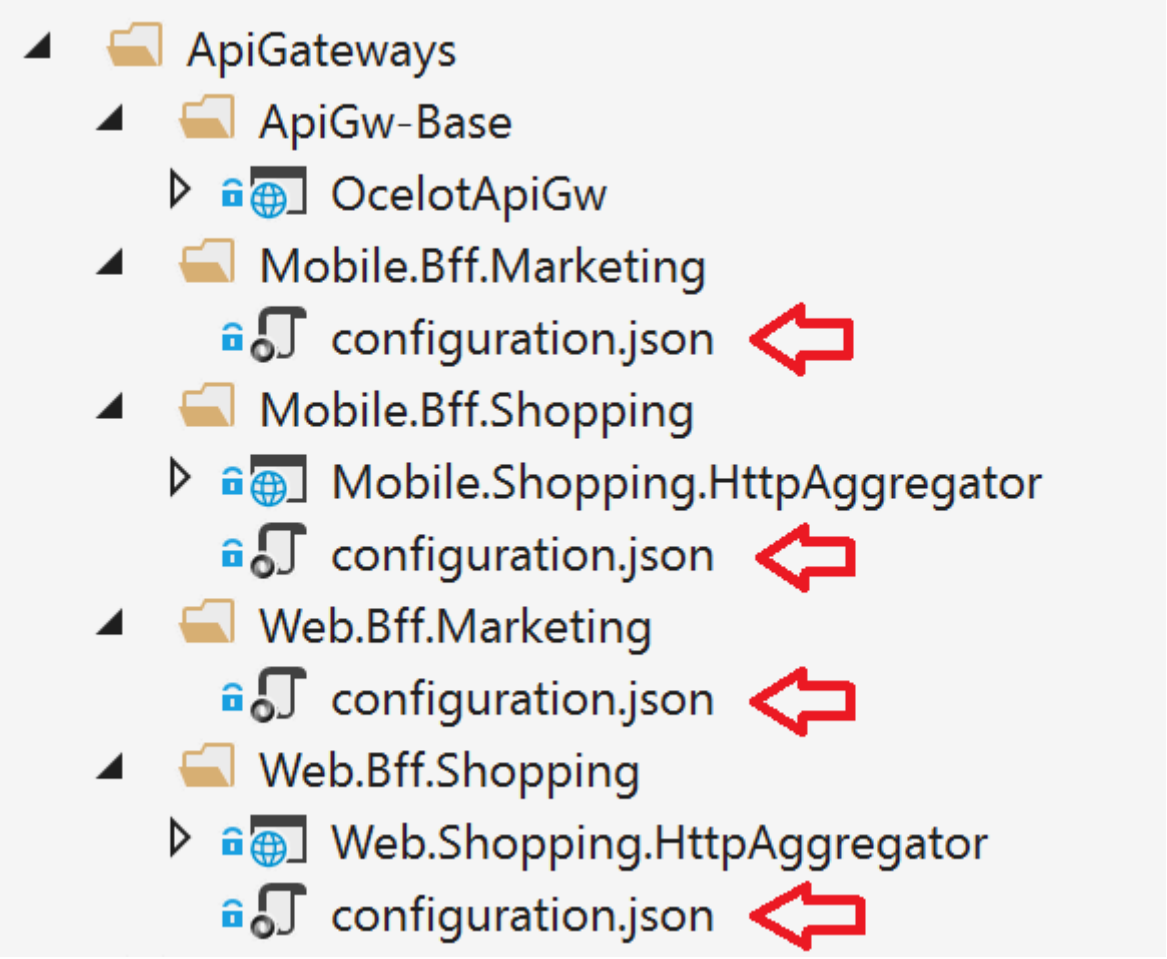


Figura 6-34. El único archivo necesario para definir cada puerta de enlace de API o BFF con Ocelot es un archivo de configuración.

Al dividir la puerta de enlace de API en varias puertas de enlace de API, los distintos equipos de desarrollo que se centran en distintos subconjuntos de microservicios pueden administrar sus propias puertas de enlace de API mediante archivos de configuración independientes de Ocelot. Además, al mismo tiempo pueden reutilizar la misma imagen de Docker de Ocelot.

Ahora, si ejecuta eShopOnContainers con las puertas de enlace de API (incluidas de forma predeterminada en VS al abrir eShopOnContainers-ServicesAndWebApps.sln solución o si ejecuta "docker-compose up"), se realizarán las rutas de ejemplo siguientes.

Por ejemplo, al visitar la dirección URL `http://host.docker.internal:5202/api/v1/c/catalog/items/2/` ascendente que sirve la puerta de enlace de API webshoppingapigw, obtendrá el mismo resultado de la dirección URL `http://catalog-api/api/v1/2` de bajada interna dentro del host de Docker, como en el explorador siguiente.

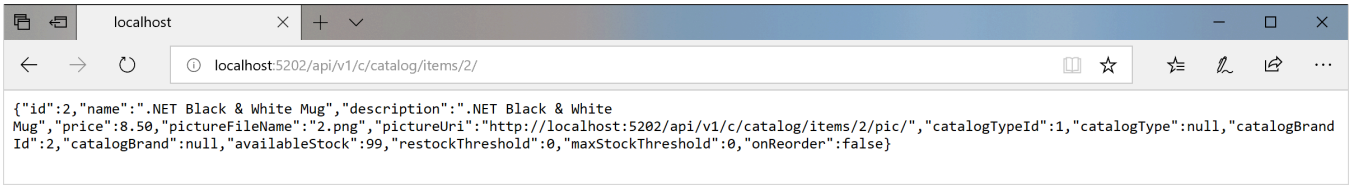


Figura 6-35. Acceso a un microservicio a través de una dirección URL proporcionada por la puerta de enlace de API

Debido a los motivos de prueba o depuración, si desea acceder directamente al contenedor de Docker de catálogo (solo en el entorno de desarrollo) sin pasar por la puerta de enlace de API, ya que "catalog-api" es una resolución DNS interna para el host de Docker (detección de servicios controlada por nombres de servicio docker-compose), la única manera de acceder directamente al contenedor es a través del puerto externo publicado en la docker-compose.override.yml, que solo se proporciona para las pruebas de desarrollo, como `http://host.docker.internal:5101/api/v1/Catalog/items/1` en el explorador siguiente.

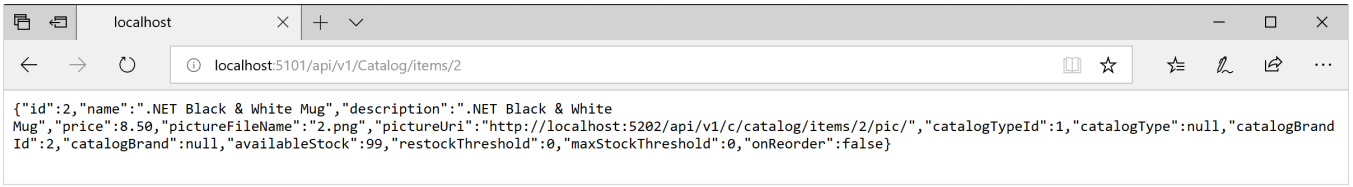


Figura 6-36. Acceso directo a un microservicio con fines de prueba

Pero la aplicación está configurada para que acceda a todos los microservicios a través de las puertas de enlace de API, no a través del puerto directo "accesos directos".

Patrón de agregación de puerta de enlace en eShopOnContainers

Como se ha introducido anteriormente, una manera flexible de implementar la agregación de solicitudes es con servicios personalizados, por código. La manera seleccionada de implementar la agregación en eShopOnContainers es con un servicio de API web principal de ASP.NET explícito para cada agregador.

Según ese enfoque, el diagrama de composición de la puerta de enlace de API se extiende un poco más al considerar los servicios agregadores que no se muestran en el diagrama de arquitectura global simplificado mostrado anteriormente.

En el diagrama siguiente, también puede ver cómo funcionan los servicios agregadores con sus puertas de enlace de API relacionadas.

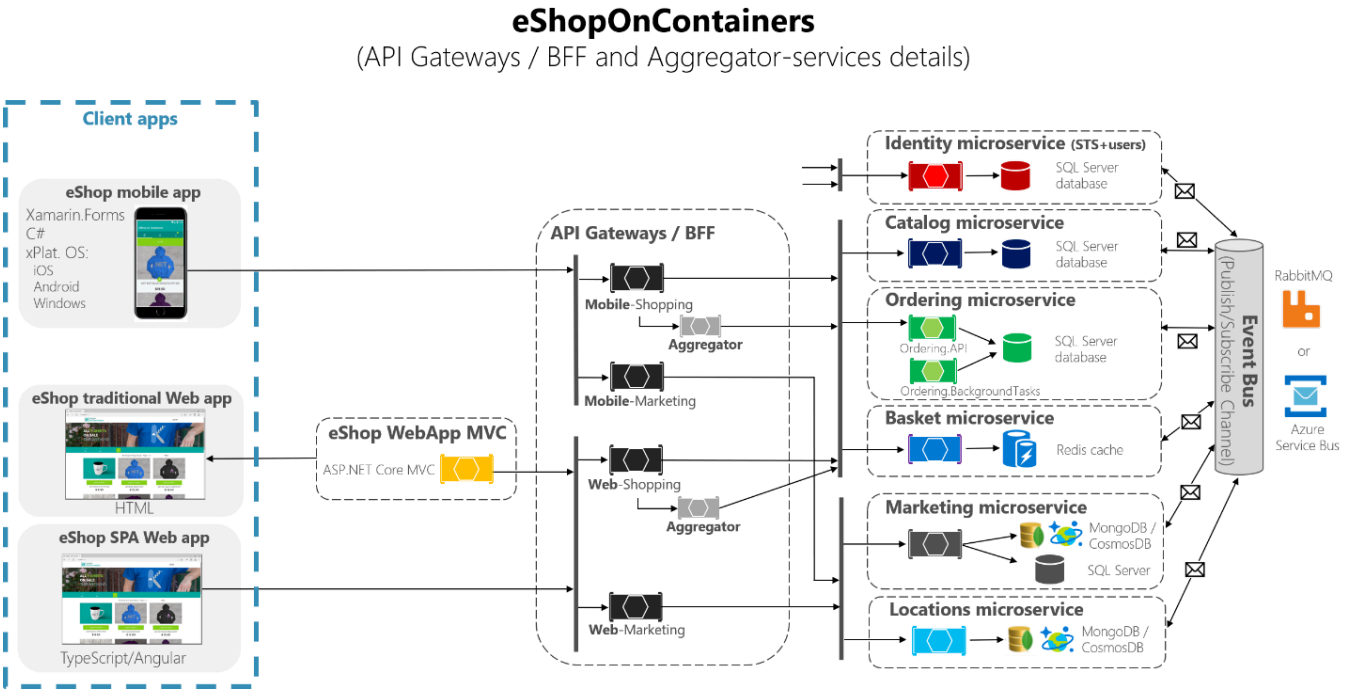


Figura 6-37. Arquitectura de eShopOnContainers con servicios de agregador

Al ampliar aún más, en el área empresarial "Shopping" de la siguiente imagen, puede ver que la chattiidad entre las aplicaciones cliente y los microservicios se reduce al usar los servicios agregadores en las puertas de enlace de API.

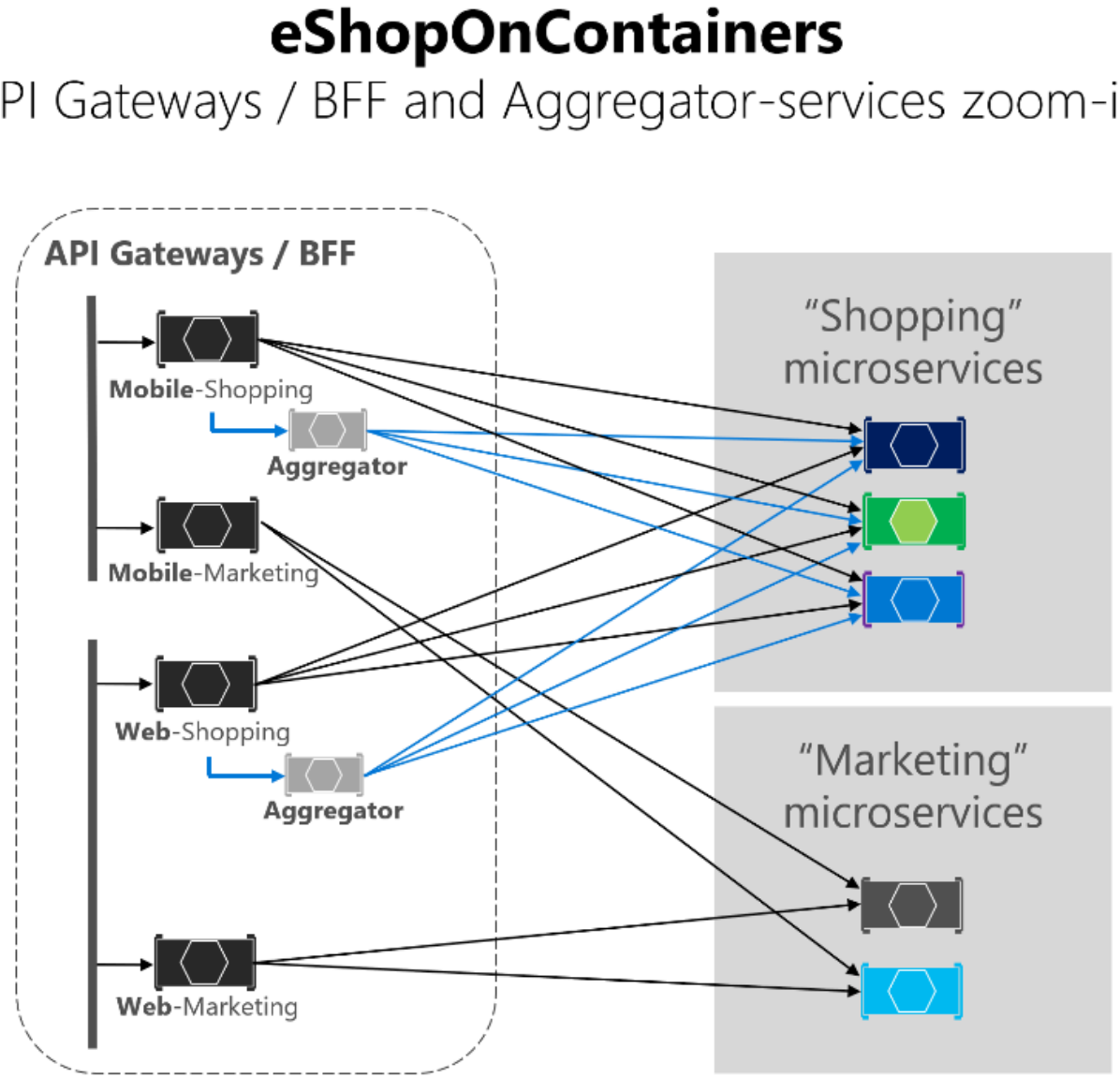


Figura 6-38. Acercar la visión de los servicios agregadores

Puede observar cómo cuando el diagrama muestra las posibles solicitudes procedentes de las puertas de enlace de API, puede ser compleja. Por otro lado, al usar el patrón de agregador, puede ver cómo las flechas en azul simplificarían la comunicación desde una perspectiva de la aplicación cliente. Este patrón no solo ayuda a reducir el chattiness y la latencia en la comunicación, sino que también mejora significativamente la experiencia del usuario para las aplicaciones remotas (aplicaciones móviles y SPA).

En el caso del área de negocio "Marketing" y los microservicios, es un caso de uso simple, por lo que no era necesario usar agregadores, pero también podría ser posible, si es necesario.

Autenticación y autorización en puertas de enlace de API de Ocelot

En una puerta de enlace de API de Ocelot, puede sentar el servicio de autenticación, como un servicio de API web principal de ASP.NET mediante IdentityServer que proporciona el token de autenticación, ya sea fuera o dentro de la puerta de enlace de API.

Dado que eShopOnContainers usa varias puertas de enlace de API con límites basados en BFF y áreas empresariales, el servicio Identity/Auth se deja fuera de las puertas de enlace de API, como se resalta en amarillo en el diagrama siguiente.

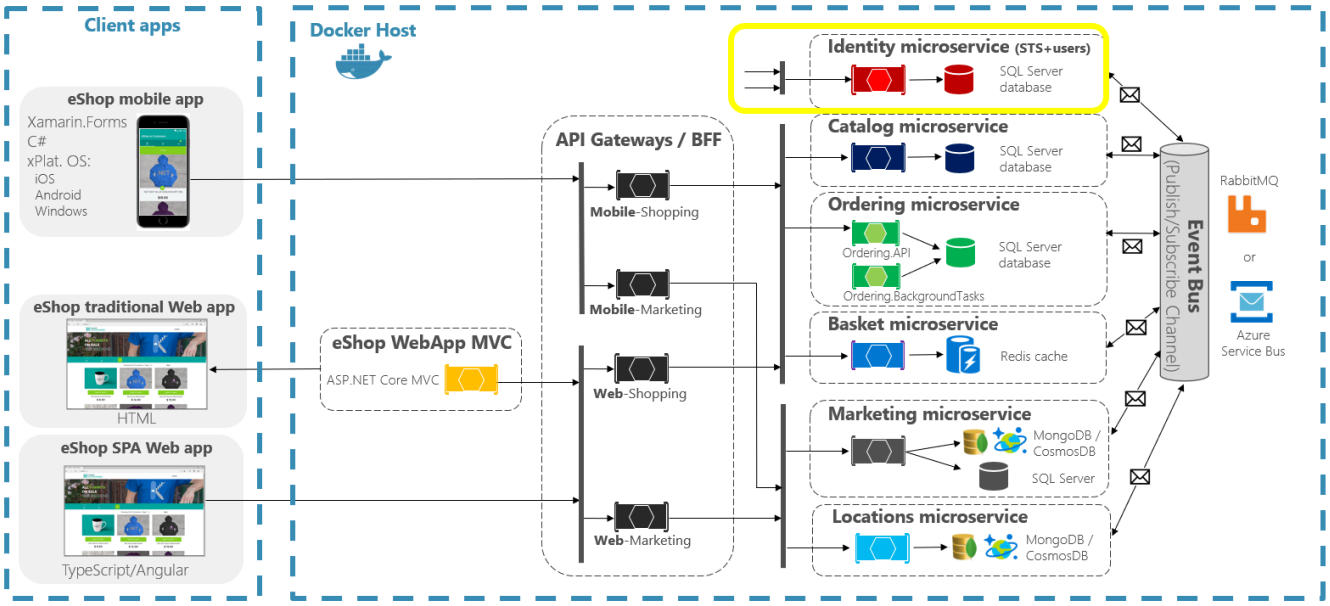


Figura 6-39. Posición del servicio de identidad en eShopOnContainers

Sin embargo, Ocelot también admite el microservicio Identity/Auth dentro del límite de la puerta de enlace de API, como en este otro diagrama.

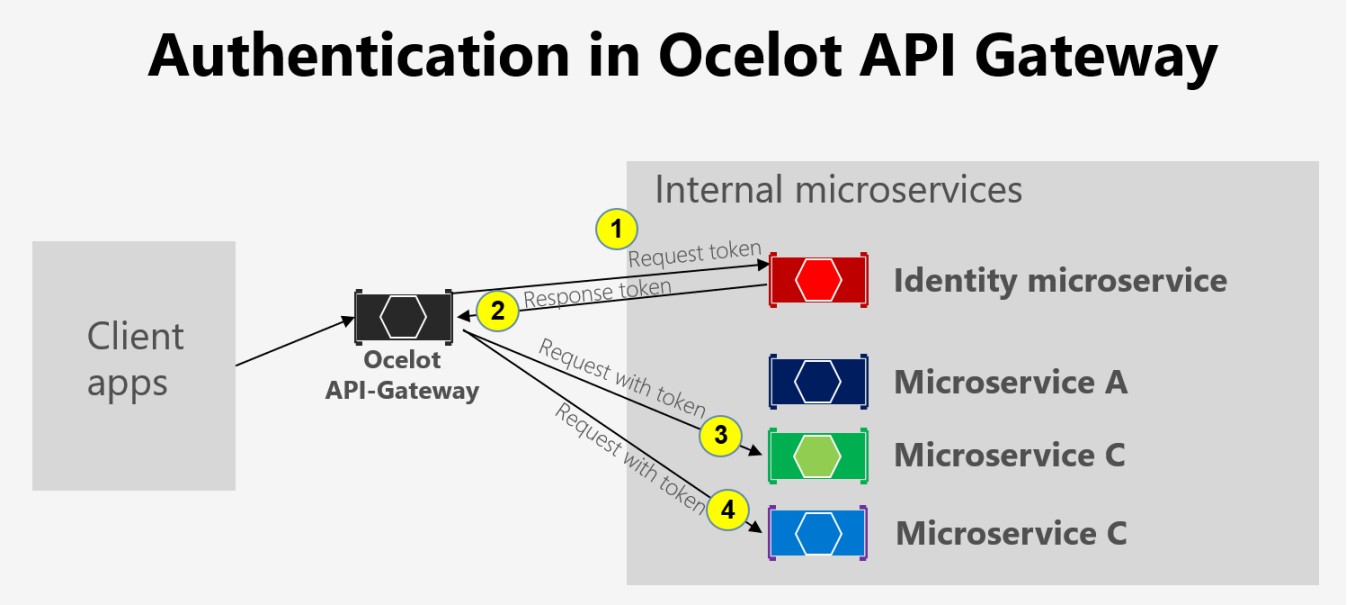


Figura 6-40. Autenticación en Ocelot

Como se muestra en el diagrama anterior, cuando el microservicio Identity está debajo de la puerta de enlace de API (AG): 1) AG solicita un token de autenticación del microservicio de

identidad, 2) El microservicio de identidad devuelve el token al grupo de disponibilidad, 3-4) solicitudes de AG desde microservicios mediante el token de autenticación. Dado que la aplicación eShopOnContainers ha dividido la puerta de enlace de API en varias instancias de BFF (back-end para front-end) y puertas de enlace de API de áreas de negocio, otra opción habría sido crear una puerta de enlace de API adicional para cuestiones transversales. Esa opción sería justa en una arquitectura más compleja basada en microservicios con varios microservicios transversales. Dado que solo hay una preocupación transversal en eShopOnContainers, se decidió simplemente controlar el servicio de seguridad fuera del dominio de la puerta de enlace de API, por motivos de simplicidad.

En cualquier caso, si la aplicación está protegida en el nivel de puerta de enlace de API, el módulo de autenticación de la puerta de enlace de API de Ocelot se visita al principio al intentar usar cualquier microservicio protegido. Que redirige la solicitud HTTP para visitar el microservicio identidad o autenticación para obtener el token de acceso para que pueda visitar los servicios protegidos con el access_token.

La forma de proteger con la autenticación de cualquier servicio en el nivel de puerta de enlace de API es estableciendo AuthenticationProviderKey en su configuración relacionada en el configuration.json.

JSON

```
{
  "DownstreamPathTemplate": "/api/{version}/{everything}",
  "DownstreamScheme": "http",
  "DownstreamHostAndPorts": [
    {
      "Host": "basket-api",
      "Port": 80
    }
  ],
  "UpstreamPathTemplate": "/api/{version}/b/{everything}",
  "UpstreamHttpMethod": [],
  "AuthenticationOptions": {
    "AuthenticationProviderKey": "IdentityApiKey",
    "AllowedScopes": []
  }
}
```

Cuando se ejecute Ocelot, examinará La autenticación de ReRoutesOptions.AuthenticationProviderKey y comprobará que hay un proveedor de autenticación registrado con la clave especificada. Si no lo hay, Ocelot no se iniciará. Si lo hay, ReRoute usará ese proveedor cuando se ejecute.

Dado que Ocelot WebHost está configurado con `authenticationProviderKey = "IdentityApiKey"`, que requerirá autenticación siempre que ese servicio tenga solicitudes sin

ningún token de autenticación.

C#

```
namespace OcelotApiGw
{
    public class Startup
    {
        private readonly IConfiguration _cfg;

        public Startup(IConfiguration configuration) => _cfg = configuration;

        public void ConfigureServices(IServiceCollection services)
        {
            var identityUrl = _cfg.GetValue<string>("IdentityUrl");
            var authenticationProviderKey = "IdentityApiKey";
                        //…
            services.AddAuthentication()
                .AddJwtBearer(authenticationProviderKey, x =>
                {
                    x.Authority = identityUrl;
                    x.RequireHttpsMetadata = false;
                    x.TokenValidationParameters = new
Microsoft.IdentityModel.Tokens.TokenValidationParameters()
                    {
                        ValidAudiences = new[] { "orders", "basket", "locations",
"marketing", "mobileshoppingagg", "webshoppingagg" }
                    };
                });
            //...
        }
    }
}
```

A continuación, también debe establecer la autorización con el atributo [Authorize] en cualquier recurso al que se acceda como los microservicios, como en el siguiente controlador de microservicios Basket.

C#

```
namespace Microsoft.eShopOnContainers.Services.Basket.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class BasketController : Controller
    {
      //...
    }
}
```

Las validAudiences como "basket" se correlacionan con la audiencia definida en cada microservicio con `AddJwtBearer()` en configureServices() de la clase Startup, como en el código siguiente.

C#

```
// prevent from mapping "sub" claim to nameidentifier.
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

var identityUrl = Configuration.GetValue<string>("IdentityUrl");

services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>
{
    options.Authority = identityUrl;
    options.RequireHttpsMetadata = false;
    options.Audience = "basket";
});
```

Si intenta acceder a cualquier microservicio protegido, como el microservicio Basket con una dirección URL de ReRoute basada en la puerta de enlace de API como `http://host.docker.internal:5202/api/v1/b/basket/1`, obtendrá un 401 No autorizado a menos que proporcione un token válido. Por otro lado, si se autentica una dirección URL de ReRoute, Ocelot invocará cualquier esquema de bajada asociado a él (la dirección URL del microservicio interno).

Autorización en el nivel ReRoutes de Ocelot. Ocelot admite la autorización basada en notificaciones evaluada después de la autenticación. Para establecer la autorización en un nivel de ruta, agregue las siguientes líneas a la configuración de ReRoute.

JSON

```
"RouteClaimsRequirement": {
    "UserType": "employee"
}
```

En ese ejemplo, cuando se llama al middleware de autorización, Ocelot encontrará si el usuario tiene el tipo de notificación "UserType" en el token y si el valor de esa notificación es "employee". Si no es así, el usuario no estará autorizado y la respuesta será 403 prohibido.

Uso de entradas de Kubernetes más puertas de enlace de API de Ocelot

Cuando se usa Kubernetes (como en un clúster de Azure Kubernetes Service), normalmente se unifican todas las solicitudes HTTP a través del nivel de entrada de Kubernetes basado en *Nginx*.

En Kubernetes, si no usa ningún enfoque de entrada, los servicios y pods solo tienen direcciones IP enrutables por la red del clúster.

Pero si usa un enfoque de entrada, tendrá un nivel intermedio entre Internet y los servicios (incluidas las puertas de enlace de API), que actúa como proxy inverso.

Como definición, una entrada es una colección de reglas que permiten que las conexiones entrantes lleguen a los servicios de clúster. Se configura una entrada para proporcionar direcciones URL accesibles externamente a los servicios, equilibrar la carga del tráfico, la terminación SSL y mucho más. Los usuarios solicitan la entrada posTing the Ingress resource to the API server(PosTing the Ingress resource to the API server).

En eShopOnContainers, al desarrollar localmente y usar solo la máquina de desarrollo como host de Docker, no se usa ninguna entrada, sino solo las varias puertas de enlace de API.

Sin embargo, cuando el destino es un entorno de "producción" basado en Kubernetes, eShopOnContainers usa una entrada delante de las puertas de enlace de API. De este modo, los clientes siguen llamando a la misma dirección URL base, pero las solicitudes se enrutan a varias puertas de enlace de API o BFF.

Las puertas de enlace de API son front-end o fachadas que solo muestran los servicios, pero no las aplicaciones web que normalmente están fuera de su ámbito. Además, las puertas de enlace de API pueden ocultar determinados microservicios internos.

Sin embargo, la entrada solo redirige las solicitudes HTTP, pero no intenta ocultar ningún microservicio o aplicación web.

Tener un nivel nginx de entrada en Kubernetes delante de las aplicaciones web, además de las varias puertas de enlace de API de Ocelot o BFF es la arquitectura ideal, como se muestra en el diagrama siguiente.

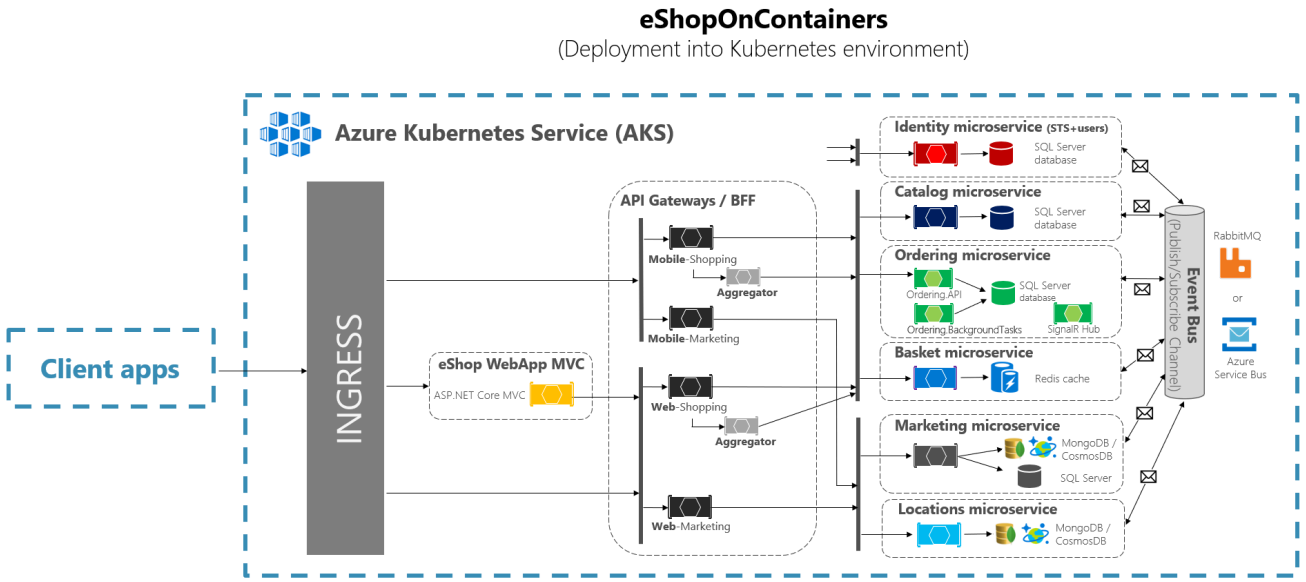


Figura 6-41. Nivel de entrada en eShopOnContainers cuando se implementa en Kubernetes

Una entrada de Kubernetes actúa como proxy inverso para todo el tráfico a la aplicación, incluidas las aplicaciones web, que están fuera del ámbito de la puerta de enlace de API. Al implementar eShopOnContainers en Kubernetes, expone solo algunos servicios o puntos de conexión a través de la *entrada*, básicamente la siguiente lista de postfijos en las direcciones URL:

- `/` para la aplicación web spa cliente
- `/webmvc` para la aplicación web MVC cliente
- `/webstatus` para la aplicación web cliente que muestra los estados o comprobaciones de estado
- `/webshoppingapigw` para el BFF web y los procesos empresariales de compras
- `/webmarketingapigw` para los procesos empresariales de marketing y BFF web
- `/mobileshoppingapigw` para el BFF móvil y los procesos empresariales de compras
- `/mobilemarketingapigw` para los procesos de negocio de marketing y BFF móviles

Al implementar en Kubernetes, cada puerta de enlace de API de Ocelot usa un archivo "configuration.json" diferente para cada *pod* que ejecuta las puertas de enlace de API. Los archivos "configuration.json" se proporcionan montando (originalmente con el script de deploy.ps1) un volumen creado en función de un *mapa de configuración* de Kubernetes denominado "ocelot". Cada contenedor monta su archivo de configuración relacionado en la carpeta del contenedor denominada `/app/configuration`.

En los archivos de código fuente de eShopOnContainers, los archivos originales "configuration.json" se pueden encontrar en la `k8s/ocelot/` carpeta . Hay un archivo para cada BFF/APIGateway.

Características transversales adicionales en una puerta de enlace de API de Ocelot

Hay otras características importantes para investigar y usar, al usar una puerta de enlace de API de Ocelot, que se describe en los vínculos siguientes.

- **Detección de servicios en el lado cliente que integra Ocelot con Consul o Eureka**
  https://ocelot.readthedocs.io/en/latest/features/servicediscovery.html
- **Almacenamiento en caché en el nivel de puerta de enlace de API**
  https://ocelot.readthedocs.io/en/latest/features/caching.html
- **Registro en el nivel de puerta de enlace de API**
  https://ocelot.readthedocs.io/en/latest/features/logging.html
- **Calidad de servicio (reintentos y disyuntores) en el nivel de puerta de enlace de API**
  https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html
- **Limitación de frecuencia**
  https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html
- **Swagger para Ocelot**
  https://github.com/Burgyn/MMLib.SwaggerForOcelot

Siguiente anterior

Abordar la complejidad empresarial en un microservicio con patrones DDD y CQRS

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Diseñe un modelo de dominio para cada microservicio o contexto enlazado que refleje la comprensión del dominio empresarial.

Esta sección se centra en microservicios más avanzados que se implementan cuando es necesario abordar subsistemas complejos o microservicios derivados del conocimiento de expertos de dominio con reglas de negocio que cambian constantemente. Los patrones de arquitectura usados en esta sección se basan en los enfoques de diseño controlado por dominio (DDD) y Segregación de responsabilidad de comandos y consultas (CQRS), como se muestra en la figura 7-1.

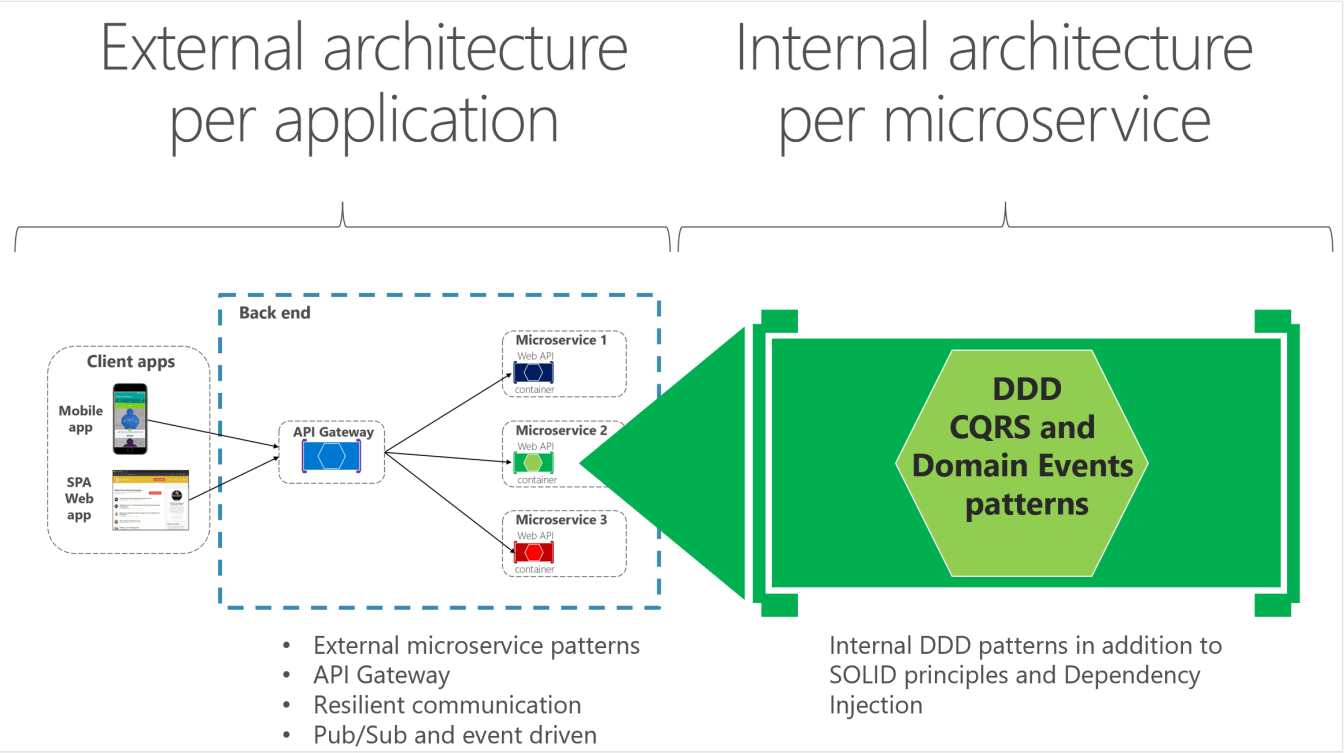


Figura 7-1. Arquitectura de microservicios externos frente a patrones de arquitectura interna para cada microservicio

Sin embargo, la mayoría de las técnicas para los microservicios controlados por datos, como cómo implementar un servicio de API web principal de ASP.NET o cómo exponer metadatos de Swagger con Swashbuckle o NSwag, también se aplican a los microservicios más avanzados implementados internamente con patrones DDD. Esta sección es una extensión de las secciones anteriores, ya que la mayoría de las prácticas que se han explicado anteriormente también se aplican aquí o para cualquier tipo de microservicio.

En esta sección se proporcionan detalles sobre los patrones CQRS simplificados usados en la aplicación de referencia eShopOnContainers. Más adelante, obtendrá información general sobre las técnicas de DDD que le permiten encontrar patrones comunes que puede reutilizar en las aplicaciones.

DDD es un tema grande con un amplio conjunto de recursos para el aprendizaje. Puedes empezar con libros como Domain-Driven Design de Eric Evans y materiales adicionales de Vaughn Vernon, Jimmy Nilsson, Greg Young, Udi Dahan, Jimmy Bogard y muchos otros expertos en DDD/CQRS. Pero, sobre todo, para aprender a aplicar técnicas DDD, debe recurrir a conversaciones, pizarras interactivas y sesiones de modelado de dominio con expertos de su ámbito empresarial específico.

Recursos adicionales

DDD (diseño guiado por el dominio)

- **Eric Evans. Idioma del dominio**
  https://domainlanguage.com/
- **Martin Fowler. Diseño basado en dominios**
  https://martinfowler.com/tags/domain%20driven%20design.html
- **Jimmy Bogard. Fortalecimiento de su dominio: una guía básica**
  https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/
- **Seminario web sobre diseño distribuido Domain-Driven**
  https://particular.net/webinars/ddd-design-no-nonsense-implementation-guide

Libros sobre DDD

- **Eric Evans. Domain-Driven Diseño: Abordar la complejidad en el corazón del software**
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/
- **Eric Evans. Referencia de Diseño de Domain-Driven: Definiciones y Resúmenes de Patrones**
  https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-2014-09-22/dp/B01N8YB4ZO/
- **Vaughn Vernon. Implementing Domain-Driven Design (Implementación del diseño controlado por dominios)**
  https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/
- **Vaughn Vernon. Domain-Driven Design Distilled (Diseño orientado al dominio simplificado)**
  https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420/
- **Jimmy Nilsson. Applying Domain-Driven Design and Patterns (Aplicación de patrones y diseños orientados al dominio)**
  https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202/
- **Cesar de la Torre. N-Layered Domain-Oriented Architecture Guide with .NET (Arquitectura orientada al dominio en N capas con .NET)**
  https://www.amazon.com/N-Layered-Domain-Oriented-Architecture-Guide-NET/dp/8493903612/

- **Abel Avram y Floyd Marinescu. Domain-Driven Design Quickly (Diseño orientado al dominio rápido)**
  https://www.amazon.com/Domain-Driven-Design-Quickly-Abel-Avram/dp/1411609255/
- **Scott Millett, Nick Tune: patrones, principios y prácticas de diseño de Domain-Driven**
  https://www.wiley.com/Patterns%2C+Principles%2C+and+Practices+of+Domain+Driven+Design-p-9781118714706

Aprendizaje de DDD

- **Julie Lerman y Steve Smith. Domain-Driven aspectos básicos del diseño**
  https://www.pluralsight.com/courses/fundamentals-domain-driven-design

Siguiente anterior

Aplicación de patrones CQRS y DDD simplificados en un microservicio

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

CQRS es un patrón arquitectónico que separa los modelos para leer y escribir datos. El término relacionado Separación de comando-consulta (CQS) fue definido originalmente por Bertrand Meyer en su libro *Object-Oriented Construcción de software*. La idea básica es que puede dividir las operaciones de un sistema en dos categorías separadas claramente:

- Consultas. Estas consultas devuelven un resultado y no cambian el estado del sistema y están libres de efectos secundarios.
- Comandos. Estos comandos cambian el estado de un sistema.

CQS es un concepto sencillo: se trata de métodos dentro del mismo objeto, ya sea consultas o comandos. Cada método devuelve el estado o muta, pero no ambos. Incluso un único objeto de patrón de repositorio puede cumplir con CQS. CQS se puede considerar un principio fundamental para CQRS.

La segregación de responsabilidades de comandos y consultas (CQRS) fue introducida por Greg Young y promocionada fuertemente por Udi Dahan y otros. Se basa en el principio CQS, aunque es más detallado. Se puede considerar un patrón basado en comandos y eventos, además de opcionalmente en mensajes asincrónicos. En muchos casos, CQRS está relacionado con escenarios más avanzados, como tener una base de datos física diferente para lecturas

(consultas) que para escrituras (actualizaciones). Además, un sistema CQRS más evolucionado podría implementar Event-Sourcing (ES) para la base de datos de actualizaciones, por lo que solo almacenaría eventos en el modelo de dominio en lugar de almacenar los datos de estado actual. Sin embargo, este enfoque no se usa en esta guía. En esta guía se usa el enfoque CQRS más sencillo, que consiste simplemente en separar las consultas de los comandos.

El aspecto de separación de CQRS se logra mediante la agrupación de operaciones de consulta en una capa y comandos de otra capa. Cada capa tiene su propio modelo de datos (tenga en cuenta que decimos modelo, no necesariamente una base de datos diferente) y se crea con su propia combinación de patrones y tecnologías. Lo que es más importante, las dos capas pueden estar dentro del mismo nivel o microservicio, como en el ejemplo (ordenación de microservicios) que se usa para esta guía. O bien, podrían implementarse en diferentes microservicios o procesos para que se puedan optimizar y escalar horizontalmente por separado sin afectar entre sí.

CQRS significa tener dos objetos para una operación de lectura y escritura donde en otros contextos hay uno. Hay razones para tener una base de datos de lecturas desnormalizada, que puede obtener información sobre en la literatura de CQRS más avanzada. Pero no estamos usando ese enfoque aquí, donde el objetivo es tener más flexibilidad en las consultas en lugar de limitar las consultas con restricciones de patrones DDD, como agregados.

Un ejemplo de este tipo de servicio es el microservicio de ordenación de la aplicación de referencia eShopOnContainers. Este servicio implementa un microservicio basado en un enfoque simplificado de CQRS. Usa un único origen de datos o base de datos, pero dos modelos lógicos más patrones DDD para el dominio transaccional, como se muestra en la figura 7-2.

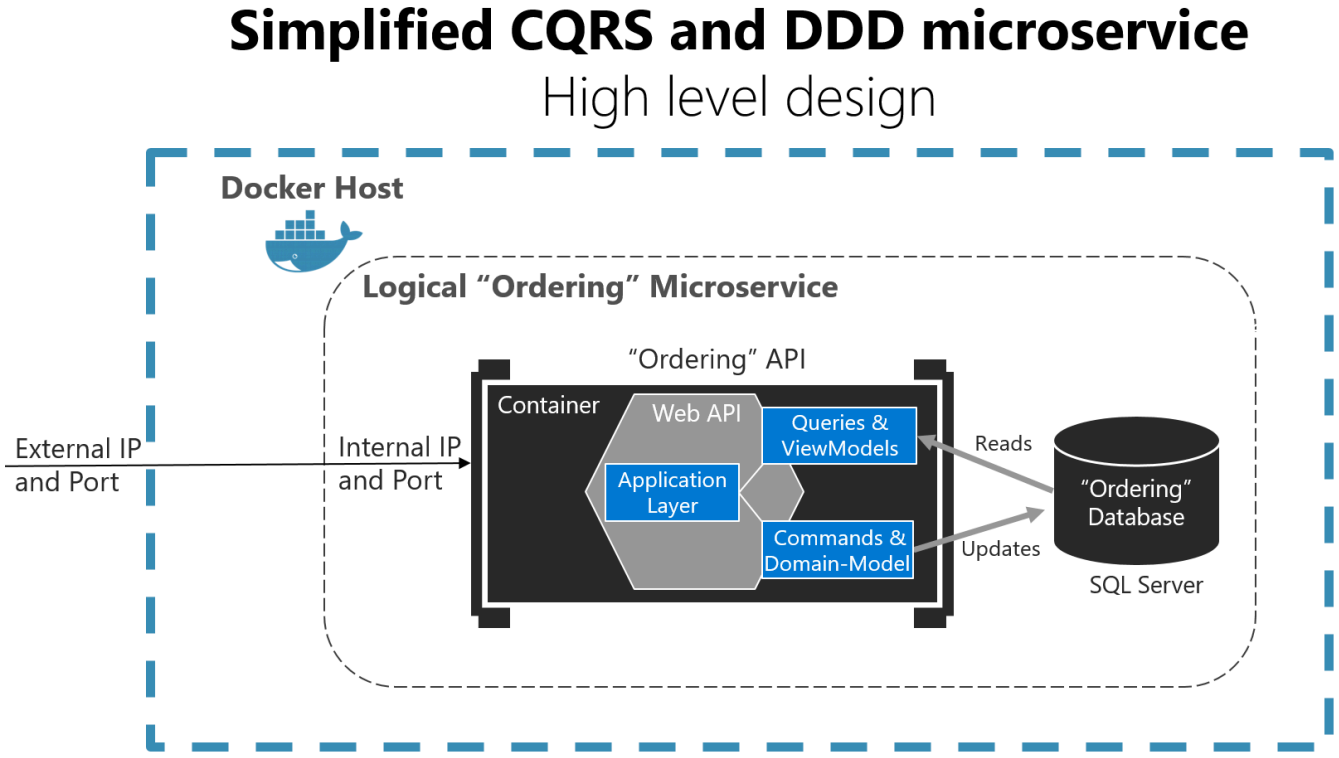


Figura 7-2. Microservicio simplificado basado en CQRS y DDD

El microservicio lógico "Ordering" incluye su base de datos Ordering, que puede ser, pero no necesariamente tiene que ser, en el mismo servidor Docker. Tener la base de datos en el mismo host de Docker es buena para el desarrollo, pero no para producción.

La capa de aplicación puede ser la propia API web. El aspecto de diseño importante aquí es que el microservicio ha dividido las consultas y ViewModels (modelos de datos especialmente creados para las aplicaciones cliente) de los comandos, el modelo de dominio y las transacciones siguiendo el patrón CQRS. Este enfoque mantiene las consultas independientes de las restricciones y restricciones procedentes de patrones DDD que solo tienen sentido para las transacciones y las actualizaciones, como se explica en secciones posteriores.

Recursos adicionales

- **Greg Young. Control de versiones en un sistema de origen de eventos** (Gratis para leer e-book en línea)
  https://leanpub.com/esversioning/read

Siguiente anterior

Aplicar enfoques de CQRS y CQS en un microservicio DDD en eShopOnContainers

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El diseño del microservicio de ordenación en la aplicación de referencia eShopOnContainers se basa en los principios de CQRS. Sin embargo, usa el enfoque más sencillo, que simplemente separa las consultas de los comandos y usa la misma base de datos para ambas acciones.

La esencia de esos patrones y el punto importante aquí es que las consultas son idempotentes: independientemente de cuántas veces consulte un sistema, el estado de ese sistema no cambiará. En otras palabras, las consultas están libres de efectos secundarios.

Por lo tanto, podría usar un modelo de datos de "lectura" distinto al modelo de dominio de "escritura" de lógica transaccional, aunque los microservicios de ordenación usen la misma base de datos. Por lo tanto, se trata de un enfoque simplificado de CQRS.

Por otro lado, los comandos, que desencadenan transacciones y actualizaciones de datos, cambian el estado en el sistema. Debe tener cuidado con los comandos cuando trabaje con la complejidad y las reglas de negocio cambiantes. Aquí es donde desea aplicar técnicas DDD para tener un sistema mejor modelado.

Los patrones DDD presentados en esta guía no se deben aplicar universalmente. Introducen restricciones en el diseño. Estas restricciones proporcionan ventajas como una mayor calidad a lo largo del tiempo, especialmente en comandos y en otro código que modifica el estado del

sistema. Sin embargo, esas restricciones agregan complejidad con menos ventajas para leer y consultar datos.

Uno de estos patrones es el patrón agregado, que examinamos más en secciones posteriores. Brevemente, en el patrón Aggregate, se tratan muchos objetos de dominio como una sola unidad como resultado de su relación en el dominio. Es posible que no siempre obtenga ventajas de este patrón en las consultas; puede aumentar la complejidad de la lógica de consulta. En el caso de las consultas de solo lectura, no se obtienen las ventajas de tratar varios objetos como un único agregado. Solo se obtiene la complejidad.

Como se muestra en la figura 7-2 de la sección anterior, esta guía sugiere el uso de patrones DDD solo en el área transaccional/actualizaciones del microservicio (es decir, como desencadenan los comandos). Las consultas pueden seguir un enfoque más sencillo y deben separarse de los comandos, siguiendo un enfoque de CQRS.

Para implementar el "lado de consultas", puede elegir entre varios enfoques, desde un ORM completo como EF Core, proyecciones de AutoMapper, procedimientos almacenados, vistas, vistas materializadas o un micro ORM.

En esta guía y en eShopOnContainers (específicamente el microservicio de ordenación) hemos elegido implementar consultas rectas mediante un micro ORM como Dapper. Esta guía le permite implementar cualquier consulta basada en instrucciones SQL para obtener el mejor rendimiento, gracias a un marco ligero con poca sobrecarga.

Cuando se usa este enfoque, las actualizaciones del modelo que afectan a cómo se conservan las entidades en una base de datos SQL también necesitan actualizaciones independientes de las consultas SQL usadas por Dapper o cualquier otro enfoque independiente (no EF) para realizar consultas.

Los patrones CQRS y DDD no son arquitecturas de nivel superior

Es importante comprender que los patrones CQRS y la mayoría de los patrones DDD (como las capas DDD o un modelo de dominio con agregados) no son estilos arquitectónicos, sino solo patrones de arquitectura. Los microservicios, SOA y la arquitectura controlada por eventos (EDA) son ejemplos de estilos arquitectónicos. Describen un sistema de muchos componentes, como muchos microservicios. Los patrones CQRS y DDD describen algo dentro de un único sistema o componente; en este caso, algo dentro de un microservicio.

Los diferentes contextos delimitados (CD) emplearán distintos patrones. Tienen diferentes responsabilidades y que conducen a diferentes soluciones. Vale la pena destacar que forzar el mismo patrón en todas partes conduce a un fracaso. No use patrones CQRS y DDD en todas

partes. Muchos subsistemas, contextos delimitados o microservicios son más sencillos y se pueden implementar más fácilmente mediante servicios CRUD simples o mediante otro enfoque.

Solo hay una arquitectura de aplicación: la arquitectura del sistema o la aplicación de un extremo a otro que está diseñando (por ejemplo, la arquitectura de microservicios). Sin embargo, el diseño de cada Contexto Delimitado o microservicio dentro de esa aplicación refleja sus propios compromisos y decisiones de diseño internas en el nivel de patrones arquitectónicos. No intente aplicar los mismos patrones arquitectónicos que CQRS o DDD en todas partes.

Recursos adicionales

- **Martin Fowler. CQRS**
  https://martinfowler.com/bliki/CQRS.html
- **Greg Young. Documentos de CQRS**
  https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
- **Udi Dahan. CQRS clarificado**
  https://udidahan.com/2009/12/09/clarified-cqrs/

Siguiente anterior

Implementación de lecturas/consultas en un microservicio CQRS

Artículo • 09/03/2023

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en **Documentación de .NET** o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Para lecturas/consultas, el microservicio de pedidos (Ordering) de la aplicación de referencia eShopOnContainers implementa las consultas de manera independiente del modelo DDD y el área transaccional. Esta implementación se hacía principalmente porque las demandas de consultas y transacciones son muy diferentes. Las escrituras ejecutan transacciones que deben ser compatibles con la lógica del dominio. Por otro lado, las consultas son idempotentes y se pueden segregar de las reglas de dominio.

El enfoque es sencillo, como se muestra en la figura 7-3. La interfaz API se implementa mediante los controladores de API Web con cualquier infraestructura, como un microasignador objeto-relacional (ORM) como Dapper, y devolviendo ViewModel dinámicos según las necesidades de las aplicaciones de interfaz de usuario.

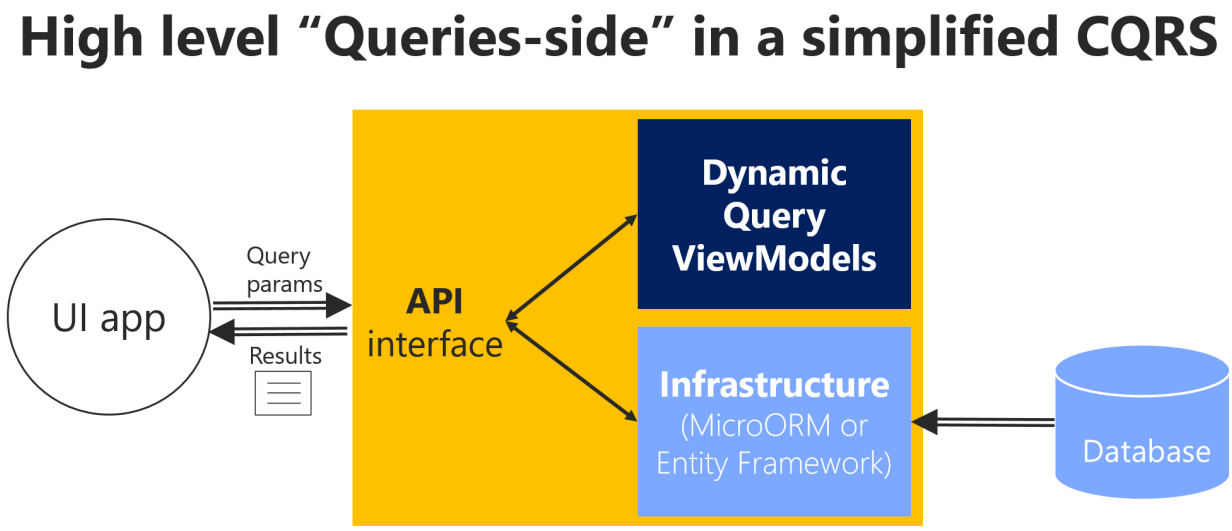


Figura 7-3. El enfoque más sencillo para las consultas en un microservicio CQRS

El enfoque más sencillo para el lado de las consultas en un enfoque CQRS simplificado se puede implementar consultando la base de datos con un Micro-ORM como Dapper, devolviendo ViewModel dinámicos. Las definiciones de consulta realizan una consulta a la base de datos y devuelven un ViewModel dinámico creado sobre la marcha para cada consulta. Puesto que las consultas son idempotentes, no cambian los datos por muchas veces que ejecute una consulta. Por lo tanto, no es necesario estar restringido por un patrón DDD usado en el lado transaccional, como agregados y otros patrones, y por eso las consultas se separan del área transaccional. Consulta la base de datos para obtener los datos que necesita la interfaz de usuario y devolver un ViewModel dinámico que no tiene que estar definido estáticamente en ningún lugar (no hay clases para los ViewModel), excepto en las propias instrucciones SQL.

Puesto que se trata de un método sencillo, el código necesario para el lado de las consultas (como código que usa un micro ORM como Dapper) pueden implementarse dentro del mismo proyecto de API Web. En la figura 7-4 se muestra este enfoque. Las consultas se definen en el proyecto de microservicio **Ordering.API** dentro de la solución eShopOnContainers.

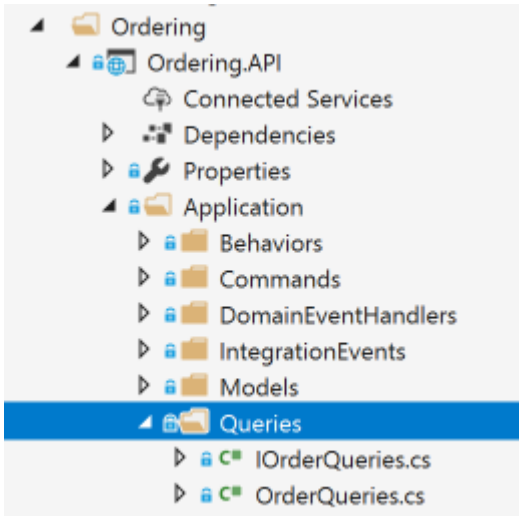


Figura 7-4. Consultas (Queries) en el microservicio de pedidos (Ordering) en eShopOnContainers

Uso de ViewModel específicos para aplicaciones de cliente, sin las restricciones del modelo de dominio

Dado que las consultas se realizan para obtener los datos que necesitan para las aplicaciones cliente, el tipo de valor devuelto puede estar hecho específicamente para los clientes, en función de los datos devueltos por las consultas. Estos modelos, u objetos de transferencia de datos (DTO), se denominan ViewModel.

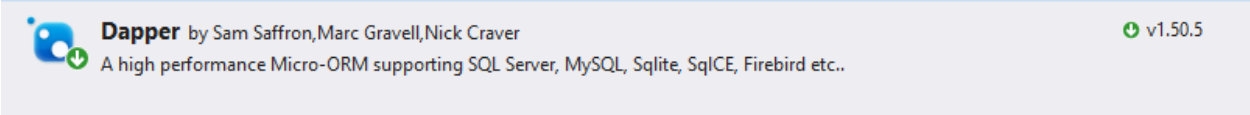
Los datos devueltos (ViewModel) pueden ser el resultado de combinar datos de varias entidades o tablas de la base de datos, o incluso de varios agregados definidos en el modelo de dominio para el área transaccional. En este caso, dado que va a crear consultas independientes del modelo de dominio, se ignoran las restricciones y los límites de agregados, y se pueden consultar cualquier tabla y columna que necesite. Este enfoque proporciona gran flexibilidad y productividad a los desarrolladores que crean o actualizan las consultas.

Los elementos ViewModels pueden ser tipos estáticos definidos en clases (como se implementa en el microservicio de pedidos). O bien, se pueden crear dinámicamente en función de las consultas realizadas, lo que resulta ágil para los desarrolladores.

Uso de Dapper como micro ORM para realizar consultas

Para la consulta puede usar cualquier micro ORM, Entity Framework Core o incluso ADO.NET estándar. En la aplicación de ejemplo, se seleccionó Dapper para el microservicio de pedidos en eShopOnContainers como un buen ejemplo de un micro ORM popular. Dapper puede ejecutar consultas SQL estándar con un gran rendimiento, porque es un marco de trabajo ligero. Con Dapper, se puede escribir una consulta SQL que puede acceder a varias tablas y combinarlas.

Dapper es un proyecto de código abierto (creado originalmente por Sam Saffron) y forma parte de los bloques de creación que se usan en Stack Overflow. Para usar Dapper, solo hay que instalarlo a través del paquete Dapper de NuGet, tal y como se muestra en la ilustración siguiente:



También debe agregar una directiva `using` para que el código tenga acceso a los métodos de extensión de Dapper.

Cuando se utiliza Dapper en el código, se usa directamente la clase SqlConnection disponible en el espacio de nombres Microsoft.Data.SqlClient. Mediante el método QueryAsync y otros métodos de extensión que extienden la clase SqlConnection, se ejecutan las consultas de una manera sencilla y eficaz.

ViewModel dinámicos frente a estáticos

Cuando se devuelven ViewModel desde el servidor a las aplicaciones cliente, se puede pensar en esos ViewModel como DTO (Objetos de transferencia de datos) que pueden ser diferentes a las entidades de dominio interno de su modelo de entidad, ya que los ViewModel contienen los datos de la forma en que la aplicación cliente necesita. Por lo tanto, en muchos casos, se pueden agregar datos procedentes de varias entidades de dominio y crear los ViewModel exactamente según la forma en que la aplicación cliente necesita los datos.

Esos ViewModels o DTO se pueden definir explícitamente (como clases de contenedor de datos), como la clase `OrderSummary` que se muestra en un fragmento de código posterior. O bien, podría devolver simplemente ViewModels dinámicos o DTO dinámicos en función de los atributos devueltos por las consultas como un tipo dinámico.

ViewModel como tipo dinámico

Como se muestra en el siguiente código, las consultas pueden devolver un `ViewModel` directamente al devolver un tipo *dinámico* que internamente se basa en los atributos devueltos por una consulta. Esto significa que el subconjunto de atributos que se devuelve se basa en la propia consulta. Por tanto, si se agrega una nueva columna a la consulta o combinación, esos datos se agregan dinámicamente al `ViewModel` devuelto.

C#

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;
```

```
public class OrderQueries : IOrderQueries
{
    public async Task<IEnumerable<dynamic>> GetOrdersAsync()
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            return await connection.QueryAsync<dynamic>(
                @"SELECT o.[Id] as ordernumber,
                o.[OrderDate] as [date],os.[Name] as [status],
                SUM(oi.units*oi.unitprice) as total
                FROM [ordering].[Orders] o
                LEFT JOIN[ordering].[orderitems] oi ON o.Id = oi.orderid
                LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId =
os.Id
                GROUP BY o.[Id], o.[OrderDate], os.[Name]");
        }
    }
}
```

Lo importante es que, mediante el uso de un tipo dinámico, la colección de datos devuelta dinámicamente se ensambla como un ViewModel.

Ventajas: este enfoque reduce la necesidad de modificar las clases estáticas de ViewModel cada vez que se actualice la frase SQL de una consulta, lo que hace que este enfoque de diseño sea ágil a la hora de codificar, sencillo y rápido de evolucionar con respecto a los cambios en el futuro.

Inconvenientes: a largo plazo, los tipos dinámicos pueden perjudicar a la claridad y afectar a la compatibilidad de un servicio con las aplicaciones cliente. Además, el software middleware como Swashbuckle no puede proporcionar el mismo nivel de documentación en tipos devueltos si se utilizan tipos dinámicos.

ViewModel como clases DTO predefinidas

Ventajas: disponer de clases ViewModel predefinidas estáticas, como "contratos" basados en clases DTO explícitas, es definitivamente mejor para las API públicas, pero también para los microservicios a largo plazo, aunque solo los use la misma aplicación.

Si quiere especificar los tipos de respuesta de Swagger, debe utilizar clases DTO explícitas como tipo de valor devuelto. Por lo tanto, las clases DTO predefinidas permiten ofrecer información más completa de Swagger. Eso mejora la documentación y la compatibilidad de la API al utilizar una API.

Inconvenientes: tal y como se mencionó anteriormente, al actualizar el código se requieren algunos pasos adicionales para actualizar las clases DTO.

Sugerencia basada en nuestra experiencia: en las consultas que se implementan en el microservicio de pedidos en eShopOnContainers, iniciamos el desarrollo con ViewModel dinámicos porque resultaba sencillo y ágil en las primeras fases de desarrollo. Pero, una vez que se estabilizó el desarrollo, optamos por refactorizar las API y usar DTO estático o predefinido para los ViewModel, porque es más fácil para los consumidores del microservicio conocer los tipos DTO explícitos, utilizados como "contratos".

En el ejemplo siguiente, puede ver cómo la consulta devuelve datos mediante una clase ViewModel DTO explícita: la clase OrderSummary.

C#

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;

public class OrderQueries : IOrderQueries
{
  public async Task<IEnumerable<OrderSummary>> GetOrdersAsync()
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            return await connection.QueryAsync<OrderSummary>(
                  @"SELECT o.[Id] as ordernumber,
                  o.[OrderDate] as [date],os.[Name] as [status],
                  SUM(oi.units*oi.unitprice) as total
                  FROM [ordering].[Orders] o
                  LEFT JOIN[ordering].[orderitems] oi ON  o.Id = oi.orderid
                  LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId =
os.Id
                  GROUP BY o.[Id], o.[OrderDate], os.[Name]
                  ORDER BY o.[Id]");
        }
    }
}
```

Descripción de los tipos de respuesta de las API Web

Lo que más preocupa a los desarrolladores que utilizan API Web y microservicios es lo que se devuelve, sobre todo los tipos de respuesta y los códigos de error (si no son los habituales). Los tipos de respuesta se administran en las anotaciones de datos y en los comentarios XML.

Sin una documentación correcta en la interfaz de usuario de Swagger, el consumidor desconoce los tipos que se devuelven o los códigos HTTP que se pueden devolver. Este problema se corrige agregando Microsoft.AspNetCore.Mvc.ProducesResponseTypeAttribute, para que Swashbuckle pueda generar información completa sobre el modelo de devolución y los valores de API, como se muestra en el siguiente código:

C#

```
namespace Microsoft.eShopOnContainers.Services.Ordering.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class OrdersController : Controller
    {
        //Additional code...
        [Route("")]
        [HttpGet]
        [ProducesResponseType(typeof(IEnumerable<OrderSummary>),
            (int)HttpStatusCode.OK)]
        public async Task<IActionResult> GetOrders()
        {
            var userid = _identityService.GetUserIdentity();
            var orders = await _orderQueries
                .GetOrdersFromUserAsync(Guid.Parse(userid));
            return Ok(orders);
        }
    }
}
```

Pero el atributo `ProducesResponseType` no puede utilizar un tipo dinámico, sino que requiere utilizar tipos explícitos, como ViewModel DTO `OrderSummary`, se mostrado en el ejemplo siguiente:

C#

```
public class OrderSummary
{
    public int ordernumber { get; set; }
    public DateTime date { get; set; }
    public string status { get; set; }
    public double total { get; set; }
}
// or using C# 8 record types:
public record OrderSummary(int ordernumber, DateTime date, string status,
double total);
```

Este es otro de los motivos por los que, a largo plazo, los tipos explícitos son mejores que los tipos dinámicos. Cuando se usa el atributo `ProducesResponseType`, también se puede especificar cuál es el resultado esperado en lo que respecta a posibles errores o códigos HTTP, como 200, 400, etc.

En la siguiente imagen, se puede ver cómo la interfaz de usuario de Swagger de interfaz de usuario muestra la información de ResponseType.

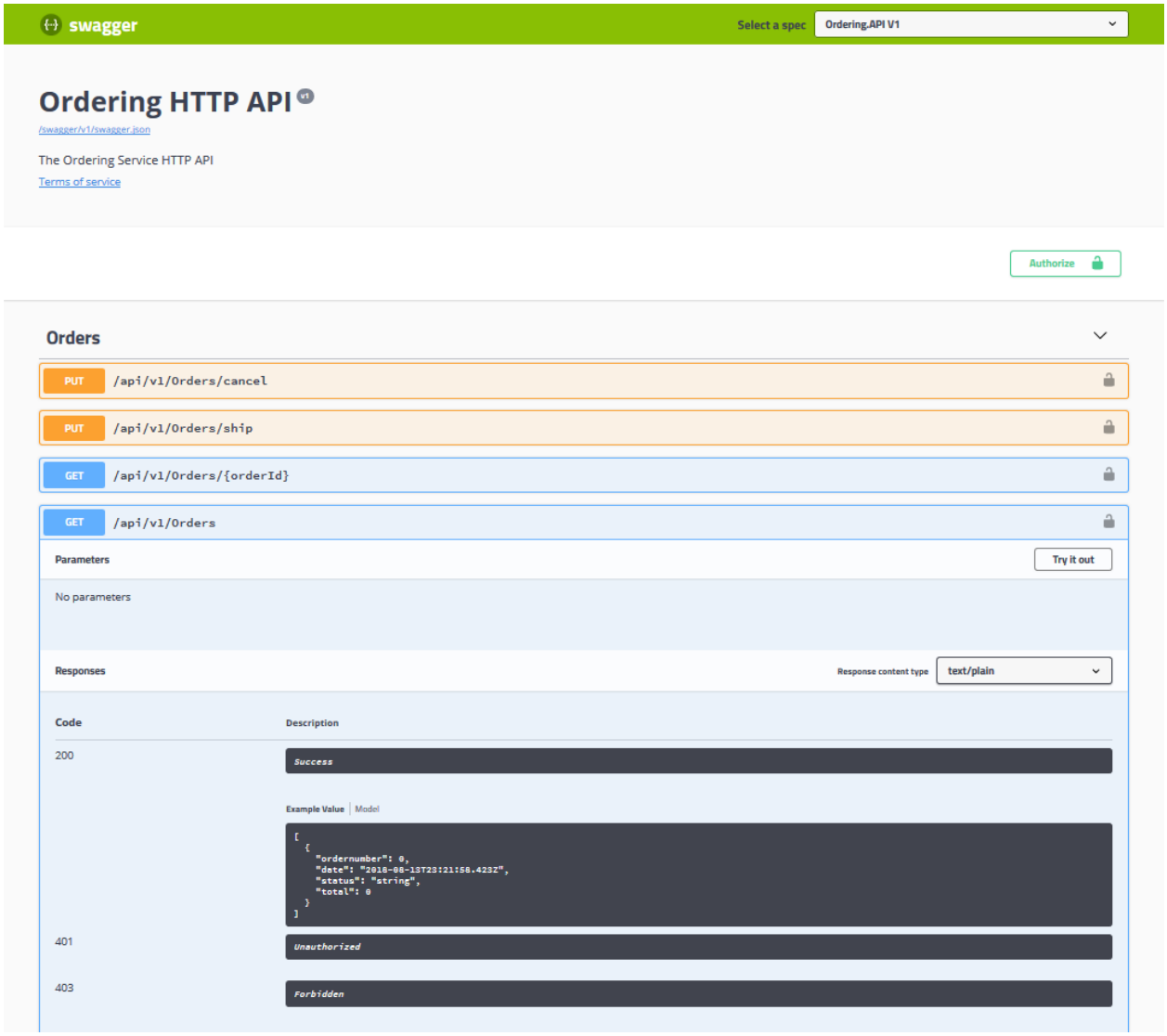


Figura 7-5. Interfaz de usuario de Swagger que muestra los tipos de respuesta y los posibles códigos de estado HTTP de una API Web

En la ilustración anterior se pueden ver algunos valores de ejemplo basados en los tipos ViewModel, además de los posibles códigos de estado HTTP que se pueden devolver.

Recursos adicionales

- **Dapper**
  https://github.com/StackExchange/dapper-dot-net

- **Julie Lerman. Puntos de datos: Dapper, Entity Framework y aplicaciones híbridas (artículo de MSDN magazine)**
  https://learn.microsoft.com/archive/msdn-magazine/2016/may/data-points-dapper-entity-framework-and-hybrid-apps

- **Páginas de ayuda de ASP.NET Core Web API mediante Swagger**
  https://learn.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio

- **Creación de tipos de registros**https://learn.microsoft.com/dotnet/csharp/whats-new/tutorials/records

Anterior Siguiente

Diseño de un microservicio orientado a DDD

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El diseño dirigido por el dominio (DDD) defiende el modelado basado en la realidad empresarial que es relevante para tus casos de uso. En el contexto de la creación de aplicaciones, DDD habla sobre problemas como dominios. Describe áreas de problemas independientes como Contextos delimitados (cada Contexto delimitado se correlaciona con un microservicio) y enfatiza un lenguaje común para hablar sobre estos problemas. También sugiere muchos conceptos y patrones técnicos, como entidades de dominio con modelos enriquecidos (sin modelo de dominio anémico), objetos de valor, agregados y reglas de raíz agregada (o entidad raíz) para admitir la implementación interna. En esta sección se presenta el diseño y la implementación de esos patrones internos.

A veces, estas reglas y patrones técnicos de DDD se perciben como obstáculos que tienen una curva de aprendizaje empinada para implementar enfoques DDD. Pero la parte importante no es los propios patrones, sino la organización del código para que esté alineado con los problemas empresariales y el uso de los mismos términos empresariales (lenguaje omnipresente). Además, los enfoques DDD solo se deben aplicar si va a implementar microservicios complejos con reglas de negocio significativas. Las responsabilidades más sencillas, como un servicio CRUD, se pueden administrar con enfoques más sencillos.

Dónde dibujar los límites es la tarea clave al diseñar y definir un microservicio. Los patrones DDD le ayudan a comprender la complejidad del dominio. Para el modelo de dominio para cada contexto enlazado, se identifican y definen las entidades, los objetos de valor y los agregados que modelan el dominio. Se compila y se refina un modelo de dominio que se encuentra dentro de un límite que define el contexto. Y eso es explícito en forma de microservicio. Los componentes dentro de esos límites terminan siendo sus microservicios, aunque en algunos casos un microservicio bc o empresarial puede estar compuesto de varios servicios físicos. DDD trata sobre límites, al igual que los microservicios.

Mantener los límites de contexto de microservicio relativamente pequeños

Determinar dónde colocar límites entre contextos delimitados busca un equilibrio entre dos objetivos en competencia. En primer lugar, desea crear inicialmente los microservicios más pequeños posibles, aunque eso no debe ser el controlador principal; debe crear un límite en torno a las cosas que necesitan cohesión. En segundo lugar, le interesa evitar comunicaciones locuaces entre microservicios. Estos objetivos pueden contradecirse entre sí. Para equilibrar los microservicios, descompón el sistema en tantos pequeños microservicios como puedas hasta que veas que los límites de comunicación crecen rápidamente con cada intento adicional de separar un nuevo Contexto Delimitado. La cohesión es clave dentro de un único contexto limitado.

Se parece a una inadecuada intuición de código de cercanía al implementar las clases. Si dos microservicios necesitan colaborar mucho entre sí, probablemente deberían ser el mismo microservicio.

Otra manera de mirar este aspecto es la autonomía. Si un microservicio debe confiar en otro servicio para atender directamente una solicitud, no es realmente autónomo.

Niveles en microservicios de DDD

La mayoría de las aplicaciones empresariales con una complejidad técnica y empresarial significativa se definen mediante varias capas. Las capas son un artefacto lógico y no están relacionados con la implementación del servicio. Existen para ayudar a los desarrolladores a administrar la complejidad en el código. Es posible que diferentes capas (como la capa de modelo de dominio frente a la capa de presentación, etc.) tengan tipos diferentes, que exigen traducciones entre esos tipos.

Por ejemplo, una entidad podría cargarse desde la base de datos. A continuación, parte de esa información o una agregación de información, incluidos datos adicionales de otras entidades, se puede enviar a la interfaz de usuario del cliente a través de una API web REST. El punto aquí

es que la entidad de dominio está contenida en la capa de modelo de dominio y no debe propagarse a otras áreas a las que no pertenece, como a la capa de presentación.

Además, debe tener entidades siempre válidas (consulte la sección Diseño de validaciones en la capa de modelo de dominio) controlada por raíces agregadas (entidades raíz). Por lo tanto, las entidades no deben estar ligadas a las vistas del cliente, ya que, en el nivel de la interfaz de usuario, puede que algunos datos aún no hayan sido validados. Este es precisamente el propósito del ViewModel. ViewModel es un modelo de datos exclusivamente para las necesidades de la capa de presentación. Las entidades de dominio no pertenecen directamente a ViewModel. En cambio, debe traducir entre ViewModels y entidades de dominio, y viceversa.

Al abordar la complejidad, es importante tener un modelo de dominio controlado por raíces agregadas que asegúrese de que todas las invariables y reglas relacionadas con ese grupo de entidades (agregado) se realizan a través de un único punto de entrada o puerta, la raíz de agregado.

En la figura 7-5 se muestra cómo se implementa un diseño en capas en la aplicación eShopOnContainers.

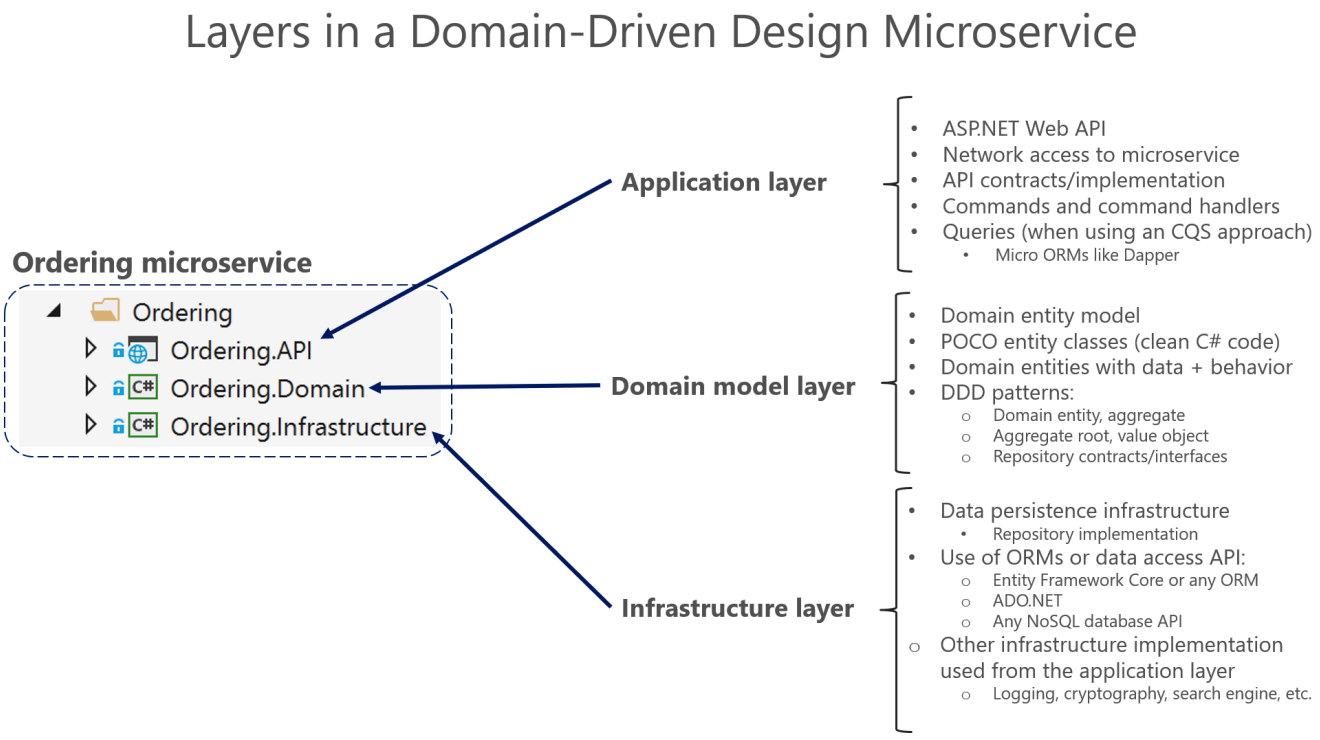


Figura 7-5. Niveles de DDD en el microservicio de ordenación en eShopOnContainers

Las tres capas de un microservicio DDD, como Ordering. Cada capa es un proyecto de VS: la capa de aplicación es Ordering.API, la capa de dominio es Ordering.Domain y la capa de infraestructura es Ordering.Infrastructure. Quiere diseñar el sistema para que cada capa se comunique solo con ciertas otras capas. Ese enfoque puede ser más fácil de aplicar si las capas se implementan como bibliotecas de clases diferentes, ya que puede identificar claramente qué

dependencias se establecen entre bibliotecas. Por ejemplo, la capa del modelo de dominio no debe depender de ninguna otra capa (las clases del modelo de dominio deberían ser objetos de clases comunes, o POCO). Como se muestra en la figura 7-6, la biblioteca de capas **Ordering.Domain** solo tiene dependencias en las bibliotecas de .NET o paquetes NuGet, pero no en ninguna otra biblioteca personalizada, como la biblioteca de datos o la biblioteca de persistencia.

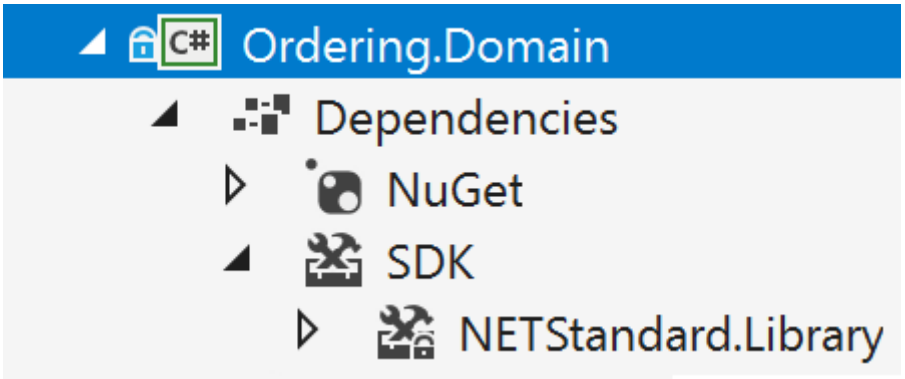


Figura 7-6. Las capas implementadas como bibliotecas permiten un mejor control de las dependencias entre capas

La capa del modelo de dominio

El excelente libro de Eric Evans Domain Driven Design dice lo siguiente sobre la capa de modelo de dominio y la capa de aplicación.

Nivel de modelo de dominio: responsable de representar conceptos del negocio, información sobre la situación del negocio y reglas de negocios. El estado que refleja la situación del negocio se controla y se usa aquí, aunque los detalles técnicos del almacenamiento se delegan a la infraestructura. Esta capa es el núcleo del software empresarial.

La capa del modelo de dominio es donde se expresa la empresa. Al implementar una capa de modelo de dominio de microservicio en .NET, esa capa se codifica como una biblioteca de clases con las entidades de dominio que capturan datos más comportamiento (métodos con lógica).

Siguiendo los principios de Ignorancia de la Persistencia y Ignorancia de la Infraestructura, esta capa debe ignorar por completo los detalles sobre la persistencia de datos. Estas tareas de persistencia deben realizarse en el nivel de infraestructura. Por lo tanto, esta capa no debe tomar dependencias directas en la infraestructura, lo que significa que una regla importante es que las clases de entidad del modelo de dominio deben ser POCOs.

Las entidades de dominio no deben tener ninguna dependencia directa (como derivar de una clase base) en ningún marco de infraestructura de acceso a datos, como Entity Framework o

NHibernate. Idealmente, las entidades de dominio no deberían derivar ni implementar cualquier tipo definido en ningún framework de infraestructura.

La mayoría de los marcos ORM modernos, como Entity Framework Core, permiten este enfoque, de modo que las clases de modelo de dominio no estén acopladas a la infraestructura. Sin embargo, no siempre es posible tener entidades POCO al usar determinadas bases de datos y marcos NoSQL, como Actors y Reliable Collections en Azure Service Fabric.

Incluso cuando es importante seguir el principio de "Ignorancia de la Persistencia" para su modelo de dominio, no debe pasar por alto las cuestiones relacionadas con la persistencia. Sigue siendo importante comprender el modelo de datos físicos y cómo se asigna a un modelo de objetos entidad. De lo contrario, puede crear diseños imposibles.

Además, este aspecto no significa que pueda tomar un modelo diseñado para una base de datos relacional y moverlo directamente a una base de datos NoSQL o orientada a documentos. En algunos modelos de entidad, el modelo podría ajustarse, pero normalmente no lo hace. Todavía hay restricciones que el modelo de entidad debe cumplir, en función de la tecnología de almacenamiento y la tecnología ORM.

La capa de aplicación

Al pasar a la capa de aplicación, podemos citar de nuevo el libro de Eric Evans Domain Driven Design:

Nivel de aplicación: define los trabajos que se supone que el software debe hacer y dirige los objetos de dominio expresivo para que resuelvan problemas. Las tareas de esta capa son significativas para la empresa o necesarias para la interacción con las capas de aplicación de otros sistemas. Esta capa se mantiene delgada. No contiene reglas de negocio ni conocimientos, sino que solo coordina tareas y delega trabajo a colaboraciones de objetos de dominio en la capa siguiente. No tiene estado que refleje la situación empresarial, pero puede tener un estado que refleje el progreso de una tarea para el usuario o el programa.

La capa de aplicación de un microservicio en .NET se codifica normalmente como un proyecto de API web de ASP.NET Core. El proyecto implementa la interacción del microservicio, el acceso a la red remota y las API web externas que se usan desde la interfaz de usuario o las aplicaciones cliente. Incluye consultas si se usa un enfoque de CQRS, comandos aceptados por el microservicio e incluso la comunicación controlada por eventos entre microservicios (eventos de integración). La API web de ASP.NET Core que representa el nivel de aplicación no debe contener reglas de negocio ni conocimientos de dominio (especialmente reglas de dominio para transacciones o actualizaciones); estos deben ser propiedad de la biblioteca de clases de modelo de dominio. La capa de aplicación solo debe coordinar las tareas y no debe

contener ni definir ningún estado de dominio (modelo de dominio). Delega la ejecución de reglas de negocio a las clases del modelo de dominio (raíces agregadas y entidades de dominio), que en última instancia actualizarán los datos dentro de esas entidades de dominio.

Básicamente, la lógica de la aplicación es donde se implementan todos los casos de uso que dependen de un front-end determinado. Por ejemplo, la implementación relacionada con un servicio de API web.

El objetivo es que la lógica de dominio en el nivel de modelo de dominio, sus invariables, el modelo de datos y las reglas de negocio relacionadas deben ser completamente independientes de las capas de presentación y aplicación. La mayor parte de todo, la capa de modelo de dominio no debe depender directamente de ningún marco de infraestructura.

El nivel de infraestructura

La capa de infraestructura es cómo los datos que se mantienen inicialmente en entidades de dominio (en memoria) se conservan en bases de datos u otro almacén persistente. Un ejemplo consiste en usar código de Entity Framework Core para implementar las clases de patrón Repository que usan dbContext para conservar los datos en una base de datos relacional.

De acuerdo con los principios de ignorancia de persistencia e ignorancia de infraestructura mencionados anteriormente, la capa de infraestructura no debe "contaminar" la capa del modelo de dominio. Debe mantener las clases de entidad del modelo de dominio independientes de la infraestructura que se usa para conservar los datos (EF o cualquier otro framework) al no adquirir dependencias fuertes con los frameworks. La biblioteca de clases de la capa del modelo de dominio debe contener únicamente código de dominio, únicamente clases de entidad POCO que constituyen el núcleo del software y totalmente desacopladas de las tecnologías de infraestructura.

Por lo tanto, las capas o bibliotecas de clases y los proyectos deben depender en última instancia de la capa del modelo de dominio (biblioteca), no viceversa, como se muestra en la figura 7-7.

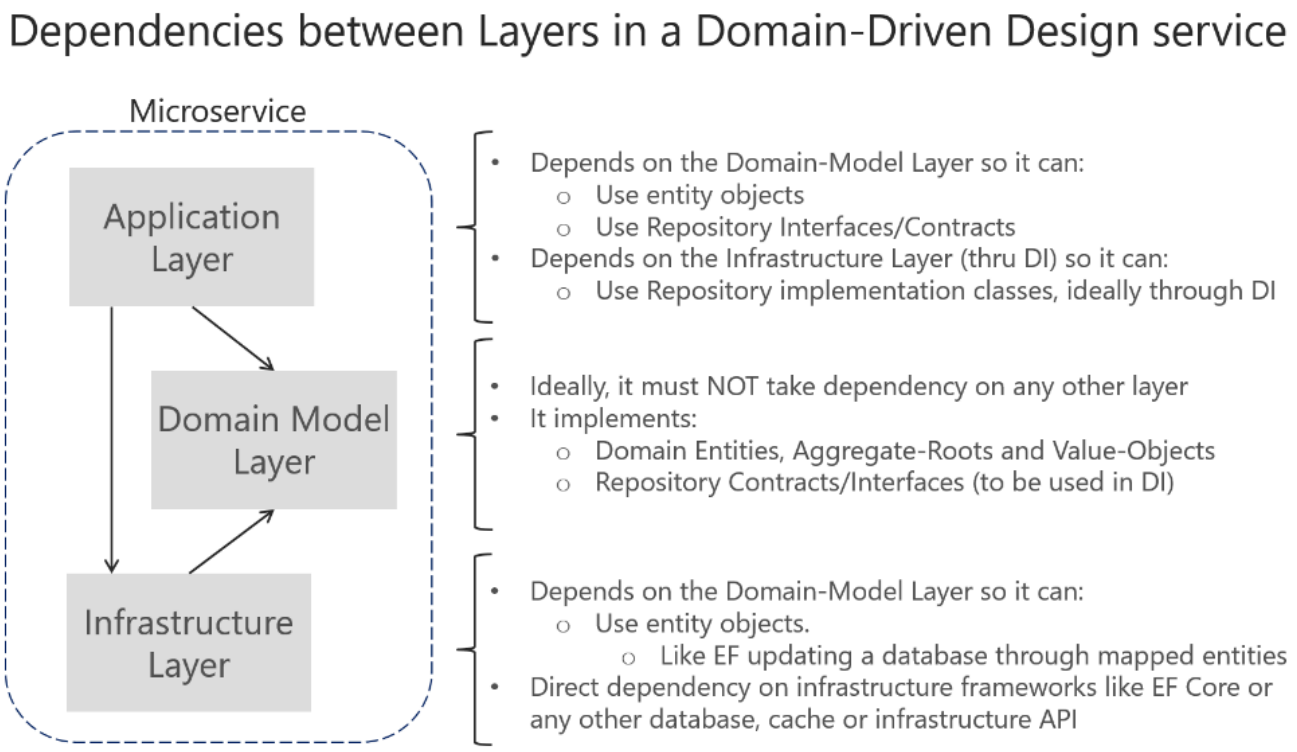


Figura 7-7. Dependencias entre capas en DDD

Las dependencias de un servicio DDD, la capa de aplicación depende del dominio y la infraestructura, y la infraestructura depende del dominio, pero el dominio no depende de ninguna capa. Este diseño de capa debe ser independiente para cada microservicio. Como se indicó anteriormente, puede implementar los microservicios más complejos siguiendo patrones DDD, al tiempo que implementa microservicios controlados por datos más sencillos (CRUD simple en una sola capa) de una manera más sencilla.

Recursos adicionales

- **DevIQ. Persistence Ignorance principle (Principio de omisión de persistencia)** https://deviq.com/persistence-ignorance/
- **Oren Eini. Ignorancia de la infraestructura** https://ayende.com/blog/3137/infrastructure-ignorance
- **Ángel López. Layered Architecture In Domain-Driven Design (Arquitectura por capas en un diseño guiado por el dominio)** https://ajlopez.wordpress.com/2008/09/12/layered-architecture-in-domain-driven-design/

Siguiente anterior

Diseño de un modelo de dominio de microservicio

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Defina un modelo de dominio enriquecido para cada microservicio empresarial o Contexto enlazado.

El objetivo es crear un único modelo de dominio cohesivo para cada microservicio empresarial o contexto enlazado (BC). Tenga en cuenta, sin embargo, que un microservicio BC o empresarial a veces podría estar compuesto de varios servicios físicos que comparten un modelo de dominio único. El modelo de dominio debe capturar las reglas, el comportamiento, el lenguaje empresarial y las restricciones del único microservicio de negocio o contexto enlazado que representa.

Patrón de entidad de dominio

Las entidades representan objetos de dominio y se definen principalmente por su identidad, continuidad y persistencia a lo largo del tiempo, y no solo por los atributos que los componen. Como dice Eric Evans, "un objeto definido principalmente por su identidad se denomina entidad". Las entidades son muy importantes en el modelo de dominio, ya que son la base de un modelo. Por lo tanto, debe identificarlos y planificarlos con cuidado.

La identidad de una entidad puede cruzar varios microservicios o contextos enlazados.

La misma identidad (es decir, el mismo `Id` valor, aunque quizás no sea la misma entidad de dominio) se puede modelar en varios contextos enlazados o microservicios. Sin embargo, esto no implica que la misma entidad, con los mismos atributos y lógica se implementaría en varios contextos enlazados. En su lugar, las entidades de cada contexto enlazado limitan sus atributos y comportamientos a los necesarios en el dominio del contexto enlazado.

Por ejemplo, la entidad compradora puede tener la mayoría de los atributos de una persona que están definidos en la entidad de usuario del microservicio de identidad o perfil, incluida la identidad. Pero la entidad compradora en el microservicio de ordenación podría tener menos atributos, ya que solo ciertos datos del comprador están relacionados con el proceso de pedido. El contexto de cada microservicio o contexto enlazado afecta a su modelo de dominio.

Las entidades de dominio deben implementar el comportamiento además de implementar atributos de datos.

Una entidad de dominio en DDD debe implementar la lógica de dominio o el comportamiento relacionados con los datos de entidad (el objeto al que se accede en memoria). Por ejemplo, como parte de una clase de entidad de pedido, debe tener lógica de negocios y operaciones implementadas mediante métodos para tareas como añadir un artículo al pedido, validación de datos y cálculo total. Los métodos de la entidad se encargan de las invariables y reglas de la entidad en lugar de tener esas reglas distribuidas en la capa de aplicación.

En la figura 7-8 se muestra una entidad de dominio que implementa no solo atributos de datos, sino operaciones o métodos con lógica de dominio relacionada.

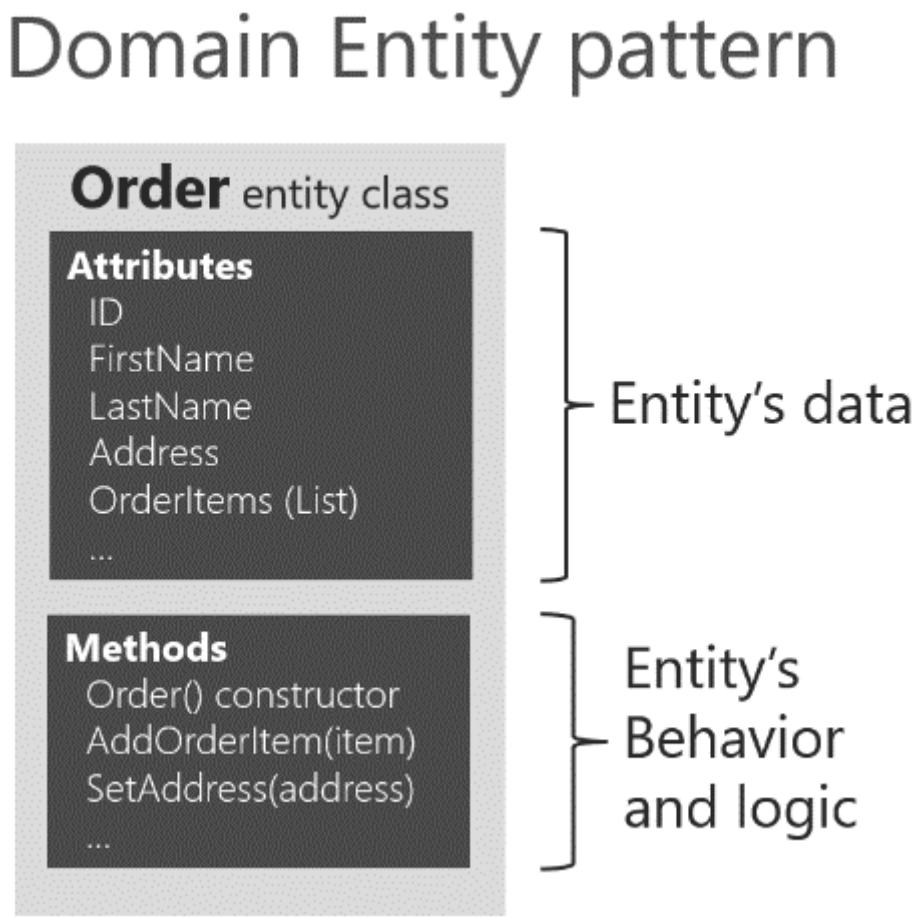


Figura 7-8. Ejemplo de diseño de entidad de dominio que implementa datos más comportamiento

Una entidad de modelo de dominio implementa comportamientos a través de métodos, es decir, no es un modelo "anémico". Por supuesto, a veces puede haber entidades que no implementen ninguna lógica dentro de la clase de entidad. Esto puede ocurrir en entidades secundarias dentro de un agregado si la entidad secundaria no tiene ninguna lógica especial porque la mayoría de la lógica se define en la raíz de agregado. Si tiene un microservicio complejo que tiene lógica implementada en las clases de servicio en lugar de en las entidades de dominio, podría caer en el modelo de dominio anémico, que se explica en la sección siguiente.

Modelo de dominio enriquecido frente al modelo de dominio anémico

En su publicación AnemicDomainModel, Martin Fowler describe un modelo de dominio anémico de esta manera:

El síntoma básico de un modelo de dominio anémico es que a primera vista parece real. Hay objetos, muchos denominados después de los nombres del espacio de dominio, y estos objetos están conectados con las relaciones enriquecidas y la estructura que tienen los modelos de dominio verdaderos. Lo interesante aparece cuando se examina el

comportamiento y se descubre que apenas hay comportamiento en estos objetos, lo que los convierte en poco más que conjuntos de captadores y establecedores.

Por supuesto, cuando se usa un modelo de dominio anémico, esos modelos de datos se usarán desde un conjunto de objetos de servicio (denominados tradicionalmente la *capa de negocio*) que capturan todo el dominio o la lógica de negocios. La capa empresarial se sitúa encima del modelo de datos y utiliza el modelo de datos como si fueran datos.

El modelo de dominio anémico es simplemente un diseño de estilo de procedimiento. Los objetos de entidad anémicos no son objetos reales porque carecen de comportamiento (métodos). Solo contienen propiedades de datos y, por tanto, no es un diseño orientado a objetos. Al colocar todo el comportamiento en objetos de servicio (la capa de negocio), básicamente termina con scripts de transacción o código de espaguetis y, por lo tanto, pierde las ventajas que proporciona un modelo de dominio.

Independientemente de si el microservicio o el contexto enlazado es muy sencillo (un servicio CRUD), el modelo de dominio anémico en forma de objetos de entidad con solo propiedades de datos podría ser lo suficientemente bueno, y es posible que no valga la pena implementar patrones DDD más complejos. En ese caso, será simplemente un modelo de persistencia, porque se ha creado deliberadamente una entidad solo con datos para fines CRUD.

Por ese motivo las arquitecturas de microservicios son perfectas para un enfoque de múltiples arquitecturas según cada contexto delimitado. Por ejemplo, en eShopOnContainers, el microservicio de ordenación implementa patrones DDD, pero el microservicio de catálogo, que es un servicio CRUD simple, no.

Algunas personas dicen que el modelo de dominio anémico es un antipatrón. Realmente depende de lo que esté implementando. Si el microservicio que está creando es lo suficientemente sencillo (por ejemplo, un servicio CRUD), siguiendo el modelo de dominio anémico, no es un antipatrón. Sin embargo, si necesitas abordar la complejidad del dominio de un microservicio que tiene muchas reglas de negocio cambiantes, el modelo de dominio anémico podría ser un antipatrón para ese microservicio o Contexto Delimitado. En ese caso, diseñarlo como un modelo enriquecido con entidades que contienen datos más comportamiento, así como implementar patrones DDD adicionales (agregados, objetos de valor, etc.) puede tener grandes ventajas para el éxito a largo plazo de dicho microservicio.

Recursos adicionales

- **DevIQ. Entidad de dominio**
  https://deviq.com/entity/
- **Martin Fowler. El modelo de dominio**
  https://martinfowler.com/eaaCatalog/domainModel.html

- **Martin Fowler. El modelo de dominio anémico**
  https://martinfowler.com/bliki/AnemicDomainModel.html

El patrón de objeto de valor

Como ha señalado Eric Evans, "Muchos objetos no tienen identidad conceptual. Estos objetos describen ciertas características de una cosa".

Una entidad requiere una identidad, pero hay muchos objetos en un sistema que no, como el patrón Value Object. Un objeto value es un objeto sin identidad conceptual que describe un aspecto de dominio. Se trata de objetos que se instancian para representar elementos de diseño que solo te preocupan temporalmente. Te *preocupas por lo que* son, no *por quiénes* son. Algunos ejemplos incluyen números y cadenas, pero también pueden ser conceptos de nivel superior, como grupos de atributos.

Algo que es una entidad de un microservicio podría no ser una entidad en otro microservicio, porque en el segundo caso, el contexto enlazado podría tener un significado diferente. Por ejemplo, una dirección de una aplicación de comercio electrónico podría no tener una identidad en absoluto, ya que podría representar solo un grupo de atributos del perfil del cliente para una persona o empresa. En este caso, la dirección debe clasificarse como un objeto de valor. Sin embargo, en una aplicación para una empresa de servicios de energía eléctrica, la dirección del cliente podría ser importante para el dominio empresarial. Por lo tanto, la dirección debe tener una identidad para que el sistema de facturación se pueda vincular directamente a la dirección. En ese caso, una dirección debe clasificarse como una entidad de dominio.

Una persona con un nombre y apellido suele ser una entidad porque una persona tiene identidad, incluso si el nombre y el apellido coinciden con otro conjunto de valores, como si esos nombres también hacen referencia a una persona diferente.

Los objetos value son difíciles de administrar en bases de datos relacionales y ORM como Entity Framework (EF), mientras que en las bases de datos orientadas a documentos son más fáciles de implementar y usar.

EF Core 2.0 y versiones posteriores incluyen la característica Entidades de propiedad que facilita el control de objetos de valor, como veremos en detalle más adelante.

Recursos adicionales

- **Martin Fowler. Patrón de objeto de valor**
  https://martinfowler.com/bliki/ValueObject.html

- **Objeto de Valor**
  https://deviq.com/value-object/
- **Objetos de valor en desarrollo de Test-Driven**
  https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects
- **Eric Evans. "Domain-Driven Design: Tackling Complexity in the Heart of Software" (Diseño orientado al dominio: abordar la complejidad en el corazón del software).** (Libro; incluye una explicación de los objetos de valor)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/

El patrón de agregado

Un modelo de dominio contiene clústeres de diferentes entidades de datos y procesos que pueden controlar un área significativa de funcionalidad, como el suministro de pedidos o el inventario. Una unidad DDD más específica es el agregado, que describe un clúster o grupo de entidades y comportamientos que se pueden tratar como una unidad cohesiva.

Normalmente, defines un agregado basado en las transacciones que necesitas. Un ejemplo clásico es un pedido que también contiene una lista de elementos de pedido. Normalmente, un elemento de pedido será una entidad. Pero será una entidad secundaria dentro del agregado de pedido, que también contendrá la entidad de pedido como su entidad raíz, denominada normalmente raíz agregada.

La identificación de agregados puede ser difícil. Un agregado es un grupo de objetos que deben ser coherentes juntos, pero no puede elegir un grupo de objetos y etiquetarlos como agregados. Debe empezar con un concepto de dominio y pensar en las entidades que se usan en las transacciones más comunes relacionadas con ese concepto. Esas entidades que deben ser coherentes de forma transaccional son las que forman un agregado. Pensar en las operaciones de transacción probablemente es la mejor manera de identificar agregados.

El modelo de raíz agregada o entidad raíz

Un agregado se compone de al menos una entidad: la raíz agregada, también denominada entidad raíz o entidad principal. Además, puede tener varias entidades secundarias y objetos de valor, con todas las entidades y objetos que funcionan conjuntamente para implementar el comportamiento y las transacciones necesarios.

El propósito de una raíz de agregado es garantizar la coherencia del agregado; debe ser el único punto de entrada para las actualizaciones del agregado a través de métodos o operaciones en la clase raíz agregada. Debe realizar cambios en las entidades dentro del

agregado solo a través de la raíz del agregado. Es el guardián de la coherencia del conjunto, teniendo en cuenta todas las invariantes y reglas de coherencia que podrían ser necesarias cumplir en el conjunto. Si cambia de forma independiente una entidad secundaria o un objeto de valor, la raíz de agregado no puede asegurarse de que el agregado está en un estado válido. Sería como una mesa con una pata coja. Mantener la coherencia es el propósito principal de la raíz agregada.

En la figura 7-9, puede ver agregados de ejemplo como el agregado de comprador, que contiene una sola entidad (el agregado raíz Comprador). El agregado de pedido contiene varias entidades y un objeto de valor.

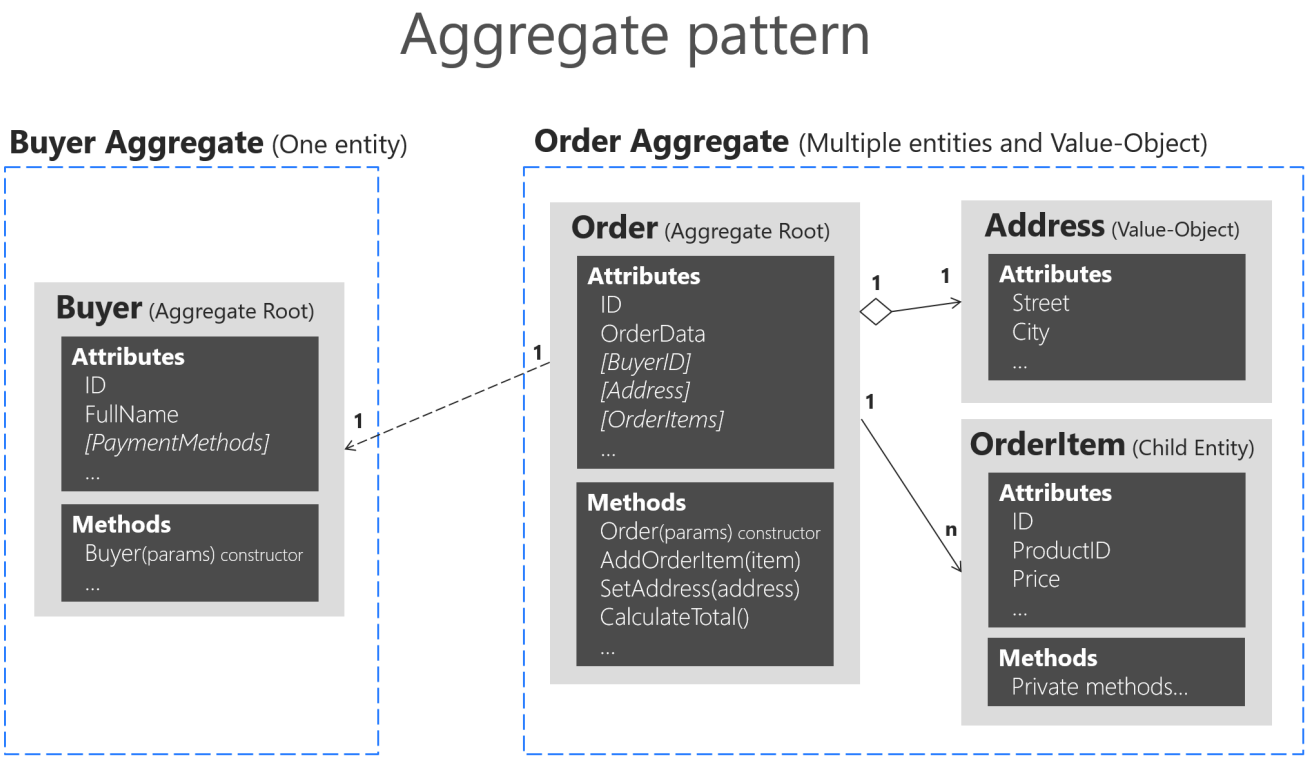


Figura 7-9. Ejemplo de agregados con varias entidades o únicas

Un modelo de dominio DDD se compone de agregados, un agregado puede tener solo una entidad o más, y también puede incluir objetos de valor. Observe que el agregado Comprador podría tener entidades secundarias adicionales, según su dominio, como ocurre en el microservicio de pedidos de la aplicación de referencia eShopOnContainers. En la figura 7-9 solo se muestra un caso en el que el comprador tiene una sola entidad, como ejemplo de un agregado que contiene solo una raíz de agregado.

Con el fin de mantener la separación de agregados y mantener claros los límites entre ellos, es un procedimiento recomendado en un modelo de dominio DDD para no permitir la navegación directa entre agregados y tener solo el campo de clave externa (FK), tal como se implementa en el modelo de dominio de microservicios de pedidos en eShopOnContainers. La entidad Order solo tiene un campo de clave externa para el comprador, pero no una propiedad de navegación de EF Core, como se muestra en el código siguiente:

```C#
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId; // FK pointing to a different aggregate root
    public OrderStatus OrderStatus { get; private set; }
    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;
    // ... Additional code
}
```

La identificación y el trabajo con agregados requieren investigación y experiencia. Para obtener más información, consulte la siguiente lista de recursos adicionales.

Recursos adicionales

- **Vaughn Vernon. Diseño de agregado efectivo: parte I: modelado de un único agregado** (de https://dddcommunity.org/) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf
- **Vaughn Vernon. Diseño de agregado efectivo: parte II: hacer que los agregados funcionen juntos** (de https://dddcommunity.org/) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Vaughn Vernon. Diseño de agregado efectivo: parte III: obtención de información a través de la detección** (de https://dddcommunity.org/) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_3.pdf
- **Sergio Grybniak. Patrones de diseño táctico de DDD** https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part
- **Chris Richardson. Desarrollo de microservicios transaccionales mediante agregados** https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson
- **DevIQ. El patrón de agregado** https://deviq.com/aggregate-pattern/

Siguiente anterior

Implementación de un modelo de dominio de microservicio con .NET

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En la sección anterior, se explicaron los principios y patrones de diseño fundamentales para diseñar un modelo de dominio. Ahora es el momento de explorar posibles formas de implementar el modelo de dominio mediante .NET (código de C# sin formato) y EF Core. El modelo de dominio se compondrá simplemente de tu código. Tendrá solo los requisitos del modelo de EF Core, pero no las dependencias reales de EF. No debe tener dependencias estrictas ni referencias a EF Core ni a ningún otro ORM en su modelo de dominio.

Estructura del modelo de dominio en una biblioteca personalizada de .NET Standard

La organización de carpetas usada para la aplicación de referencia eShopOnContainers muestra el modelo DDD para la aplicación. Es posible que encuentre que una organización de carpetas diferente comunica con más claridad las opciones de diseño realizadas para la aplicación. Como puede ver en la figura 7-10, en el modelo de dominio de ordenación hay dos agregados, el agregado de pedido y el agregado del comprador. Cada agregado es un grupo de entidades de dominio y objetos de valor, aunque podría tener un agregado compuesto por una sola entidad de dominio (la raíz agregada o la entidad raíz).

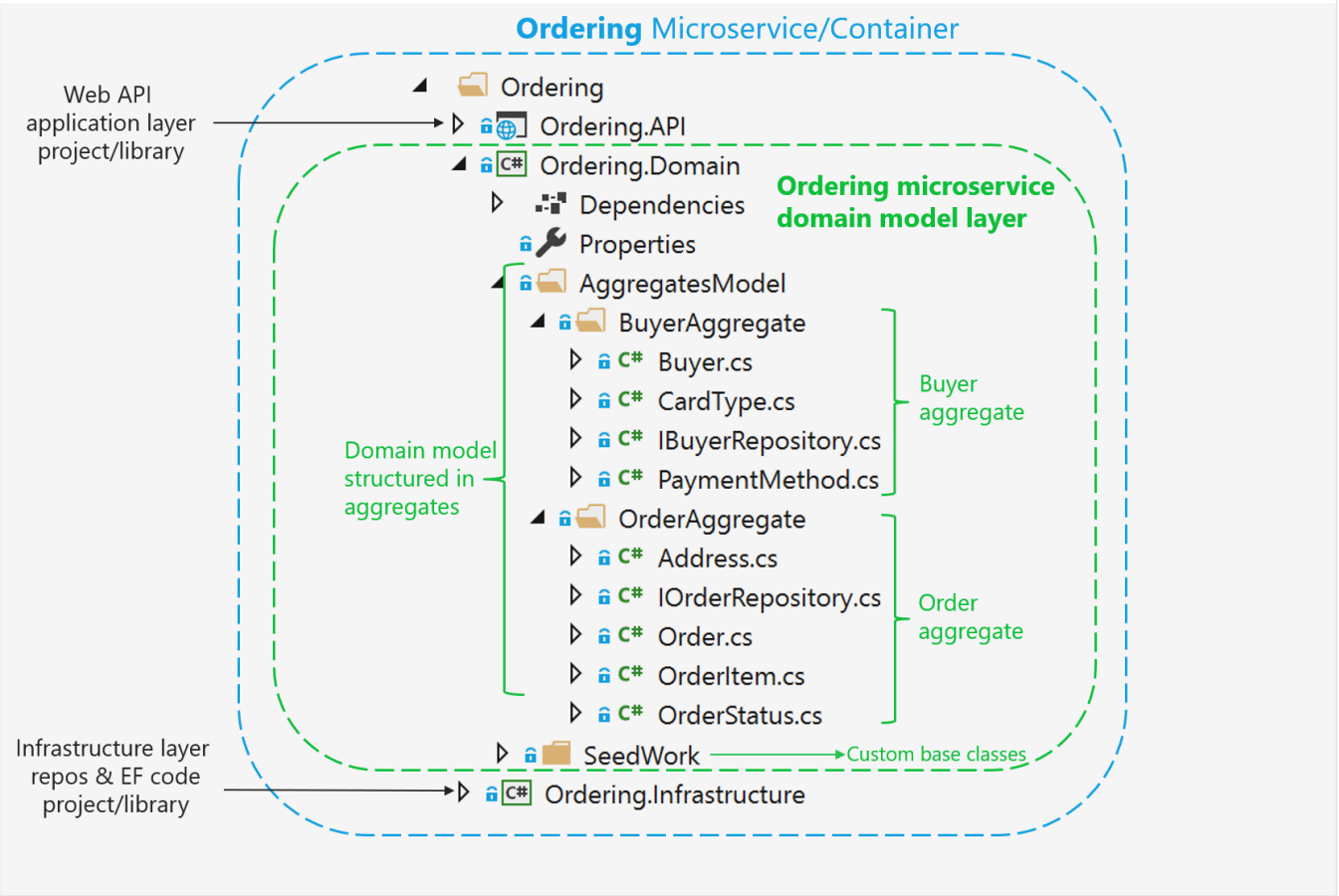


Figura 7-10. Estructura del modelo de dominio para el microservicio de ordenación en eShopOnContainers

Además, la capa de modelo de dominio incluye los contratos de repositorio (interfaces) que son los requisitos de infraestructura del modelo de dominio. En otras palabras, estas interfaces expresan qué repositorios y los métodos que debe implementar la capa de infraestructura. Es fundamental que la implementación de los repositorios se coloque fuera de la capa de modelo de dominio, en la biblioteca de capas de infraestructura, por lo que la capa de modelo de dominio no está "contaminada" por api o clases de tecnologías de infraestructura, como Entity Framework.

También puede ver una carpeta SeedWork que contiene clases base personalizadas que puede usar como base para las entidades de dominio y los objetos de valor, por lo que no tiene código redundante en la clase de objeto de cada dominio.

Estructuración de los agregados en una biblioteca personalizada de .NET Standard

Un agregado hace referencia a un clúster de objetos de dominio agrupados para que coincidan con la coherencia transaccional. Esos objetos podrían ser instancias de entidades (una de las cuales es la entidad raíz o raíz agregada) además de cualquier objeto de valor adicional.

La coherencia transaccional significa que se garantiza que un agregado sea coherente y actualizado al final de una acción empresarial. Por ejemplo, el agregado de pedidos del modelo de dominio de microservicios de eShopOnContainers está compuesto como se muestra en la Figura 7-11.

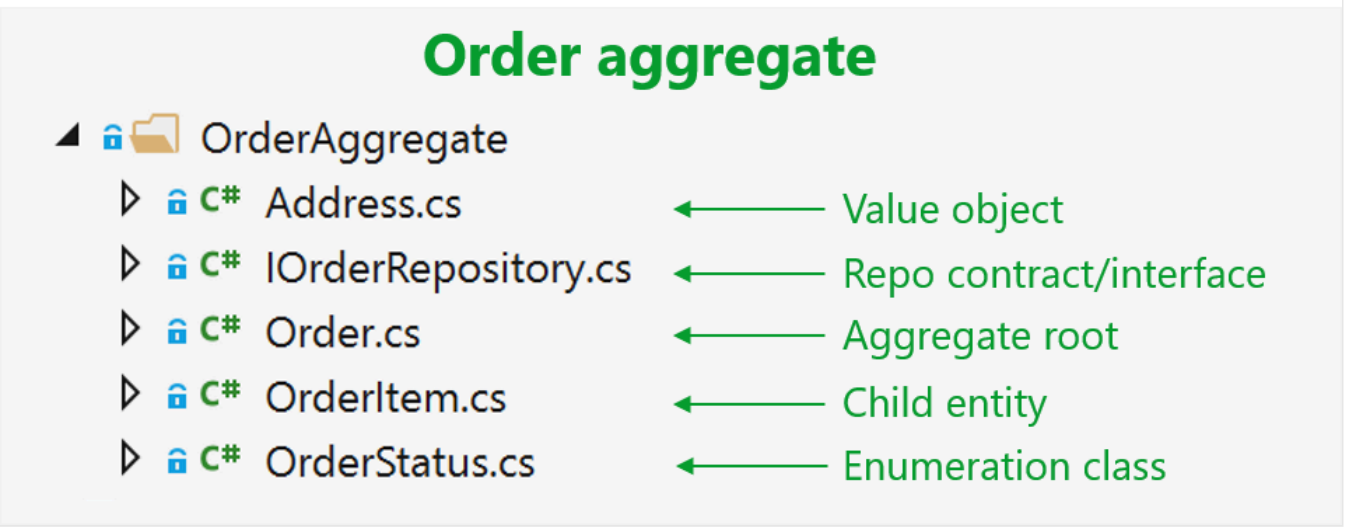


Figura 7-11. Agregado Order en la solución de Visual Studio

Si abre cualquiera de los archivos en una carpeta de agregado, puede ver cómo se marca como una clase base personalizada o una interfaz, como entidad o objeto de valor, tal como se implementa en la carpeta SeedWork .

Implementación de entidades de dominio como clases POCO

Para implementar un modelo de dominio en .NET, cree clases POCO que implementen las entidades de dominio. En el ejemplo siguiente, la clase Order se define como una entidad y también como una raíz de agregado. Dado que la clase Order deriva de la clase base Entity, puede reutilizar el código común relacionado con las entidades. Tenga en cuenta que estas interfaces y clases base se definen en el proyecto de modelo de dominio, por lo que es el código, no el código de infraestructura de un ORM como EF.

C#

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE 5.0
// Entity is a custom base class with the ID
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId;

    public OrderStatus OrderStatus { get; private set; }
```

```
    private int _orderStatusId;

    private string _description;
    private int? _paymentMethodId;

    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    public Order(string userId, Address address, int cardTypeId, string
cardNumber, string cardSecurityNumber,
            string cardHolderName, DateTime cardExpiration, int? buyerId = null,
int? paymentMethodId = null)
    {
        _orderItems = new List<OrderItem>();
        _buyerId = buyerId;
        _paymentMethodId = paymentMethodId;
        _orderStatusId = OrderStatus.Submitted.Id;
        _orderDate = DateTime.UtcNow;
        Address = address;

        // ...Additional code ...
    }

    public void AddOrderItem(int productId, string productName,
                             decimal unitPrice, decimal discount,
                             string pictureUrl, int units = 1)
    {
        //...
        // Domain rules/logic for adding the OrderItem to the order
        // ...

        var orderItem = new OrderItem(productId, productName, unitPrice, discount,
pictureUrl, units);

        _orderItems.Add(orderItem);

    }
    // ...
    // Additional methods with domain rules/logic related to the Order aggregate
    // ...
}
```

Es importante tener en cuenta que se trata de una entidad de dominio implementada como una clase POCO. No tiene ninguna dependencia directa de Entity Framework Core ni de ningún otro marco de infraestructura. Esta implementación es como debería estar en DDD, solo el código de C# que implementa un modelo de dominio.

Además, la clase está decorada con una interfaz denominada IAggregateRoot. Esa interfaz es una interfaz vacía, a veces denominada *interfaz de marcador*, que se usa solo para indicar que esta clase de entidad también es una raíz de agregado.

A veces, una interfaz de marcador se considera un antipatrón; sin embargo, también es una manera limpia de marcar una clase, especialmente cuando esa interfaz podría estar evolucionando. Un atributo podría ser la otra opción para el marcador, pero es más rápido ver la clase base (Entity) junto a la interfaz IAggregate en lugar de colocar un marcador de atributo Aggregate encima de la clase. Es una cuestión de preferencias, en cualquier caso.

Tener una raíz de agregado significa que la mayoría del código relacionado con las reglas de coherencia y negocios de las entidades del agregado se deben implementar como métodos en la clase raíz de agregado Order (por ejemplo, AddOrderItem al agregar un objeto OrderItem al agregado). No debe crear ni actualizar objetos OrderItems de forma independiente o directamente; La clase AggregateRoot debe mantener el control y la coherencia de cualquier operación de actualización en sus entidades secundarias.

Encapsular datos en las entidades de dominio

Un problema común en los modelos de entidad es que exponen las propiedades de navegación de colecciones como tipos de lista accesibles públicamente. Esto permite que cualquier desarrollador de colaboradores manipule el contenido de estos tipos de colección, lo que puede omitir reglas de negocios importantes relacionadas con la colección, posiblemente dejando el objeto en un estado no válido. La solución a esto consiste en exponer el acceso de solo lectura a colecciones relacionadas y proporcionar explícitamente métodos que definen formas en las que los clientes pueden manipularlas.

En el código anterior, tenga en cuenta que muchos atributos son de solo lectura o privados y solo son actualizables por los métodos de clase, por lo que cualquier actualización consideravariantes de dominio empresarial y lógica especificada dentro de los métodos de clase.

Por ejemplo, siguiendo los patrones DDD, ***no* debe hacer lo siguiente** desde ningún método de controlador de comandos o clase de capa de aplicación (en realidad, debería ser imposible hacerlo):

C#

```
// WRONG ACCORDING TO DDD PATTERNS – CODE AT THE APPLICATION LAYER OR
// COMMAND HANDLERS
// Code in command handler methods or Web API controllers
//... (WRONG) Some code with business logic out of the domain classes ...
OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName,
    pictureUrl, unitPrice, discount, units);

//... (WRONG) Accessing the OrderItems collection directly from the application
layer // or command handlers
```

```
myOrder.OrderItems.Add(myNewOrderItem);
//...
```

En este caso, el método Add es puramente una operación para agregar datos, con acceso directo a la colección OrderItems. Por lo tanto, la mayoría de la lógica de dominio, las reglas o las validaciones relacionadas con esa operación con las entidades secundarias se distribuirán en la capa de aplicación (controladores de comandos y controladores de API web).

Si omite la raíz de agregado, esta no puede garantizar sus invariables, su validez ni su coherencia. Al final se acaba con código espagueti o un código de script transaccional.

Para seguir los patrones DDD, las entidades no deben tener establecedores públicos en ninguna propiedad de entidad. Los cambios en una entidad deben estar controlados por métodos explícitos con lenguaje ubicuo explícito sobre el cambio que realizan en la entidad.

Además, las colecciones dentro de la entidad (como los elementos de pedido) deben ser propiedades de solo lectura (el método AsReadOnly explicado más adelante). Debe ser capaz de actualizarla solo desde los métodos de la clase raíz de agregado o los métodos de entidad secundaria.

Como puede ver en el código de la raíz de agregado Order, todos los establecedores deben ser privados o al menos de solo lectura externamente para que cualquier operación en los datos de la entidad o sus entidades secundarias tenga que realizarse mediante métodos en la clase de entidad. Esto mantiene la coherencia de una manera controlada y orientada a objetos en lugar de implementar código de script transaccional.

El siguiente fragmento de código muestra la manera adecuada de codificar la tarea de agregar un objeto OrderItem al agregado Order.

C#

```
// RIGHT ACCORDING TO DDD--CODE AT THE APPLICATION LAYER OR COMMAND HANDLERS
// The code in command handlers or WebAPI controllers, related only to application
stuff
// There is NO code here related to OrderItem object's business logic
myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount,
units);

// The code related to OrderItem params validations or domain rules should
// be WITHIN the AddOrderItem method.

//...
```

En este fragmento de código, la mayoría de las validaciones o lógica relacionadas con la creación de un objeto OrderItem estará bajo el control de la raíz de agregado Order, en el método AddOrderItem, especialmente validaciones y lógica relacionadas con otros elementos

del agregado. Por ejemplo, puede obtener el mismo elemento de producto como resultado de varias llamadas a AddOrderItem. En ese método, podría examinar los elementos de producto y consolidar los mismos elementos de producto en un único objeto OrderItem con varias unidades. Además, si hay diferentes cantidades de descuento, pero el identificador del producto es el mismo, es probable que aplique el mayor descuento. Este principio se aplica a cualquier otra lógica de dominio para el objeto OrderItem.

Además, la nueva operación OrderItem(params) también es controlada y realizada por el método AddOrderItem de la raíz de agregado Order. Por lo tanto, la mayoría de la lógica o las validaciones relacionadas con esa operación (especialmente todo lo que afecta a la coherencia entre otras entidades secundarias) estará en una única ubicación dentro de la raíz de agregado. Ese es el propósito final del patrón raíz agregado.

Cuando se usa Entity Framework Core 1.1 o posterior, se puede expresar mejor una entidad DDD porque permite la asignación a campos además de las propiedades. Esto resulta útil al proteger colecciones de entidades secundarias u objetos de valor. Con esta mejora, puede usar campos privados simples en lugar de propiedades y puede implementar cualquier actualización en la colección de campos en métodos públicos y proporcionar acceso de solo lectura a través del método AsReadOnly.

En DDD, se desea actualizar la entidad solo a través de los métodos dentro de la entidad (o el constructor) para controlar cualquier invariante y la consistencia de los datos, por lo que las propiedades solo se definen con un accesor get. Las propiedades están respaldadas por campos privados. A los miembros privados solo se puede acceder desde la clase. Sin embargo, hay una excepción: EF Core también debe establecer estos campos (por lo que puede devolver el objeto con los valores adecuados).

Asignación de propiedades con solo los descriptores de acceso get a los campos de la tabla de base de datos

La asignación de propiedades a columnas de tabla de base de datos no es responsabilidad de dominio, sino parte de la infraestructura y la capa de persistencia. Esto se menciona aquí para que conozca las nuevas funcionalidades de EF Core 1.1 o posterior relacionada con cómo puede modelar entidades. En la sección infraestructura y persistencia se explican detalles adicionales sobre este tema.

Cuando se usa EF Core 1.0 o posterior, en DbContext es necesario asignar las propiedades definidas únicamente con captadores a los campos reales de la tabla de base de datos. Esto se hace con el método HasField de la clase PropertyBuilder.

Mapear campos sin propiedades

Con la característica de EF Core 1.1 o posterior para asignar columnas a campos, también es posible no usar propiedades. En su lugar, solo puede asignar columnas de una tabla a campos. Un caso de uso común para esto es campos privados para un estado interno al que no es necesario tener acceso desde fuera de la entidad.

Por ejemplo, en el ejemplo de código OrderAggregate anterior, hay varios campos privados, como el `_paymentMethodId` campo , que no tienen ninguna propiedad relacionada para un establecedor o captador. Ese campo también se puede calcular dentro de la lógica de negocios del pedido y usarse a partir de los métodos del pedido, pero también debe conservarse en la base de datos. Por lo tanto, en EF Core (desde la versión 1.1), hay una manera de asignar un campo sin una propiedad relacionada a una columna de la base de datos. Esto también se explica en la sección Capa de infraestructura de esta guía.

Recursos adicionales

- **Vaughn Vernon. Modelado de agregados con DDD y Entity Framework.** Tenga en cuenta que *esto no* es Entity Framework Core. https://kalele.io/blog-posts/modeling-aggregates-with-ddd-and-entity-framework/
- **Julie Lerman. Puntos de datos: programación para un diseño guiado por el dominio: sugerencias para los desarrolladores enfocados en datos** https://learn.microsoft.com/archive/msdn-magazine/2013/august/data-points-coding-for-domain-driven-design-tips-for-data-focused-devs
- **Udi Dahan. Cómo crear modelos de dominio totalmente encapsulados** https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/
- **Steve Smith. ¿Cuál es la diferencia entre un DTO y un POCO?** \ https://ardalis.com/dto-or-poco/

Siguiente anterior

Seedwork (clases base reutilizables e interfaces para el modelo de dominio)

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

La carpeta de solución contiene una carpeta *SeedWork* . Esta carpeta contiene clases base personalizadas que puede usar como base para las entidades de dominio y los objetos de valor. Use estas clases base para que no tenga código redundante en la clase de objeto de cada dominio. La carpeta de estos tipos de clases se denomina *SeedWork* y no es algo parecido a *Framework*. Se denomina *SeedWork* porque la carpeta contiene solo un pequeño subconjunto de clases reutilizables que realmente no se pueden considerar un marco. *Seedwork* es un término introducido por Michael Feathers y popularizado por Martin Fowler, pero también podría nombrar esa carpeta Common, SharedKernel o similar.

La figura 7-12 muestra las clases que forman la infraestructura básica del modelo de dominio en el microservicio de pedidos. Tiene algunas clases base personalizadas, como `Entity`, `ValueObject` y `Enumeration`, además de algunas interfaces. Estas interfaces (`IRepository` y `IUnitOfWork`) informan al nivel de infraestructura sobre lo que debe implementarse. Estas interfaces también se usan mediante la inserción de dependencias del nivel de aplicación.

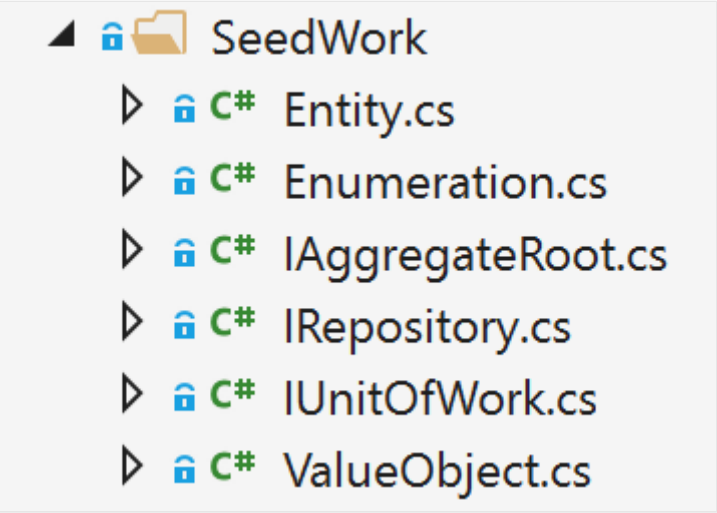


Figura 7-12. Un conjunto de ejemplos de clases e interfaces base del modelo de dominio "seedwork"

Este es el tipo de reutilización 'copy-paste' que muchos desarrolladores comparten entre proyectos, no un marco formal. Puede tener seedworks en cualquier nivel o biblioteca. Sin embargo, si el conjunto de clases e interfaces es lo suficientemente grande, es posible que quiera crear una sola biblioteca de clases.

La clase base de entidad personalizada

El código siguiente es un ejemplo de una clase base de entidad en la que puede colocar código que se puede usar de la misma manera por cualquier entidad de dominio, como el identificador de entidad, los operadores de igualdad, una lista de eventos de dominio por entidad, etc.

C#

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE (1.1 and later)
public abstract class Entity
{
    int? _requestedHashCode;
    int _Id;
    private List<INotification> _domainEvents;
    public virtual int Id
    {
        get
        {
            return _Id;
        }
        protected set
        {
            _Id = value;
        }
    }
```

```
    public List<INotification> DomainEvents => _domainEvents;
    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }
    public void RemoveDomainEvent(INotification eventItem)
    {
        if (_domainEvents is null) return;
        _domainEvents.Remove(eventItem);
    }

    public bool IsTransient()
    {
        return this.Id == default(Int32);
    }

    public override bool Equals(object obj)
    {
        if (obj == null || !(obj is Entity))
            return false;
        if (Object.ReferenceEquals(this, obj))
            return true;
        if (this.GetType() != obj.GetType())
            return false;
        Entity item = (Entity)obj;
        if (item.IsTransient() || this.IsTransient())
            return false;
        else
            return item.Id == this.Id;
    }

    public override int GetHashCode()
    {
        if (!IsTransient())
        {
            if (!_requestedHashCode.HasValue)
                _requestedHashCode = this.Id.GetHashCode() ^ 31;
            // XOR for random distribution. See:
            // https://learn.microsoft.com/archive/blogs/ericlippert/guidelines-
and-rules-for-gethashcode
            return _requestedHashCode.Value;
        }
        else
            return base.GetHashCode();
    }
    public static bool operator ==(Entity left, Entity right)
    {
        if (Object.Equals(left, null))
            return (Object.Equals(right, null));
        else
            return left.Equals(right);
    }
    public static bool operator !=(Entity left, Entity right)
    {
```

```
        return !(left == right);
    }
}
```

El código anterior que usa una lista de eventos de dominio por entidad se explicará en las secciones siguientes al centrarse en eventos de dominio.

Contratos de repositorio (interfaces) en la capa de modelo de dominio

Los contratos de repositorio son simplemente interfaces de .NET que expresan los requisitos de contrato de los repositorios que se usarán para cada agregado.

Los propios repositorios, con código de EF Core o cualquier otra dependencia y código de infraestructura (Linq, SQL, etc.), no deben implementarse dentro del modelo de dominio; los repositorios solo deben implementar las interfaces que defina en el modelo de dominio.

Un patrón relacionado con esta práctica (colocar las interfaces del repositorio en la capa de modelo de dominio) es el patrón de interfaz separada. Como explica Martin Fowler, "Usar interfaz separada para definir una interfaz en un paquete, pero implementarla en otra. De este modo, un cliente que necesita la dependencia a la interfaz puede no ser consciente de la implementación".

Si sigue el patrón de interfaz separada, la capa de aplicación (en este caso, el proyecto de API web para el microservicio) tiene una dependencia de los requisitos definidos en el modelo de dominio, pero no una dependencia directa a la capa de infraestructura o persistencia. Además, puede usar la inserción de dependencias para aislar la implementación, que se implementa en la capa de infraestructura o persistencia mediante repositorios.

Por ejemplo, en el ejemplo siguiente con la interfaz IOrderRepository se define qué operaciones tendrá que implementar la clase OrderRepository en el nivel de infraestructura. En la implementación actual de la aplicación, el código solo necesita agregar o actualizar pedidos a la base de datos, ya que las consultas se dividen siguiendo el enfoque simplificado de CQRS.

C#

```
// Defined at IOrderRepository.cs
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);

    void Update(Order order);

    Task<Order> GetAsync(int orderId);
```

```
}

// Defined at IRepository.cs (Part of the Domain Seedwork)
public interface IRepository<T> where T : IAggregateRoot
{
    IUnitOfWork UnitOfWork { get; }
}
```

Recursos adicionales

- **Martin Fowler. Interfaz separada.**
  https://www.martinfowler.com/eaaCatalog/separatedInterface.html

Siguiente anterior

Implementación de objetos de valor

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Como se explicó en secciones anteriores sobre entidades y agregados, la identidad es fundamental para las entidades. Sin embargo, hay muchos objetos y elementos de datos en un sistema que no requieren identidad ni seguimiento de identidad, como los objetos de valor.

Un objeto value puede hacer referencia a otras entidades. Por ejemplo, en una aplicación que genera una ruta que describe cómo obtener de un punto a otro, esa ruta sería un objeto de valor. Sería una instantánea de puntos en una ruta específica, pero esta ruta sugerida no tendría una identidad, aunque internamente podría hacer referencia a entidades como City, Road, etc.

En la figura 7-13 se muestra el objeto de valor Address dentro del agregado Order.

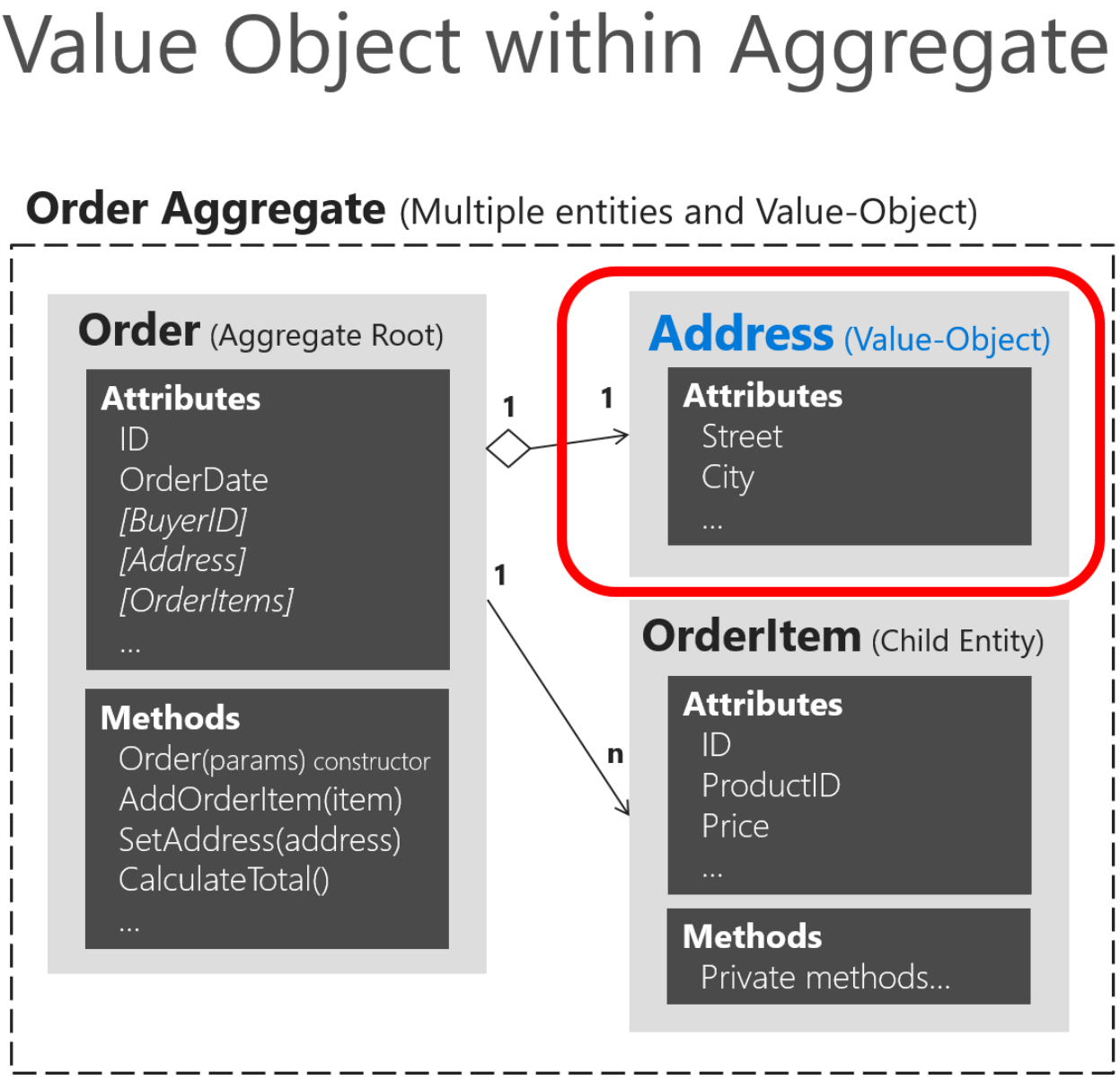


Figura 7-13. Objeto de valor Dirección en el agregado Pedido

Como se muestra en la figura 7-13, una entidad normalmente se compone de varios atributos. Por ejemplo, la `Order` entidad se puede modelar como una entidad con una identidad y compuesta internamente de un conjunto de atributos como OrderId, OrderDate, OrderItems, etc. Pero la dirección, que es simplemente un valor complejo compuesto por país o región, calle, ciudad, etc., y no tiene identidad en este dominio, debe modelarse y tratarse como un objeto de valor.

Características importantes de los objetos de valor

Hay dos características principales para los objetos de valor:

- No tienen identidad.
- Son inmutables.

Ya se ha analizado la primera característica. La inmutabilidad es un requisito importante. Los valores de un objeto de valor deben ser inmutables una vez creado el objeto. Por lo tanto, cuando se construye el objeto, debe proporcionar los valores necesarios, pero no debe permitirles cambiar durante la vigencia del objeto.

Los objetos value permiten realizar ciertos trucos para el rendimiento, gracias a su naturaleza inmutable. Esto es especialmente cierto en los sistemas en los que puede haber miles de instancias de objeto de valor, muchas de las cuales tienen los mismos valores. Su naturaleza inmutable les permite reutilizarse; pueden ser objetos intercambiables, ya que sus valores son los mismos y no tienen identidad. Este tipo de optimización a veces puede marcar la diferencia entre el software que se ejecuta lentamente y el software con un buen rendimiento. Por supuesto, todos estos casos dependen del entorno de aplicación y del contexto de implementación.

Implementación de objetos de valor en C#

En términos de implementación, puede tener una clase base de objeto de valor que tenga métodos de utilidad básicos como la igualdad en función de la comparación entre todos los atributos (ya que un objeto de valor no debe basarse en la identidad) y otras características fundamentales. En el ejemplo siguiente se muestra una clase base de objeto de valor utilizada en el microservicio de ordenación de eShopOnContainers.

C#

```
public abstract class ValueObject
{
    protected static bool EqualOperator(ValueObject left, ValueObject right)
    {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
        {
            return false;
        }
        return ReferenceEquals(left, right) || left.Equals(right);
    }

    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }

    protected abstract IEnumerable<object> GetEqualityComponents();

    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
```

```
        }

        var other = (ValueObject)obj;

        return
this.GetEqualityComponents().SequenceEqual(other.GetEqualityComponents());
    }

    public override int GetHashCode()
    {
        return GetEqualityComponents()
            .Select(x => x != null ? x.GetHashCode() : 0)
            .Aggregate((x, y) => x ^ y);
    }
    // Other utility methods
}
```

`ValueObject` es un tipo `abstract class`, pero en este ejemplo, no sobrecarga los operadores `==` y `!=`. Puede optar por hacerlo, haciendo que las comparaciones se deleguen en la invalidación `Equals`. Por ejemplo, considere las siguientes sobrecargas de operador en el tipo `ValueObject`:

C#

```
public static bool operator ==(ValueObject one, ValueObject two)
{
    return EqualOperator(one, two);
}

public static bool operator !=(ValueObject one, ValueObject two)
{
    return NotEqualOperator(one, two);
}
```

Puede usar esta clase al implementar su objeto de valor real, como el objeto de valor `Address` que se muestra en el ejemplo siguiente:

C#

```
public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }

    public Address() { }

    public Address(string street, string city, string state, string country,
```

```
string zipcode)
    {
        Street = street;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;
    }

    protected override IEnumerable<object> GetEqualityComponents()
    {
        // Using a yield return statement to return each element one at a time
        yield return Street;
        yield return City;
        yield return State;
        yield return Country;
        yield return ZipCode;
    }
}
```

Esta implementación de objeto de valor de `Address` no tiene identidad y, por tanto, no se define ningún campo de identificador para él, ya sea en la `Address` definición de clase o en la definición de clase `ValueObject` .

No era posible tener ningún campo de identificador en una clase que Entity Framework (EF) usara hasta EF Core 2.0, lo que ayuda en gran medida a implementar objetos de mejor valor sin identificador. Es precisamente la explicación de la sección siguiente.

Podría decirse que los objetos de valor, que son inmutables, deben ser de solo lectura (es decir, tener propiedades get-only) y eso es cierto. Sin embargo, los objetos de valor normalmente se serializan y deserializan para pasar por las colas de mensajes y, al ser de solo lectura, se impide que el deserializador asigne valores, por lo que simplemente los deja como `private set`, lo que es lo suficientemente de solo lectura como para ser práctico.

Semántica de comparación de objetos de valor

Se pueden comparar dos instancias del `Address` tipo mediante todos los métodos siguientes:

C#

```
var one = new Address("1 Microsoft Way", "Redmond", "WA", "US", "98052");
var two = new Address("1 Microsoft Way", "Redmond", "WA", "US", "98052");

Console.WriteLine(EqualityComparer<Address>.Default.Equals(one, two)); // True
Console.WriteLine(object.Equals(one, two)); // True
Console.WriteLine(one.Equals(two)); // True
Console.WriteLine(one == two); // True
```

Cuando todos los valores son iguales, las comparaciones se evalúan correctamente como `true`. Si no eligió sobrecargar los operadores `==` y `!=`, entonces la última comparación de `one == two` se evaluaría como `false`. Para obtener más información, vea Sobrecarga de los operadores de igualdad de ValueObject.

Cómo conservar objetos de valor en la base de datos con EF Core 2.0 y versiones posteriores

Acaba de ver cómo definir un objeto de valor en el modelo de dominio. Pero ¿cómo puede conservarlo en la base de datos mediante Entity Framework Core, dado que suele tener como destino las entidades con identidad?

Contexto histórico y los enfoques anteriores utilizando EF Core 1.1

Como fondo, una limitación al usar EF Core 1.0 y 1.1 era que no podía usar tipos complejos tal como se define en EF 6.x en .NET Framework tradicional. Por lo tanto, si usa EF Core 1.0 o 1.1, debe almacenar el objeto de valor como una entidad ef con un campo id. A continuación, por lo que parecía un objeto de valor sin identidad, podría ocultar su identificador para aclarar que la identidad de un objeto de valor no es importante en el modelo de dominio. Ese identificador se podía ocultar usándolo como propiedad reemplazada. Dado que esa configuración para ocultar el identificador en el modelo se establece en el nivel de infraestructura de EF, sería bastante transparente para tu modelo de dominio.

En la versión inicial de eShopOnContainers (.NET Core 1.1), el identificador oculto necesario para la infraestructura de EF Core se implementó de la siguiente manera en el nivel dbContext, mediante la API de Fluent en el proyecto de infraestructura. Por lo tanto, el identificador se ha ocultado desde el punto de vista del modelo de dominio, pero sigue presente en la infraestructura.

C#

```
// Old approach with EF Core 1.1
// Fluent API within the OrderingContext:DbContext in the Infrastructure project
void ConfigureAddress(EntityTypeBuilder<Address> addressConfiguration)
{
    addressConfiguration.ToTable("address", DEFAULT_SCHEMA);

    addressConfiguration.Property<int>("Id")  // Id is a shadow property
        .IsRequired();
    addressConfiguration.HasKey("Id");   // Id is a shadow property
}
```

Sin embargo, la persistencia de ese objeto de valor en la base de datos se realizó como una entidad regular en una tabla diferente.

Con EF Core 2.0 y versiones posteriores, hay nuevas y mejores formas de conservar objetos de valor.

Conservar objetos de valor como tipos de entidad propiedad en EF Core 2.0 y versiones posteriores

Incluso con algunas brechas entre el patrón de objetos de valor canónico en DDD y el tipo de entidad propiedad de EF Core, actualmente es la mejor manera de conservar objetos de valor con EF Core 2.0 y versiones posteriores. Puede ver las limitaciones al final de esta sección.

La característica de tipo de entidad propiedad se agregó a EF Core desde la versión 2.0.

Un tipo de entidad propiedad permite asignar tipos que no tienen su propia identidad definida explícitamente en el modelo de dominio y se usan como propiedades, como un objeto de valor, dentro de cualquiera de las entidades. Un tipo de entidad propiedad comparte el mismo tipo CLR con otro tipo de entidad (es decir, es simplemente una clase normal). La entidad que contiene la navegación definitoria es la entidad propietaria. Al consultar al propietario, los tipos de propiedad se incluyen de forma predeterminada.

Si se examina el modelo de dominio, parece que los tipos de propiedad no tienen ninguna identidad pero, en el fondo, la tienen, aunque la propiedad de navegación del propietario forma parte de esta identidad.

La identidad de las instancias de los tipos de propiedad no es totalmente suya. Consta de tres componentes:

- Identidad del propietario
- La propiedad de navegación que los señala
- En el caso de las colecciones de tipos de propiedad, un componente independiente (compatible con EF Core 2.2 y versiones posteriores).

Por ejemplo, en el modelo de dominio Ordering de eShopOnContainers, como parte de la entidad Order, el objeto Address value se implementa como un tipo de entidad propiedad dentro de la entidad propietaria, que es la entidad Order. `Address` es un tipo sin ninguna propiedad de identidad definida en el modelo de dominio. Se usa como propiedad del tipo Order para especificar la dirección de envío de un pedido en concreto.

Por convención, se crea una clave principal paralela para el tipo de propiedad y se asignará a la misma tabla que el propietario mediante la división de tabla. Esto permite usar tipos de propiedad de forma similar a cómo se usan los tipos complejos en EF6 en .NET Framework tradicional.

Es importante tener en cuenta que los tipos de propiedad nunca se detectan por convención en EF Core, por lo que debe declararlos explícitamente.

En eShopOnContainers, en el archivo OrderingContext.cs, dentro del `OnModelCreating()` método , se aplican varias configuraciones de infraestructura. Uno de ellos está relacionado con la entidad Order.

C#

```
// Part of the OrderingContext.cs class at the Ordering.Infrastructure project
//
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new ClientRequestEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new PaymentMethodEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderItemEntityTypeConfiguration());
    //...Additional type configurations
}
```

En el código siguiente, la infraestructura de persistencia se define para la entidad Order:

C#

```
// Part of the OrderEntityTypeConfiguration.cs class
//
public void Configure(EntityTypeBuilder<Order> orderConfiguration)
{
    orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
    orderConfiguration.HasKey(o => o.Id);
    orderConfiguration.Ignore(b => b.DomainEvents);
    orderConfiguration.Property(o => o.Id)
        .ForSqlServerUseSequenceHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

    //Address value object persisted as owned entity in EF Core 2.0
    orderConfiguration.OwnsOne(o => o.Address);

    orderConfiguration.Property<DateTime>("OrderDate").IsRequired();

    //...Additional validations, constraints and code...
    //...
}
```

En el código anterior, el `orderConfiguration.OwnsOne(o => o.Address)` método especifica que la `Address` propiedad es una entidad propiedad del `Order` tipo.

De forma predeterminada, las convenciones de EF Core asignan a las columnas de base de datos el nombre de las propiedades del tipo de entidad propiedad como `EntityProperty_OwnedEntityProperty`. Por lo tanto, las propiedades internas de `Address` aparecerán en la `Orders` tabla con los nombres `Address_Street`, `Address_City` (y así sucesivamente para `State`, `Country` y `ZipCode`).

Puede adjuntar el método fluent `Property().HasColumnName()` para cambiar el nombre de esas columnas. En el caso en que `Address` es una propiedad pública, las asignaciones serían similares a lo siguiente:

C#

```
orderConfiguration.OwnsOne(p => p.Address)

.Property(p=>p.Street).HasColumnName("ShippingStreet");

orderConfiguration.OwnsOne(p => p.Address)
                            .Property(p=>p.City).HasColumnName("ShippingCity");
```

Es posible encadenar el método `OwnsOne` en un mapeo fluido. En el ejemplo hipotético siguiente, `OrderDetails` posee `BillingAddress` y `ShippingAddress`, que son ambos `Address` tipos. Luego, `OrderDetails` es propiedad del tipo `Order`.

C#

```
orderConfiguration.OwnsOne(p => p.OrderDetails, cb =>
    {
        cb.OwnsOne(c => c.BillingAddress);
        cb.OwnsOne(c => c.ShippingAddress);
    });
//...
//...
public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}

public class OrderDetails
{
    public Address BillingAddress { get; set; }
    public Address ShippingAddress { get; set; }
}

public class Address
```

```
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

Detalles adicionales sobre los tipos de entidad de propiedad

- Los tipos de propiedad se definen al configurar una propiedad de navegación en un tipo determinado mediante la API fluida OwnsOne.
- La definición de un tipo propio en nuestro modelo de metadatos es una composición de: el tipo de propietario, la propiedad de navegación y el tipo CLR del tipo propio.
- La identidad (clave) de una instancia de tipo de propiedad en nuestra pila es una composición de la identidad del tipo de propietario y la definición del tipo de propiedad.

Capacidades de entidades propias

- Los tipos de propiedad pueden hacer referencia a otras entidades, ya sean de propiedad (tipos de propiedad anidados) o de no propiedad (propiedades de navegación de referencia normal a otras entidades).
- Se puede asignar el mismo tipo CLR como otros tipos de propiedad en la misma entidad de propietario mediante propiedades de navegación independientes.
- La división de tablas se configura por convención, pero puede dejar de usarla si asigna el tipo de propiedad a otra tabla mediante ToTable.
- En los tipos de propiedad se efectúa una carga diligente de forma automática; es decir, no es necesario llamar a `.Include()` en la consulta.
- Se puede configurar con el atributo `[Owned]`, mediante EF Core 2.1 y versiones posteriores.
- Puede controlar colecciones de tipos de propiedad (con la versión 2.2 y posteriores).

Limitaciones de las entidades de propiedad

- No se puede crear un elemento `DbSet<T>` de un tipo de propiedad (por cuestiones de diseño).
- No se puede llamar a `ModelBuilder.Entity<T>()` en los tipos de propiedad (actualmente por cuestiones de diseño).

- No se admiten los tipos de propiedad opcionales (es decir, que aceptan valores NULL) que se asignan con el propietario en la misma tabla (es decir, mediante la división de tablas). Esto se debe a que la asignación se realiza para cada propiedad; no hay un centinela independiente para el valor complejo NULL como un todo.
- No hay compatibilidad con la asignación de herencia para los tipos de propiedad, pero se deberían poder asignar dos tipos hoja de las mismas jerarquías de herencia como otros tipos de propiedad. EF Core no razonará sobre el hecho de que forman parte de la misma jerarquía.

Principales diferencias con los tipos complejos de EF6

- La división de tablas es opcional (es decir, opcionalmente se pueden asignar a una tabla independiente y seguir siendo tipos de propiedad).

Recursos adicionales

- **Martin Fowler. ValueObject pattern (El patrón ValueObject)**
  https://martinfowler.com/bliki/ValueObject.html
- **Eric Evans. "Domain-Driven Design: Tackling Complexity in the Heart of Software" (Diseño orientado al dominio: abordar la complejidad en el corazón del software).** (Libro; incluye una explicación de los objetos de valor)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/
- **Vaughn Vernon. Implementación del diseño orientado al dominio.** (Libro; incluye una explicación de los objetos de valor)
  https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/
- **Tipos de entidad propiedad**
  https://learn.microsoft.com/ef/core/modeling/owned-entities
- **Propiedades de sombra**
  https://learn.microsoft.com/ef/core/modeling/shadow-properties
- **Tipos complejos o objetos de valor**. Descripción en el repositorio de GitHub de EF Core (pestaña Problemas)
  https://github.com/dotnet/efcore/issues/246
- **ValueObject.cs.** Clase de objeto de valor base en eShopOnContainers.
  https://github.com/dotnet-

architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs

- **ValueObject.cs.** Clase de objeto de valor base en CSharpFunctionalExtensions. https://github.com/vkhorikov/CSharpFunctionalExtensions/blob/master/CSharpFunctionalExtensions/ValueObject/ValueObject.cs

- **Clase Address.** Clase de objeto de valor de ejemplo en eShopOnContainers. https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Address.cs

Siguiente anterior

Utilice clases de enumeración en lugar de tipos enumerados

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Las enumeraciones (o *los tipos de enumeración* abreviadamente) son un envoltorio de lenguaje ligero alrededor de un tipo integral. Es posible que quiera limitar su uso a cuando se almacena un valor de un conjunto cerrado de valores. La clasificación basada en tamaños (pequeños, medianos y grandes) es un buen ejemplo. Usar las enumeraciones para el flujo de control o abstracciones más sólidas puede producir un problema en el código. Este tipo de uso conduce a código frágil con muchas instrucciones de flujo de control que comprueban los valores de la enumeración.

En su lugar, puede crear clases de enumeración que permitan todas las características enriquecidas de un lenguaje orientado a objetos.

Sin embargo, esto no es un tema crítico y, en muchos casos, por motivos de simplicidad, puede seguir usando tipos de enumeración normales si es su preferencia. El uso de clases de enumeración está más relacionado con los conceptos relacionados con la empresa.

Implementación de una clase base de enumeración

El microservicio de ordenación en eShopOnContainers proporciona una implementación de clase base de enumeración de ejemplo, como se muestra en el ejemplo siguiente:

C#

```
public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration(int id, string name) => (Id, Name) = (id, name);

    public override string ToString() => Name;

    public static IEnumerable<T> GetAll<T>() where T : Enumeration =>
        typeof(T).GetFields(BindingFlags.Public |
                            BindingFlags.Static |
                            BindingFlags.DeclaredOnly)
                 .Select(f => f.GetValue(null))
                 .Cast<T>();

    public override bool Equals(object obj)
    {
        if (obj is not Enumeration otherValue)
        {
            return false;
        }

        var typeMatches = GetType().Equals(obj.GetType());
        var valueMatches = Id.Equals(otherValue.Id);

        return typeMatches && valueMatches;
    }

    public int CompareTo(object other) => Id.CompareTo(((Enumeration)other).Id);

    // Other utility methods ...
}
```

Puede usar esta clase como un tipo en cualquier entidad o objeto de valor, como para la siguiente `CardType` clase : `Enumeration`

C#

```
public class CardType
    : Enumeration
{
    public static CardType Amex = new(1, nameof(Amex));
    public static CardType Visa = new(2, nameof(Visa));
    public static CardType MasterCard = new(3, nameof(MasterCard));

    public CardType(int id, string name)
        : base(id, name)
    {
```

```
    }
}
```

Recursos adicionales

- **Jimmy Bogard. Enumeration classes (Clases de enumeración)**
  https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/
- **Steve Smith. Enum Alternatives in C# (Alternativas de enumeración en C#)**
  https://ardalis.com/enum-alternatives-in-c
- **Enumeration.cs.** Base Enumeration class in eShopOnContainers (Clase base de enumeración en eShopOnContainers)
  https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/Enumeration.cs
- **CardType.cs**. Sample Enumeration class in eShopOnContainers (CardType.cs. Clase de enumeración de ejemplo en eShopOnContainers)
  https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/AggregatesModel/BuyerAggregate/CardType.cs
- **SmartEnum**. Ardalis - Classes to help produce strongly typed smarter enums in .NET. (Ardalis: clases para ayudar a crear enumeraciones fuertemente tipadas de manera más inteligente en .NET)
  https://www.nuget.org/packages/Ardalis.SmartEnum/

Siguiente anterior

Diseño de validaciones en la capa de modelo de dominio

Artículo • 21/05/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En DDD, las reglas de validación se pueden considerar invariables. La responsabilidad principal de un agregado es aplicar invariantes a lo largo de los cambios de estado para todas las entidades dentro de ese agregado.

Las entidades de dominio siempre deben ser entidades válidas. Hay un número determinado de invariables para un objeto que siempre deben ser verdaderas. Por ejemplo, un objeto de artículo de pedido siempre tiene que tener una cantidad que debe ser un entero positivo, además de un nombre de artículo y un precio. Por lo tanto, la aplicación de invariantes es responsabilidad de las entidades de dominio (especialmente de la raíz agregada) y un objeto de entidad no debe poder existir sin ser válido. Las reglas invariables se expresan simplemente como contratos y se generan excepciones o notificaciones cuando se infringen.

El razonamiento detrás de esto es que muchos errores se producen porque los objetos están en un estado en el que nunca deberían haber estado.

Supongamos que ahora tenemos un servicio de envío de correos electrónicos de creación de usuarios que acepta un perfil de usuario... ¿cómo podemos justificar en ese servicio que el nombre no sea nulo? ¿Lo revisamos de nuevo? O más probable... no te molestas en comprobar y "esperar lo mejor": esperas que alguien se moleste en validarlo antes de enviarlo a ti. Por supuesto, al usar TDD, una de las primeras pruebas que deberíamos escribir es que, si envío un

cliente con un nombre nulo, debería generar un error. Pero una vez que empezamos a escribir estos tipos de pruebas de nuevo, nos damos cuenta ... "¿Qué ocurre si nunca permitimos que el nombre se convierta en null? no tendríamos todas estas pruebas!".

Implementación de validaciones en la capa de modelo de dominio

Las validaciones normalmente se implementan en constructores de entidades de dominio o en métodos que pueden actualizar la entidad. Hay varias maneras de implementar validaciones, como comprobar los datos y generar excepciones si se produce un error en la validación. También hay patrones más avanzados, como el uso del patrón de especificación para las validaciones, y el patrón de notificación para devolver una colección de errores en lugar de devolver una excepción para cada validación a medida que se produce.

Validar las condiciones y lanzar excepciones

En el ejemplo de código siguiente se muestra el enfoque más sencillo para la validación en una entidad de dominio mediante la generación de una excepción. En la tabla de referencias al final de esta sección puede ver vínculos a implementaciones más avanzadas basadas en los patrones que hemos analizado anteriormente.

C#

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
```

Un ejemplo mejor demostraría la necesidad de asegurarse de que el estado interno no cambió o que se produjeron todas las mutaciones de un método. Por ejemplo, la siguiente implementación dejaría el objeto en un estado no válido:

C#

```
public void SetAddress(string line1, string line2,
    string city, string state, int zip)
{
    _shippingAddress.line1 = line1 ?? throw new ...
    _shippingAddress.line2 = line2;
    _shippingAddress.city = city ?? throw new ...
    _shippingAddress.state = (IsValid(state) ? state : throw new …);
}
```

Si el valor del estado no es válido, ya se han modificado la ciudad y la primera línea de dirección. Esto podría hacer que la dirección no sea válida.

Se puede usar un enfoque similar en el constructor de la entidad, lo que genera una excepción para asegurarse de que la entidad es válida una vez creada.

Uso de atributos de validación en el modelo en función de las anotaciones de datos

Las anotaciones de datos, como los atributos Required o MaxLength, se pueden usar para configurar las propiedades del campo de base de datos de EF Core, como se explica en detalle en la sección Asignación de tablas, pero ya no funcionan para la validación de entidades en EF Core (ni para el IValidatableObject.Validate método), como han hecho desde EF 4.x en .NET Framework.

Las anotaciones de datos y la interfaz IValidatableObject todavía se pueden usar para la validación del modelo durante el enlace del modelo, antes de la invocación habitual de las acciones del controlador, pero ese modelo está pensado para ser un ViewModel o DTO, y eso es un problema de MVC o API, no un problema de modelo de dominio.

Después de aclarar la diferencia conceptual, todavía puede usar anotaciones de datos y `IValidatableObject` en la clase de entidad para la validación si sus acciones reciben un objeto de clase de entidad como parámetro, lo cual no se recomienda. En ese caso, la validación se producirá en el enlace del modelo, justo antes de invocar la acción, y puede comprobar la propiedad ModelState.IsValid del controlador para comprobar el resultado, pero, de nuevo, ocurre en el controlador, no antes de persistir el objeto de entidad en el DbContext, como se hacía desde EF 4.x.

Todavía puede implementar la validación personalizada en la clase de entidad mediante anotaciones de datos y el método `IValidatableObject.Validate`, sobrescribiendo el método SaveChanges de DbContext.

Puede ver una implementación de ejemplo para validar `IValidatableObject` entidades en este comentario en GitHub. Ese ejemplo no realiza validaciones basadas en atributos, pero deberían ser fáciles de implementar mediante la reflexión en la misma sobrescritura.

Sin embargo, desde un punto de vista de DDD, es mejor mantener el modelo de dominio sencillo utilizando excepciones en los métodos de comportamiento de la entidad, o implementando el patrón de especificación y notificación para imponer reglas de validación.

Puede tener sentido usar anotaciones de datos en el nivel de aplicación en las clases ViewModel (en lugar de entidades de dominio) que aceptarán la entrada, para permitir la

validación del modelo dentro de la capa de interfaz de usuario. Sin embargo, esto no debe realizarse a costa de la validación dentro del modelo de dominio.

Validación de entidades mediante la implementación del patrón de especificación y el patrón de notificación

Por último, un enfoque más elaborado para implementar validaciones en el modelo de dominio es mediante la implementación del patrón de especificación junto con el patrón de notificación, como se explica en algunos de los recursos adicionales enumerados más adelante.

Merece la pena mencionar que también puede usar solo uno de esos patrones, por ejemplo, validar manualmente con instrucciones de control, pero usar el patrón de notificación para apilar y devolver una lista de errores de validación.

Uso de la validación diferida en el dominio

Hay varios enfoques para tratar las validaciones diferidas en el dominio. En su libro Implementación del Diseño Domain-Driven, Vaughn Vernon describe estos en la sección sobre validación.

Validación en dos pasos

Considere también la verificación en dos pasos. Utiliza la validación de nivel de campo en tus Objetos de Transferencia de Datos (DTOs) y la validación de nivel de dominio dentro de tus entidades. Para ello, devuelva un objeto de resultado en lugar de excepciones para facilitar el control de los errores de validación.

El uso de la validación de campos con anotaciones de datos, por ejemplo, no duplica la definición de validación. Sin embargo, la ejecución puede ser del lado servidor y del lado cliente en el caso de DTO (comandos y ViewModels, por ejemplo).

Recursos adicionales

- **Rachel Appel. Introducción a la validación de modelos en ASP.NET Core MVC**
  https://learn.microsoft.com/aspnet/core/mvc/models/validation
- **Adición de validación**
  https://learn.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation
- **Martin Fowler. Reemplazar el lanzamiento de excepciones por notificaciones en validaciones**

https://martinfowler.com/articles/replaceThrowWithNotification.html

- **Patrones de especificación y notificación**
  https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns

- **Lev Gorodinski. Validación en Diseño Domain-Driven (DDD)**
  http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/

- **Colin Jack. Validación del modelo de dominio**
  https://colinjack.blogspot.com/2008/03/domain-model-validation.html

- **Jimmy Bogard. Validación en el mundo de DDD**
  https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/

Siguiente anterior

Validación del lado cliente (validación en las capas de presentación)

Artículo • 21/05/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Incluso cuando el origen de la verdad es el modelo de dominio y, en última instancia, debe tener validación en el nivel de modelo de dominio, la validación todavía se puede controlar tanto en el nivel de modelo de dominio (lado servidor) como en la interfaz de usuario (lado cliente).

La validación del lado cliente es una gran comodidad para los usuarios. Ahorra tiempo que, de lo contrario, dedicarían a esperar un recorrido de ida y vuelta al servidor que podría devolver errores de validación. En términos empresariales, incluso unas pocas fracciones de segundos multiplicados cientos de veces cada día se suma a una gran cantidad de tiempo, gastos y frustración. La validación directa e inmediata permite a los usuarios trabajar de forma más eficaz y producir una mejor entrada y salida de calidad.

Al igual que el modelo de vista y el modelo de dominio son diferentes, la validación del modelo de vista y la validación del modelo de dominio pueden ser similares, pero sirven para un propósito diferente. Si le preocupa DRY (el principio No repetirse), tenga en cuenta que, en este caso, la reutilización del código también puede implicar acoplamiento y, en las aplicaciones empresariales, es más importante evitar acoplar el servidor al cliente que seguir el principio DRY.

Incluso al usar la validación del lado cliente, siempre debe validar los comandos o los DTO de entrada en el código de servidor, ya que las API de servidor son un vector de ataque posible. Normalmente, hacer ambas es la mejor opción porque si tiene una aplicación cliente, desde una perspectiva de la experiencia de usuario, es mejor ser proactivo y no permitir que el usuario escriba información no válida.

Por lo tanto, en el código del lado cliente, normalmente se validan los ViewModels. También puede validar los DTO o comandos de salida del cliente antes de enviarlos a los servicios.

La implementación de la validación del lado cliente depende del tipo de aplicación cliente que está desarrollando. Será diferente si va a validar los datos en una aplicación web MVC con la mayoría del código de .NET frente a una aplicación web SPA con el código de validación en JavaScript o TypeScript.

Recursos adicionales

Validación en aplicaciones de ASP.NET Core

- **Adición de validación**
  https://learn.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation

Validación en aplicaciones web SPA (Angular 2, TypeScript, JavaScript, Blazor WebAssembly)

- **Validación de formularios**
  https://angular.io/guide/form-validation
- **Validación.** Documentación de Breeze.
  https://breeze.github.io/doc-js/validation.html
- **Formularios y componentes de entrada de ASP.NET Core Blazor** \
  </aspnet/core/blazor/forms-and-input-components>

En resumen, estos son los conceptos más importantes en lo que respecta a la validación:

- Las entidades y agregados deben aplicar su propia coherencia y ser "siempre válidas". Las raíces agregadas son responsables de la coherencia de varias entidades dentro del mismo agregado.
- Si cree que una entidad necesita entrar en un estado no válido, considere usar un modelo de objeto diferente, por ejemplo, un DTO temporal hasta que cree la entidad de dominio final.

- Si necesita crear varios objetos relacionados, como un agregado, y solo son válidos una vez creados todos ellos, considere la posibilidad de usar el patrón Factory.
- En la mayoría de los casos, tener una validación redundante en el lado cliente es bueno, ya que la aplicación puede ser proactiva.

Siguiente anterior

Eventos de dominio: diseño e implementación

Artículo • 29/03/2023

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en **Documentación de .NET** o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Uso de eventos de dominio para implementar explícitamente los efectos secundarios de los cambios en el dominio. En otras palabras y con la terminología de DDD, usar eventos de dominio para implementar explícitamente los efectos secundarios entre varios agregados. Opcionalmente, para una mejor escalabilidad y un menor impacto en los bloqueos de base de datos, use la coherencia final entre agregados dentro del mismo dominio.

¿Qué es un evento de dominio?

Un evento es algo que ha sucedido en el pasado. Un evento de dominio es algo que ha sucedido en el dominio que quiere que otras partes del mismo dominio (en curso) tengan en cuenta. Normalmente las partes notificadas reaccionan de alguna manera a los eventos.

Una ventaja importante de los eventos de dominio es que los efectos secundarios se pueden expresar explícitamente.

Por ejemplo, si simplemente usa Entity Framework y debe haber una reacción a algún evento, probablemente codificaría cualquier cosa que necesite cerca de lo que desencadena el evento. Por tanto, la regla se acopla, implícitamente, en el código, y tendrá que mirar el código para, con suerte, descubrir que la regla se implementa allí.

Por otro lado, el uso de los eventos de dominio hace el concepto explícito, porque hay un `DomainEvent` y al menos un `DomainEventHandler` implicados.

Por ejemplo, en la aplicación eShopOnContainers, cuando se crea un pedido, el usuario se convierte en un comprador, por tanto se genera un `OrderStartedDomainEvent` y se controla en el `ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler`, por lo que el concepto subyacente es evidente.

En resumen, los eventos de dominio le ayudan a expresar explícitamente las reglas de dominio, en función del lenguaje ubicuo proporcionado por los expertos de dominio. Además, los eventos de dominio permiten una mejor separación de cuestiones entre clases dentro del mismo dominio.

Es importante asegurarse de que, al igual que en una transacción de base de datos, o todas las operaciones relacionadas con un evento de dominio finalizan correctamente o ninguna lo hace.

Los eventos de dominio son similares a los eventos de estilo de mensajería, con una diferencia importante. Con la mensajería real, las colas de mensajes, los agentes de mensajes o un bus de servicio con AMQP, un mensaje siempre se envía de forma asincrónica y se comunica entre procesos y equipos. Esto es útil para integrar varios contextos delimitados, microservicios o incluso otras aplicaciones. Pero con los eventos de dominio le interesa generar un evento desde la operación de dominio que se está ejecutando actualmente, pero que los efectos secundarios se produzcan dentro del mismo dominio.

Los eventos de dominio y sus efectos secundarios (las acciones iniciadas después que se administren mediante controladores de eventos) se deben producir casi de inmediato, por lo general en el proceso y dentro del mismo dominio. Por tanto, los eventos de dominio pueden ser sincrónicos o asincrónicos. Pero los eventos de integración siempre deben ser asincrónicos.

Eventos de dominio frente a eventos de integración

Semánticamente, los eventos de dominio y los de integración son lo mismo: notificaciones sobre algo que acaba de ocurrir. Pero su implementación debe ser

diferente. Los eventos de dominio son simplemente mensajes insertados en un distribuidor de eventos de dominio, que se puede implementar como un mediador en memoria basado en un contenedor de IoC o cualquier otro método.

Por otro lado, el propósito de los eventos de integración es propagar las transacciones confirmadas y actualizaciones a subsistemas adicionales, con independencia de que sean otros microservicios, contextos delimitados o incluso aplicaciones externas. Por tanto, solo se deben producir si la entidad se conserva correctamente, de otra forma será como si toda la operación nunca se hubiera producido.

Como se ha mencionado anteriormente, los eventos de integración se deben basar en la comunicación asincrónica entre varios microservicios (otros contextos delimitados) o incluso sistemas o aplicaciones externos.

Por tanto, la interfaz de bus de eventos necesita una infraestructura que permita la comunicación entre procesos y distribuida entre servicios potencialmente remotos. Se pueden basar en un bus de servicio comercial, colas, una base de datos compartida que se use como un buzón o cualquier otro sistema de mensajería distribuido e, idealmente, basado en inserciones.

Eventos de dominio como método preferido para desencadenar efectos secundarios entre varios agregados dentro del mismo dominio

Si la ejecución de un comando relacionado con una instancia de agregado requiere reglas de dominio adicionales para ejecutarse en uno o varios agregados adicionales, debe diseñar e implementar esos efectos secundarios para que se desencadenen mediante eventos de dominio. Como se muestra en la figura 7-14 y como uno de los casos de uso más importantes, se debe usar un evento de dominio para propagar los cambios de estado entre varios agregados dentro del mismo modelo de dominio.

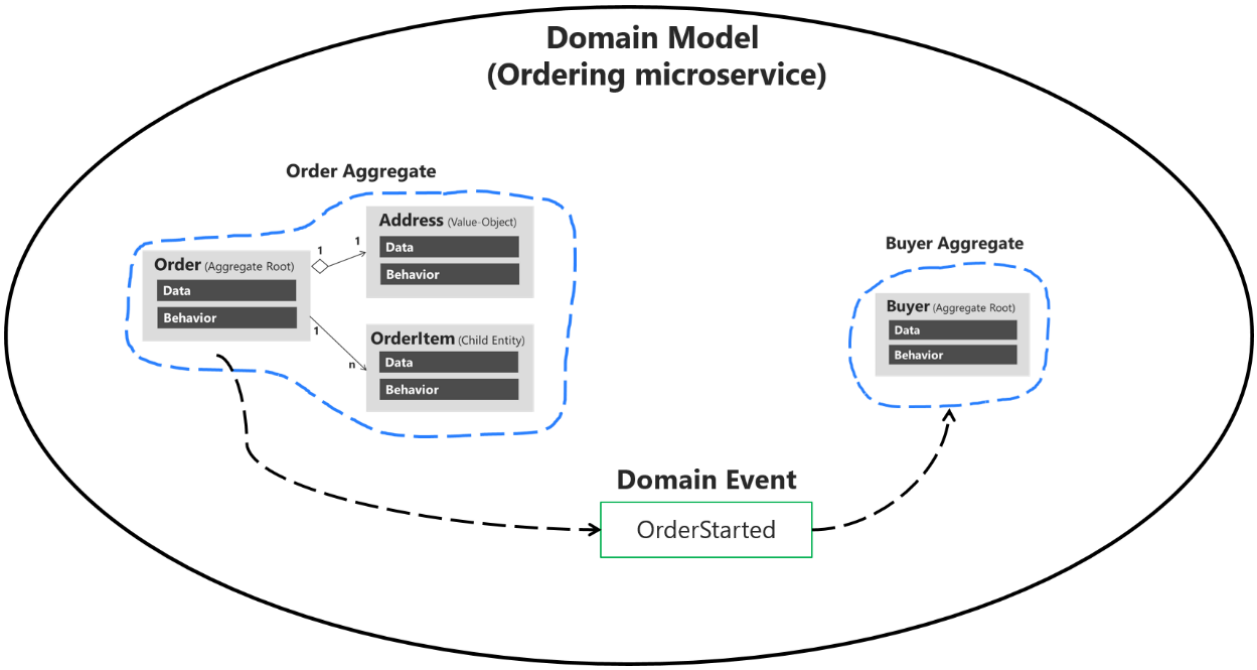


Figura 7-14. Eventos de dominio para exigir la coherencia entre varios agregados dentro del mismo dominio

En la figura 7-14 se muestra cómo consiguen los eventos de dominio la coherencia entre los agregados. Cuando el usuario inicia un pedido, el agregado Order envía un evento de dominio `OrderStarted`. El agregado Buyer controla el evento de dominio OrderStarted para crear un objeto Buyer en el microservicio de pedidos, según la información de usuario original del microservicio de identidades (con la información proporcionada en el comando CreateOrder).

Como alternativa, puede hacer que la raíz agregada se suscriba a los eventos generados por los miembros de sus agregados (las entidades secundarias). Por ejemplo, cada entidad secundaria OrderItem puede generar un evento cuando el precio del artículo sea mayor que una cantidad específica, o bien cuando la cantidad del elemento de producto sea demasiado alta. Después, la raíz agregada puede recibir esos eventos y realizar un cálculo o una agregación global.

Es importante comprender que este tipo de comunicación basada en eventos no se implementa de forma directa dentro de los agregados; tendrá que implementar controladores de eventos de dominio.

El control de los eventos de dominio es una cuestión de la aplicación. El nivel de modelo de dominio solo debe centrarse en la lógica del dominio, en lo que un experto de dominio debería entender, no en la infraestructura de la aplicación como controladores y acciones de persistencia de efectos secundarios mediante repositorios. Por tanto, el nivel de aplicación es donde los controladores de eventos de dominio deberían desencadenar acciones cuando se produzca un evento de dominio.

Los eventos de dominio también se pueden usar para desencadenar cualquier número de acciones de la aplicación y, lo que es más importante, deben ser abiertos para aumentar ese número en el futuro de forma desacoplada. Por ejemplo, al iniciar el pedido, es posible que le interese publicar un evento de dominio para propagar esa información a otros agregados o incluso para generar acciones de la aplicación como notificaciones.

El punto clave es el número abierto de acciones que se van a ejecutar cuando se produce un evento de dominio. Con el tiempo, las acciones y reglas en el dominio y la aplicación aumentarán. La complejidad o el número de acciones de efectos secundarios aumentará cuando ocurra algo, pero si el código se acoplara con "adherencia" (es decir, la creación de objetos específicos con `new`), cada vez que necesitara agregar una acción nueva también tendría que cambiar el código funcional y probado.

Este cambio podría provocar nuevos errores y este enfoque también va en contra del principio abierto/cerrado de SOLID. No solo eso, la clase original que orquestaba las operaciones no dejaría de crecer, algo contrario al Principio de responsabilidad única (SRP).

Por otro lado, si usa eventos de dominio, puede crear una implementación específica y desacoplada mediante la separación de las responsabilidades de esta manera:

1. Envíe un comando (por ejemplo, CreateOrder).
2. Reciba el comando en un controlador de comandos.

    - Ejecute la transacción de un solo agregado.
    - (Opcional) Genere eventos de dominio para los efectos secundarios (por ejemplo, OrderStartedDomainEvent).

3. Controle los eventos de dominio (en el proceso actual) que van a ejecutar un número abierto de efectos secundarios en varios agregados o acciones de la aplicación. Por ejemplo:

    - Compruebe o cree el comprador y el método de pago.
    - Cree y envíe un evento de integración relacionado al bus de eventos para propagar los estados entre los microservicios o desencadenar acciones externas como enviar un correo electrónico al comprador.
    - Controle otros efectos secundarios.

Como se muestra en la figura 7-15, empezando desde el mismo evento de dominio, puede controlar varias acciones relacionadas con otros agregados en el dominio o acciones de la aplicación adicionales que tenga que realizar entre los microservicios conectados con eventos de integración y el bus de eventos.

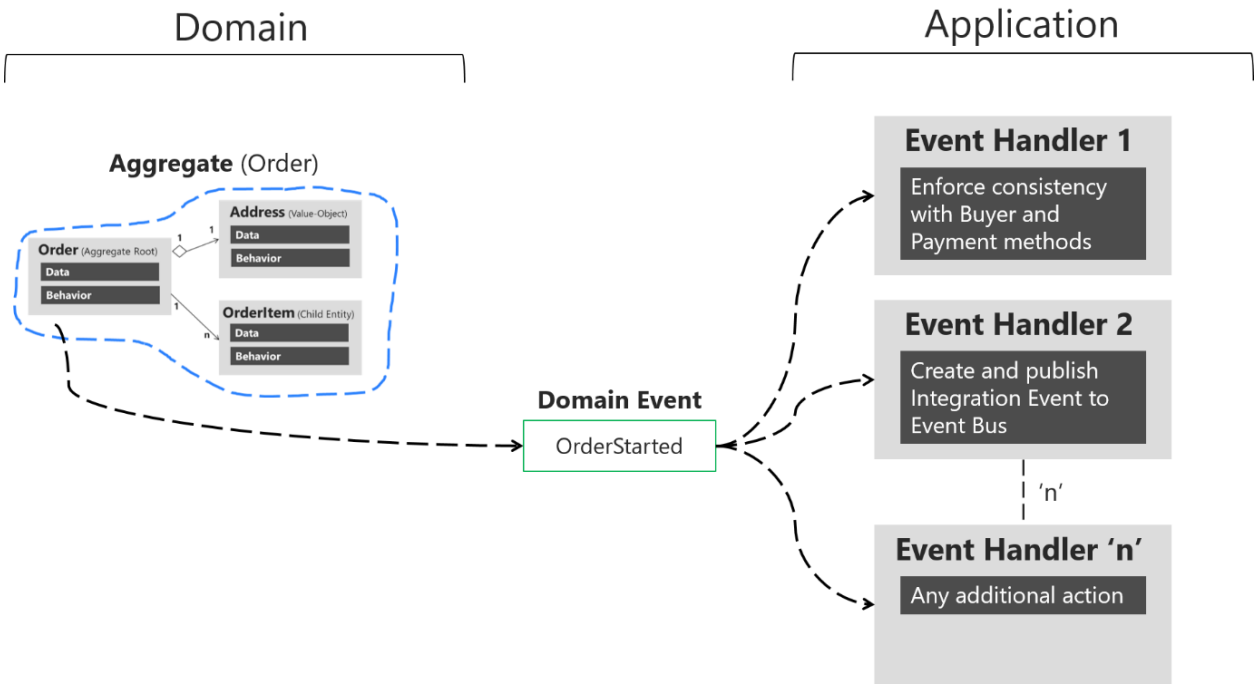


Figura 7-15. Control de varias acciones por dominio

Puede haber varios controladores para el mismo evento de dominio en el nivel de aplicación, un controlador puede resolver la coherencia entre agregados y otro controlador puede publicar un evento de integración, por lo que otros microservicios pueden hacer algo con él. Normalmente los controladores de eventos se encuentran en la capa de aplicación, porque los objetos de infraestructura como los repositorios o una API de aplicación se usarán para el comportamiento del microservicio. En ese sentido, los controladores de eventos son similares a los controladores de comandos, por lo que ambos forman parte del nivel de aplicación. La diferencia más importante es que un comando solo se debe procesar una vez. Un evento de dominio se podría procesar cero o *n* veces, porque lo pueden recibir varios receptores o controladores de eventos con un propósito diferente para cada controlador.

Tener un número abierto de controladores por evento de dominio permite agregar tantas reglas de dominio como sea necesario sin que el código actual se vea afectado. Por ejemplo, la implementación de la siguiente regla de negocio podría ser tan fácil como agregar algunos controladores de eventos (o incluso solo uno):

> Cuando la cantidad total adquirida por un cliente en el almacén, en cualquier número de pedidos, supera los 6000 dólares, se aplica un 10 % de descuento a cada pedido nuevo y se notifica ese descuento para futuros pedidos a los clientes con un correo electrónico.

Implementación de eventos de dominio

En C#, un evento de dominio es simplemente una estructura o clase que almacena datos, como un DTO, con toda la información relacionada con lo que ha sucedido en el dominio, como se muestra en el ejemplo siguiente:

C#

```
public class OrderStartedDomainEvent : INotification
{
    public string UserId { get; }
    public string UserName { get; }
    public int CardTypeId { get; }
    public string CardNumber { get; }
    public string CardSecurityNumber { get; }
    public string CardHolderName { get; }
    public DateTime CardExpiration { get; }
    public Order Order { get; }

    public OrderStartedDomainEvent(Order order, string userId, string
userName,
                                   int cardTypeId, string cardNumber,
                                   string cardSecurityNumber, string
cardHolderName,
                                   DateTime cardExpiration)
    {
        Order = order;
        UserId = userId;
        UserName = userName;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}
```

Esto es básicamente una clase que contiene todos los datos relacionados con el evento OrderStarted.

En cuanto al lenguaje ubicuo del dominio, como un evento es algo que tuvo lugar en el pasado, el nombre de clase del evento se debe representar como un verbo en pasado, como OrderStartedDomainEvent u OrderShippedDomainEvent. Esta es la forma de implementar el evento de dominio en el microservicio de pedidos en eShopOnContainers.

Como se indicó anteriormente, una característica importante de los eventos es que, como un evento es algo que se produjo en el pasado, no debería cambiar. Por tanto, debe ser una clase inmutable. En el código anterior se puede ver que las propiedades son de solo lectura. No hay ninguna manera de actualizar el objeto, solo se pueden establecer valores al crearlo.

Es importante destacar aquí que si los eventos de dominio tuvieran que administrarse de forma asincrónica, mediante una cola que necesitase serializar y deserializar los objetos de eventos, las propiedades tendrían que ser "conjunto privado" en lugar de solo lectura, por lo que el deserializador podría asignar los valores tras quitar de la cola. Esto no es un problema en el microservicio Ordering, ya que el evento de dominio pub/sub se implementa sincrónicamente con MediatR.

Generación de eventos de dominio

La siguiente pregunta es cómo generar un evento de dominio para que llegue a sus controladores de eventos relacionados. Se pueden usar varios enfoques.

Originalmente, Udi Dahan propuso el uso de una clase estática para administrar y generar los eventos (por ejemplo, en algunas publicaciones relacionadas, como Domain Events – Take 2 [Eventos de dominio: Toma 2]). Esto podría incluir una clase estática denominada DomainEvents que generaría los eventos de dominio inmediatamente cuando es llamada, con una sintaxis similar a `DomainEvents.Raise(Event myEvent)`. Jimmy Bogard escribió una entrada de blog [Strengthening your domain: Domain Events (Reforzar el dominio: eventos de dominio)] que recomienda un enfoque similar.

Pero cuando la clase de eventos de dominio es estática, también lo envía a los controladores inmediatamente. Esto dificulta las pruebas y la depuración, dado que los controladores de eventos con la lógica de efectos secundarios se ejecutan inmediatamente después de que se genera el evento. Durante las pruebas y la depuración, únicamente le interesa centrarse en lo que sucede en las clases agregadas actuales; no quiere que de repente se le redirija a otros controladores de eventos por efectos secundarios relacionados con otros agregados o la lógica de aplicación. Es el motivo de que otros métodos hayan evolucionado, como se explica en la sección siguiente.

El enfoque diferido para generar y enviar eventos

En lugar de enviar a un controlador de eventos de dominio de forma inmediata, un enfoque más adecuado consiste en agregar los eventos de dominio a una colección y, después, enviarlos *justo antes* o *justodespués* de confirmar la transacción (como ocurre con SaveChanges en EF). (Este enfoque lo describió Jimmy Bogard en esta publicación A better domain events pattern [Un patrón de eventos de dominio mejor]).

Decidir si enviar los eventos de dominio justo antes o justo después de confirmar la transacción es importante, ya que determina si se van a incluir los efectos secundarios como parte de la misma transacción o en transacciones diferentes. En este último caso,

debe controlar la coherencia final entre varios agregados. Este tema se analiza en la sección siguiente.

El enfoque diferido es el que se usa en eShopOnContainers. En primer lugar, se agregan los eventos que tienen lugar en las entidades a una colección o lista de eventos por entidad. Esa lista debe formar parte del objeto de entidad, o incluso mejor, de la clase de entidad base, como se muestra en el ejemplo siguiente de la clase base Entity:

C#

```
public abstract class Entity
{
     //...
     private List<INotification> _domainEvents;
     public List<INotification> DomainEvents => _domainEvents;

     public void AddDomainEvent(INotification eventItem)
     {
         _domainEvents = _domainEvents ?? new List<INotification>();
         _domainEvents.Add(eventItem);
     }

     public void RemoveDomainEvent(INotification eventItem)
     {
         _domainEvents?.Remove(eventItem);
     }
     //... Additional code
}
```

Cuando quiera generar un evento, simplemente agréguelo a la colección de eventos desde el código en cualquier método de la entidad raíz agregada.

En el código siguiente, parte de la raíz agregada Order de eShopOnContainers, se muestra un ejemplo:

C#

```
var orderStartedDomainEvent = new OrderStartedDomainEvent(this, //Order
object
                                                          cardTypeId,
cardNumber,

cardSecurityNumber,
                                                          cardHolderName,
                                                          cardExpiration);
this.AddDomainEvent(orderStartedDomainEvent);
```

Tenga en cuenta que lo único que hace el método AddDomainEvent es agregar un evento a la lista. Todavía no se distribuye ningún evento, ni tampoco se invoca ningún controlador de eventos.

Lo que realmente le interesa es enviar los eventos después, cuando la transacción se confirme en la base de datos. Si usa Entity Framework Core, eso significa hacerlo en el método SaveChanges del DbContext de EF, como en el código siguiente:

C#

```
// EF Core DbContext
public class OrderingContext : DbContext, IUnitOfWork
{
    // ...
    public async Task<bool> SaveEntitiesAsync(CancellationToken
cancellationToken = default(CancellationToken))
    {
        // Dispatch Domain Events collection.
        // Choices:
        // A) Right BEFORE committing data (EF SaveChanges) into the DB.
This makes
        // a single transaction including side effects from the domain event
        // handlers that are using the same DbContext with Scope lifetime
        // B) Right AFTER committing data (EF SaveChanges) into the DB. This
makes
        // multiple transactions. You will need to handle eventual
consistency and
        // compensatory actions in case of failures.
        await _mediator.DispatchDomainEventsAsync(this);

        // After this line runs, all the changes (from the Command Handler
and Domain
        // event handlers) performed through the DbContext will be committed
        var result = await base.SaveChangesAsync();
    }
}
```

Con este código, los eventos de entidad se envían a sus controladores de eventos correspondientes.

El resultado general es que se separa la generación de un evento de dominio (una sencilla adición a una lista en memoria) de su envío a un controlador de eventos. Además, en función del tipo de distribuidor que se use, los eventos se podrían enviar de forma sincrónica o asincrónica.

Tenga en cuenta que aquí los límites transaccionales tienen una importancia especial. Si la unidad de trabajo y la transacción pueden abarcar más de un agregado (como ocurre cuando se usa EF Core y una base de datos relacional), esto puede funcionar bien. Pero

si la transacción no puede abarcar agregados debe implementar pasos adicionales para lograr la coherencia. Este es otro motivo por el que la omisión de persistencia no es universal; depende del sistema de almacenamiento que se use.

Transacción única entre agregados frente a coherencia final entre agregados

La pregunta de si se debe realizar una única transacción entre agregados en lugar de depender de la coherencia final entre esos agregados es controvertida. Muchos autores de DDD, como Eric Evans y Vaughn Vernon, promueven la regla "una transacción = un agregado" y argumentan, por tanto, la coherencia final entre agregados. Por ejemplo, en su libro *Domain-Driven Design* (Diseño controlado por eventos), Eric Evans afirma lo siguiente:

> No se espera que las reglas que abarcan agregados estén actualizadas en todo momento. A través del procesamiento de eventos, el procesamiento por lotes u otros mecanismos de actualización, se pueden resolver otras dependencias dentro de un periodo determinado. (página 128)

Vaughn Vernon afirma lo siguiente en Effective Aggregate Design. Part II: Making Aggregates Work Together (Diseño eficaz de agregados - Parte II: hacer que los agregados funcionen de forma conjunta):

> Por tanto, si la ejecución de un comando en una instancia del agregado requiere que se ejecuten reglas de negocio adicionales en uno o varios agregados, use la coherencia final [...] Hay una forma práctica de admitir la coherencia final en un modelo de DDD. Un método de agregado publica un evento de dominio que con el tiempo se entrega a uno o varios suscriptores asincrónicos.

Esta lógica se basa en la adopción de transacciones específicas en lugar de transacciones distribuidas entre varios agregados o entidades. La idea es que, en el segundo caso, el número de bloqueos de base de datos será relevante en aplicaciones a gran escala con necesidades de alta escalabilidad. Asumir el hecho de que las aplicaciones de alta escalabilidad no necesitan coherencia transaccional entre varios agregados ayudará a aceptar el concepto de la coherencia final. A menudo los cambios atómicos no son necesarios por parte de la empresa y, en cualquier caso, es la responsabilidad de los expertos de dominio indicar si determinadas operaciones necesitan transacciones atómicas o no. Si una operación siempre necesita una transacción atómica entre varios agregados, podría preguntarse si el agregado debería ser mayor o si no se diseñó correctamente.

Pero otros desarrolladores y arquitectos como Jimmy Bogard se conforman con una sola transacción que abarque varios agregados, pero solo cuando esos agregados adicionales estén relacionados con efectos secundarios para el mismo comando original. Por ejemplo, en A better domain events pattern, Bogard afirma lo siguiente:

> Normalmente, es interesante que los efectos secundarios de un evento de dominio se produzcan en la misma transacción lógica, pero no necesariamente en el mismo ámbito de generación del evento de dominio [...] Justo antes de que se confirme la transacción, los eventos se envían a sus correspondientes controladores.

Si los eventos de dominio se envían justo *antes* de confirmar la transacción original, es porque interesa que los efectos secundarios de esos eventos se incluyan en la misma transacción. Por ejemplo, si se produce un error en el método SaveChanges de DbContext de EF, la transacción revertirá todos los cambios, incluido el resultado de cualquier operación de efecto secundario implementada por los controladores de eventos de dominio relacionados. Esto se debe a que el ámbito de duración de DbContext se define de forma predeterminada como "con ámbito". Por lo tanto, el objeto DbContext se comparte entre varios objetos de repositorio que se crean instancias en el mismo ámbito o gráfico de objetos. Esto coincide con el ámbito de HttpRequest al desarrollar aplicaciones de API web o MVC.

En realidad, ambos enfoques (única transacción atómica y coherencia final) pueden ser correctos. Realmente depende de los requisitos empresariales o de dominio, y de lo que los expertos de dominio digan. También depende de la capacidad de escalabilidad que deba tener el servicio (las transacciones más granulares tienen un impacto menor en relación con los bloqueos de base de datos). Y depende de la inversión que esté dispuesto a realizar en el código, puesto que la coherencia final requiere un código más complejo con el fin de detectar posibles incoherencias entre los agregados y la necesidad de implementar acciones de compensación. Tenga en cuenta que si confirma los cambios en el agregado original y después, cuando los eventos se estén distribuyendo, si se produce un problema y los controladores de eventos no pueden confirmar sus efectos secundarios, tendrá incoherencias entre los agregados.

Una manera de permitir acciones de compensación sería almacenar los eventos de dominio en tablas de base de datos adicionales para que puedan formar parte de la transacción original. Después, podría tener un proceso por lotes que detectara las incoherencias y ejecutara acciones de compensación comparando la lista de eventos con el estado actual de los agregados. Las acciones de compensación forman parte de un tema complejo que requerirá un análisis profundo por su parte, incluido su análisis con los usuarios empresariales y expertos de dominio.

En cualquier caso, puede elegir el enfoque que necesite. Pero el enfoque diferido inicial (generar los eventos antes de la confirmación y usar una sola transacción) es el más sencillo cuando se usa EF Core y una base de datos relacional. Es más fácil de implementar y resulta válido en muchos casos de negocio. También es el enfoque que se usa en el microservicio de pedidos de eShopOnContainers.

¿Pero cómo se envían realmente los eventos a sus correspondientes controladores de eventos? ¿Qué es el objeto `_mediator` que ve en el ejemplo anterior? Tiene que ver con las técnicas y los artefactos que se usan para la asignación entre eventos y sus controladores de eventos.

El distribuidor de eventos de dominio: asignación de eventos a controladores de eventos

Una vez que se puedan enviar o publicar los eventos, se necesita algún tipo de artefacto que publique el evento, para que todos los controladores relacionados puedan obtenerlo y procesar efectos secundarios en función de ese evento.

Un enfoque es un sistema de mensajería real o incluso un bus de eventos, posiblemente basado en un bus de servicio en lugar de en eventos en memoria. Pero para el primer caso, la mensajería real sería excesiva para el procesamiento de eventos de dominio, ya que solo es necesario procesar los eventos dentro del mismo proceso (es decir, dentro del mismo nivel de dominio y aplicación).

Cómo suscribirse a eventos de dominio

Cuando se usa MediatR, todos los controladores de eventos deben usar un tipo de evento que se proporciona en el parámetro genérico de la interfaz `INotificationHandler`, como se puede ver en el código siguiente:

C#

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
  : INotificationHandler<OrderStartedDomainEvent>
```

En función de la relación entre el evento y el controlador de eventos, que se puede considerar la suscripción, el artefacto de MediatR puede detectar todos los controladores de eventos para cada evento y desencadenar cada uno de ellos.

Cómo controlar eventos de dominio

Por último, el controlador de eventos normalmente implementa código de nivel de aplicación en el que se usan repositorios de infraestructura para obtener los agregados adicionales necesarios y para ejecutar la lógica del dominio de efectos secundarios. En el siguiente código de controlador de eventos de dominio de eShopOnContainers, se muestra un ejemplo de implementación.

C#

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : INotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILogger _logger;
    private readonly IBuyerRepository _buyerRepository;
    private readonly IOrderingIntegrationEventService
_orderingIntegrationEventService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(

ILogger<ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler>
logger,
        IBuyerRepository buyerRepository,
        IOrderingIntegrationEventService orderingIntegrationEventService)
    {
        _buyerRepository = buyerRepository ?? throw new
ArgumentNullException(nameof(buyerRepository));
        _orderingIntegrationEventService = orderingIntegrationEventService
?? throw new ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task Handle(
        OrderStartedDomainEvent domainEvent, CancellationToken
cancellationToken)
    {
        var cardTypeId = domainEvent.CardTypeId != 0 ?
domainEvent.CardTypeId : 1;
        var buyer = await _buyerRepository.FindAsync(domainEvent.UserId);
        var buyerExisted = buyer is not null;

        if (!buyerExisted)
        {
            buyer = new Buyer(domainEvent.UserId, domainEvent.UserName);
        }

        buyer.VerifyOrAddPaymentMethod(
            cardTypeId,
            $"Payment Method on {DateTime.UtcNow}",
            domainEvent.CardNumber,
            domainEvent.CardSecurityNumber,
            domainEvent.CardHolderName,
            domainEvent.CardExpiration,
            domainEvent.Order.Id);
```

```
        var buyerUpdated = buyerExisted ?
            _buyerRepository.Update(buyer) :
            _buyerRepository.Add(buyer);

        await _buyerRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);

        var integrationEvent = new
OrderStatusChangedToSubmittedIntegrationEvent(
            domainEvent.Order.Id, domainEvent.Order.OrderStatus.Name,
buyer.Name);
        await
_orderingIntegrationEventService.AddAndSaveEventAsync(integrationEvent);

        OrderingApiTrace.LogOrderBuyerAndPaymentValidatedOrUpdated(
            _logger, buyerUpdated.Id, domainEvent.Order.Id);
    }
}
```

El código anterior de controlador de eventos de dominio se considera código de nivel de aplicación porque usa repositorios de infraestructura, como se explica en la sección siguiente sobre el nivel de persistencia de infraestructura. Los controladores de eventos también pueden usar otros componentes de infraestructura.

Los eventos de dominio pueden generar eventos de integración para publicarse fuera de los límites del microservicio

Por último, es importante mencionar que en ocasiones es posible que le interese propagar los eventos a través de varios microservicios. Dicha propagación es un evento de integración y se podría publicar a través de un bus de eventos desde cualquier controlador de eventos de dominio específico.

Conclusiones sobre los eventos de dominio

Como se mencionó, los eventos de dominio se usan para implementar explícitamente los efectos secundarios de los cambios en el dominio. Para usar la terminología de DDD, los eventos de dominio se usan para implementar explícitamente los efectos secundarios a través de uno o varios agregados. Además, para una mejor escalabilidad y un menor impacto en los bloqueos de base de datos, la coherencia final se usa entre agregados dentro del mismo dominio.

La aplicación de referencia usa MediatR para propagar los eventos de dominio sincrónicamente entre agregados, dentro de una única transacción. No obstante, también puede usar una implementación de AMQP como RabbitMQ o Azure Service Bus para propagar los eventos de dominio de forma asincrónica con la coherencia

eventual. Pero, como se mencionó anteriormente, hay que tener en cuenta la necesidad de acciones compensatorias en caso de que se produzcan errores.

Recursos adicionales

- **Greg Young. ¿Qué es un evento de dominio?**
  https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf#page=25
- **Jan Stenberg. Domain Events and Eventual Consistency (Eventos de dominio y coherencia definitiva)**
  https://www.infoq.com/news/2015/09/domain-events-consistency
- **Jimmy Bogard. A better domain events pattern (Un mejor patrón de eventos de dominio)**
  https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/
- **Vaughn Vernon. Effective Aggregate Design - Part II: Making Aggregates Work Together (Diseño de agregados efectivo, parte II: Conseguir que los agregados funcionen juntos)**
  https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Jimmy Bogard. Strengthening your domain: Domain Events (Reforzar el dominio: eventos de dominio)**
  https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/
- **Udi Dahan. How to create fully encapsulated Domain Models (Cómo crear modelos de dominio totalmente encapsulados)**
  https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/
- **Udi Dahan. Domain Events – Take 2 (Eventos de dominio: parte 2)**
  https://udidahan.com/2008/08/25/domain-events-take-2/
- **Udi Dahan. Domain Events – Salvation (Eventos de dominio: salvación)**
  https://udidahan.com/2009/06/14/domain-events-salvation/
- **Cesar de la Torre. Eventos de dominio frente a eventos de integración en arquitecturas de DDD y microservicios**
  https://devblogs.microsoft.com/cesardelatorre/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/

Diseño de la capa de persistencia de infraestructura

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Los componentes de persistencia de datos proporcionan acceso a los datos hospedados dentro de los límites de un microservicio (es decir, la base de datos de un microservicio). Contienen la implementación real de componentes como repositorios y clases unitarias de trabajo , como objetos de Entity Framework (EF) DbContext personalizados. DbContext de EF implementa los patrones de repositorio y de unidad de trabajo.

El patrón de repositorio

El patrón Repositorio es un patrón de diseño Domain-Driven diseñado para mantener los problemas de persistencia fuera del modelo de dominio del sistema. Una o varias abstracciones de persistencia (interfaces) se definen en el modelo de dominio y estas abstracciones tienen implementaciones en forma de adaptadores específicos de persistencia definidos en otra parte de la aplicación.

Las implementaciones del repositorio son clases que encapsulan la lógica necesaria para acceder a los orígenes de datos. Centralizan la funcionalidad de acceso a datos comunes, lo que proporciona una mejor capacidad de mantenimiento y desacoplan la infraestructura o la tecnología usadas para acceder a las bases de datos desde el modelo de dominio. Si se usa un

asignador relacional de objetos (ORM) como Entity Framework, se simplifica el código que se debe implementar, gracias a LINQ y al establecimiento inflexible de tipos. Esto le permite centrarse en la lógica de persistencia de datos en lugar de en la fontanería de acceso a datos.

El patrón Repository es una forma bien documentada de trabajar con un origen de datos. En el libro Patrones de arquitectura de aplicaciones empresariales, Martin Fowler describe un repositorio de la siguiente manera:

> Un repositorio realiza las tareas de un intermediario entre las capas del modelo de dominio y la asignación de datos, actuando de forma similar a un conjunto de objetos de dominio en memoria. Los objetos cliente crean consultas mediante declaración y las envían a los repositorios para obtener respuestas. Conceptualmente, un repositorio encapsula un conjunto de objetos almacenados en la base de datos y las operaciones que se pueden realizar en ellos, lo que proporciona una manera más cercana a la capa de persistencia. Los repositorios, además, cumplen el propósito de separar, de manera clara y en una sola dirección, la dependencia entre el ámbito de trabajo y la asignación o mapeo de datos.

Definir un repositorio por agregado

Para cada agregado o raíz agregada, se debe crear una clase de repositorio. Puede aprovechar los genéricos de C# para reducir el número total de clases concretas que necesita mantener (como se muestra más adelante en este capítulo). En un microservicio basado en patrones de Domain-Driven Design (DDD), el único canal que debe utilizar para actualizar la base de datos son los repositorios. Esto se debe a que tienen una relación uno a uno con la raíz de agregado, que controla las invariables del agregado y la coherencia transaccional. Está bien consultar la base de datos a través de otros canales (como puede hacer siguiendo un enfoque de CQRS), ya que las consultas no cambian el estado de la base de datos. Sin embargo, el área transaccional (es decir, las actualizaciones) siempre debe controlarse mediante los repositorios y las raíces agregadas.

Básicamente, un repositorio permite rellenar los datos en la memoria que proceden de la base de datos en forma de entidades de dominio. Una vez que las entidades están en memoria, se pueden cambiar y, a continuación, conservarse de nuevo en la base de datos a través de transacciones.

Como se indicó anteriormente, si usa el patrón de arquitectura CQS/CQRS, las consultas iniciales se realizan mediante consultas en paralelo fuera del modelo de dominio, mediante instrucciones SQL simples utilizando Dapper. Este enfoque es mucho más flexible que los repositorios, ya que puede consultar y combinar las tablas que necesite, y estas consultas no están restringidas por reglas de los agregados. Esos datos van a la capa de presentación o a la aplicación cliente.

Si el usuario realiza cambios, los datos que se van a actualizar proceden de la aplicación cliente o la capa de presentación a la capa de aplicación (por ejemplo, un servicio de API web). Cuando recibe un comando en un controlador de comandos, se usan repositorios para obtener los datos que desea actualizar desde la base de datos. Actualizas los datos en memoria mediante los comandos proporcionados y luego agregas o actualizas las entidades de dominio en la base de datos a través de una transacción.

Es importante resaltar de nuevo que solo debe definir un repositorio para cada raíz de agregado, como se muestra en la figura 7-17. Para lograr el objetivo del elemento raíz del agregado con el fin de mantener la coherencia transaccional entre todos los objetos dentro del agregado, nunca debe crear un repositorio para cada tabla de la base de datos.

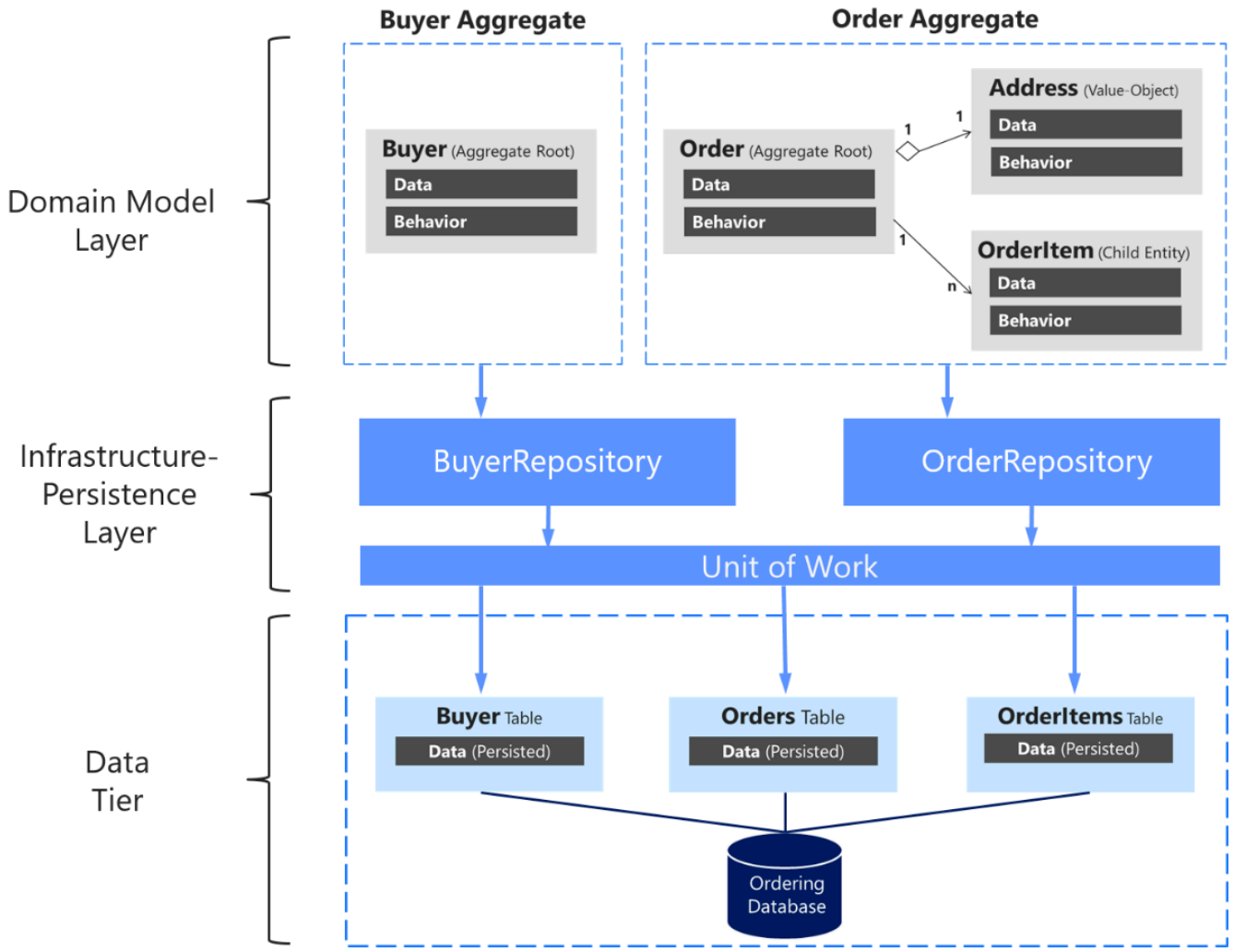


Figura 7-17. Relación entre repositorios, agregados y tablas de base de datos

En el diagrama anterior se muestran las relaciones entre las capas de Dominio e Infraestructura: el agregado de compradores depende de la interfaz IBuyerRepository y el agregado de pedidos depende de las interfaces IOrderRepository. Estas interfaces se implementan en la capa de Infraestructura por los repositorios correspondientes que dependen de UnitOfWork, también implementados allí, que acceden a las tablas del nivel de Datos.

Aplicación de una raíz agregada por repositorio

Puede ser útil implementar el diseño del repositorio de tal manera que aplique la regla que solo las raíces agregadas deben tener repositorios. Puede crear un tipo de repositorio genérico o base que restrinja el tipo de entidades con las que funciona para asegurarse de que tienen la interfaz de `IAggregateRoot` marcador.

Por lo tanto, cada clase de repositorio implementada en el nivel de infraestructura implementa su propio contrato o interfaz, como se muestra en el código siguiente:

C#

```
namespace
Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
      // ...
    }
}
```

Cada interfaz de repositorio específica implementa la interfaz IRepository genérica:

C#

```
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);
    // ...
}
```

Sin embargo, una mejor manera de hacer que el código aplique la convención de que cada repositorio está relacionado con un único agregado es implementar un tipo de repositorio genérico. De este modo, es explícito que usa un repositorio para tener como destino un agregado específico. Esto se puede hacer fácilmente mediante la implementación de una interfaz base genérica `IRepository` , como en el código siguiente:

C#

```
public interface IRepository<T> where T : IAggregateRoot
{
    //....
}
```

El patrón Repository facilita la prueba de la lógica de la aplicación.

El patrón Repository permite probar fácilmente la aplicación con pruebas unitarias. Recuerde que las pruebas unitarias solo prueban el código, no la infraestructura, por lo que las abstracciones del repositorio facilitan la consecución de ese objetivo.

Como se indicó en una sección anterior, se recomienda definir y colocar las interfaces del repositorio en el nivel de modelo de dominio para que la capa de aplicación, como el microservicio de api web, no dependa directamente del nivel de infraestructura donde haya implementado las clases de repositorio reales. Al hacerlo y usar la inserción de dependencias en los controladores de la API web, puede implementar repositorios ficticios que devuelvan datos falsos en lugar de datos de la base de datos. Este enfoque desacoplado permite crear y ejecutar pruebas unitarias que centran la lógica de la aplicación sin necesidad de conectividad con la base de datos.

Las conexiones a bases de datos pueden producir errores y, lo que es más importante, ejecutar cientos de pruebas en una base de datos es incorrecta por dos motivos. En primer lugar, puede tardar mucho tiempo debido al gran número de pruebas. En segundo lugar, los registros de base de datos podrían cambiar y afectar a los resultados de las pruebas, especialmente si las pruebas se ejecutan en paralelo, de modo que podrían no ser coherentes. Las pruebas unitarias normalmente se pueden ejecutar en paralelo; Es posible que las pruebas de integración no admitan la ejecución en paralelo en función de su implementación. Las pruebas en la base de datos no son una prueba unitaria, sino una prueba de integración. Debe tener muchas pruebas unitarias que se ejecutan rápidamente, pero menos pruebas de integración en las bases de datos.

En cuanto a la separación de intereses para las pruebas unitarias, la lógica funciona en entidades de dominio en memoria, ya que supone que la clase de repositorio las ha entregado. Una vez que la lógica modifica las entidades de dominio, supone que la clase de repositorio las almacenará correctamente. El punto importante aquí es crear pruebas unitarias contra tu modelo de dominio, así como su lógica de dominio. Las raíces agregadas son los límites de coherencia principales en DDD.

Los repositorios implementados en eShopOnContainers se basan en la implementación dbContext de EF Core de los patrones de repositorio y unidad de trabajo mediante su seguimiento de cambios, por lo que no duplican esta funcionalidad.

La diferencia entre el patrón Repository y el patrón heredado de la clase Data Access (clase DAL)

Un objeto DAL típico realiza directamente operaciones de acceso a datos y persistencia en el almacenamiento, a menudo en el nivel de una sola tabla y fila. Las operaciones CRUD simples implementadas con un conjunto de clases DAL con frecuencia no admiten transacciones (aunque esto no siempre es así). La mayoría de los enfoques de clase DAL hacen un uso mínimo de abstracciones, lo que da lugar a un acoplamiento estricto entre las clases de aplicación o capa lógica de negocios (BLL) que llaman a los objetos DAL.

Al usar el repositorio, los detalles de implementación de la persistencia se encapsulan fuera del modelo de dominio. El uso de una abstracción facilita la extensión del comportamiento a través de patrones como Decoradores o Proxies. Por ejemplo, los problemas transversales, como el almacenamiento en caché, el registro y el control de errores, se pueden aplicar mediante estos patrones en lugar de codificarse de forma rígida en el propio código de acceso a datos. También es sencillo admitir varios adaptadores de repositorio que se pueden usar en diferentes entornos, desde el desarrollo local hasta los entornos de ensayo compartidos y de producción.

Implementación de unidades de trabajo

Una unidad de trabajo hace referencia a una sola transacción que implica varias operaciones de inserción, actualización o eliminación. En términos simples, significa que para una acción de usuario específica, como un registro en un sitio web, todas las operaciones de inserción, actualización y eliminación se controlan en una sola transacción. Esto es más eficaz que manejar varias operaciones de base de datos de una manera más comunicativa.

Estas múltiples operaciones de persistencia se realizan posteriormente en una sola acción cuando el código de la capa de aplicación lo ordena. La decisión sobre la aplicación de los cambios en memoria al almacenamiento de base de datos real se basa normalmente en el patrón unidad de trabajo. En EF, el patrón Unit of Work se implementa mediante DbContext y se ejecuta cuando se realiza una llamada a `SaveChanges`.

En muchos casos, este patrón o forma de aplicar operaciones en el almacenamiento puede aumentar el rendimiento de la aplicación y reducir la posibilidad de incoherencias. También reduce el bloqueo de transacciones en las tablas de base de datos, ya que todas las operaciones previstas se confirman como parte de una transacción. Esto es más eficaz en comparación con la ejecución de muchas operaciones aisladas en la base de datos. Por lo tanto, el ORM seleccionado puede optimizar la ejecución en la base de datos mediante la agrupación de varias acciones de actualización dentro de la misma transacción, en lugar de muchas ejecuciones de transacciones pequeñas y independientes.

El patrón Unit of Work se puede implementar con o sin usar el patrón Repository.

Los repositorios no deben ser obligatorios

Los repositorios personalizados son útiles por los motivos mencionados anteriormente y es el enfoque para el microservicio de ordenación en eShopOnContainers. Sin embargo, no es un patrón esencial implementar en un diseño DDD o incluso en el desarrollo general de .NET.

Por ejemplo, Jimmy Bogard, al proporcionar comentarios directos para esta guía, dijo lo siguiente:

> Esto probablemente será mi comentario más significativo. Realmente no soy un fan de repositorios, principalmente porque ocultan los detalles importantes del mecanismo de persistencia subyacente. Es por eso que también prefiero MediatR para los comandos. Puedo usar toda la potencia de la capa de persistencia e insertar todo ese comportamiento de dominio en mis raíces agregadas. Normalmente no quiero simular mis repositorios: todavía necesito tener esa prueba de integración con lo real. Adoptar CQRS significó que realmente ya no teníamos necesidad de los repositorios.

Los repositorios pueden ser útiles, pero no son críticos para el diseño de DDD de la manera en que el patrón Aggregate y un modelo de dominio enriquecido son. Por lo tanto, use el patrón de repositorio o no, como considere oportuno.

Recursos adicionales

Patrón de repositorio

- **Edward Hieatt y Rob me. Patrón de repositorio.**
  https://martinfowler.com/eaaCatalog/repository.html
- **El patrón de repositorio**
  https://learn.microsoft.com/previous-versions/msp-n-p/ff649690(v=pandp.10)
- **Eric Evans. "Domain-Driven Design: Tackling Complexity in the Heart of Software" (Diseño orientado al dominio: abordar la complejidad en el corazón del software).**
  (Libro; incluye una explicación del patrón repositorio)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/

Patrón de unidad de trabajo

- **Martin Fowler. Patrón de unidad de trabajo.**
  https://martinfowler.com/eaaCatalog/unitOfWork.html

- **Implementación del repositorio y los patrones de unidad de trabajo en una aplicación MVC de ASP.NET**
  https://learn.microsoft.com/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

Siguiente anterior

Implementación del nivel de persistencia de infraestructura con Entity Framework Core

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Cuando se usan bases de datos relacionales como SQL Server, Oracle o PostgreSQL, se recomienda implementar la capa de persistencia basada en Entity Framework (EF). EF es compatible con LINQ y proporciona objetos fuertemente tipados para el modelo, así como una persistencia simplificada en la base de datos.

Entity Framework tiene un historial largo como parte de .NET Framework. Al usar .NET, también debe usar Entity Framework Core, que se ejecuta en Windows o Linux de la misma manera que .NET. EF Core es una reescritura completa de Entity Framework que se implementa con una superficie mucho más pequeña y mejoras importantes en el rendimiento.

Introducción a Entity Framework Core

Entity Framework (EF) Core es una versión ligera, extensible y multiplataforma de la popular tecnología de acceso a datos de Entity Framework. Se introdujo con .NET Core a mediados de 2016.

Como una introducción a EF Core ya está disponible en la documentación de Microsoft, aquí simplemente proporcionamos vínculos a esa información.

Recursos adicionales

- **Entity Framework Core**
  https://learn.microsoft.com/ef/core/
- **Introducción a ASP.NET Core y Entity Framework Core mediante Visual Studio**
  https://learn.microsoft.com/aspnet/core/data/ef-mvc/
- **Clase DbContext**
  https://learn.microsoft.com/dotnet/api/microsoft.entityframeworkcore.dbcontext
- **Comparar EF Core & EF6.x**
  https://learn.microsoft.com/ef/efcore-and-ef6/index

Infraestructura en Entity Framework Core desde una perspectiva de DDD

Desde un punto de vista de DDD, una funcionalidad importante de EF es la capacidad de usar entidades de dominio POCO, también conocidas en la terminología de EF como *entidades de código primero POCO*. Si usa las entidades de dominio POCO, las clases de modelo de dominio ignoran la persistencia, siguiendo los principios de omisión de persistencia y omisión de infraestructura.

Según los patrones de DDD, debe encapsular el comportamiento de dominio y las reglas dentro de la propia clase de entidad, para que pueda controlar las invariantes, validaciones y reglas al acceder a cualquier colección. Por lo tanto, en DDD no se recomienda permitir el acceso público a colecciones de entidades secundarias u objetos de valor. En su lugar, quiere exponer métodos que controlan cómo y cuándo se pueden actualizar los campos y las colecciones de propiedades, y qué comportamiento y acciones deben producirse cuando esto sucede.

Desde la versión 1.1 de EF Core, para satisfacer estos requisitos de DDD, puede tener campos sin formato en las entidades en lugar de propiedades públicas. Si no desea que un campo de entidad sea accesible externamente, solo puede crear el atributo o campo en lugar de una propiedad. También puede utilizar establecedores de propiedades privadas.

De forma parecida, ahora puede tener acceso de solo lectura a las colecciones usando una propiedad pública del tipo `IReadOnlyCollection<T>`, que está respaldada por un miembro de campo privado para la colección (como `List<T>`) en la entidad que se basa en EF para la

persistencia. Las versiones anteriores de Entity Framework requerían propiedades de colección para admitir `ICollection<T>`, lo que significaba que cualquier desarrollador que use la clase de entidad primaria podría agregar o quitar elementos a través de sus colecciones de propiedades. Esa posibilidad sería contra los patrones recomendados en DDD.

Puede usar una colección privada para exponer un objeto `IReadOnlyCollection<T>` de solo lectura, como se muestra en el ejemplo de código siguiente.

C#

```
public class Order : Entity
{
    // Using private fields, allowed since EF Core 1.1
    private DateTime _orderDate;
    // Other fields ...

    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    protected Order() { }

    public Order(int buyerId, int paymentMethodId, Address address)
    {
        // Initializations ...
    }

    public void AddOrderItem(int productId, string productName,
                             decimal unitPrice, decimal discount,
                             string pictureUrl, int units = 1)
    {
        // Validation logic...

        var orderItem = new OrderItem(productId, productName,
                                      unitPrice, discount,
                                      pictureUrl, units);
        _orderItems.Add(orderItem);
    }
}
```

Solo se puede tener acceso de solo lectura a la propiedad `OrderItems` con `IReadOnlyCollection<OrderItem>`. Este tipo es de solo lectura, por lo que está protegido frente a actualizaciones externas normales.

EF Core proporciona una manera de asignar el modelo de dominio a la base de datos física sin "contaminar" el modelo de dominio. Es código POCO puro de .NET, ya que la acción de mapeo se implementa en la capa de persistencia. En esa acción de asignación, debe configurar la asignación de campos a base de datos. En el siguiente ejemplo del método `OnModelCreating` de `OrderingContext` y de la clase `OrderEntityTypeConfiguration`, la llamada a

`SetPropertyAccessMode` indica a EF Core que acceda a la propiedad `OrderItems` a través de su campo.

C#

```
// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   // ...
   modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
   // Other entities' configuration ...
}

// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
        // Other configuration

        var navigation =

orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));

        //EF access the OrderItem collection property through its backing field
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

        // Other configuration
    }
}
```

Cuando se usan campos en lugar de propiedades, la `OrderItem` entidad se conserva como si tuviera una `List<OrderItem>` propiedad . Pero expone un descriptor de acceso único, el método `AddOrderItem`, para agregar nuevos elementos al pedido. Como resultado, el comportamiento y los datos están vinculados y serán coherentes en cualquier código de aplicación que use el modelo de dominio.

Implementación de repositorios personalizados con Entity Framework Core

En el nivel de implementación, un repositorio es simplemente una clase con código de persistencia de datos coordinado por una unidad de trabajo (DBContext en EF Core) al realizar actualizaciones, como se muestra en la siguiente clase:

C#

```
// using directives...
namespace
Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository
    {
        private readonly OrderingContext _context;
        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }

        public BuyerRepository(OrderingContext context)
        {
            _context = context ?? throw new
ArgumentNullException(nameof(context));
        }

        public Buyer Add(Buyer buyer)
        {
            return _context.Buyers.Add(buyer).Entity;
        }

        public async Task<Buyer> FindAsync(string buyerIdentityGuid)
        {
            var buyer = await _context.Buyers
                .Include(b => b.Payments)
                .Where(b => b.FullName == buyerIdentityGuid)
                .SingleOrDefaultAsync();

            return buyer;
        }
    }
}
```

La `IBuyerRepository` interfaz procede de la capa de modelo de dominio como un contrato. Sin embargo, la implementación del repositorio se realiza en el nivel de persistencia e infraestructura.

DbContext de EF pasa mediante el constructor a través de la inserción de dependencias. Se comparte entre varios repositorios dentro del mismo ámbito de solicitud HTTP, gracias a su duración predeterminada (`ServiceLifetime.Scoped`) en el contenedor de IoC (que también se puede establecer explícitamente con `services.AddDbContext<>`).

Métodos para implementar en un repositorio (actualizaciones o transacciones frente a consultas)

Dentro de cada clase de repositorio, debe colocar los métodos de persistencia que actualizan el estado de las entidades contenidas en su agregado relacionado. Recuerde que hay una relación uno a uno entre un agregado y su repositorio relacionado. Tenga en cuenta que un objeto de entidad raíz agregada podría tener entidades secundarias incrustadas dentro de su grafo de EF. Por ejemplo, un comprador puede tener varias formas de pago como entidades secundarias relacionadas.

Dado que el enfoque para el microservicio de ordenación en eShopOnContainers también se basa en CQS/CQRS, la mayoría de las consultas no se implementan en repositorios personalizados. Los desarrolladores tienen la libertad de crear las consultas y combinaciones que necesitan para la capa de presentación sin las restricciones impuestas por agregados, repositorios personalizados por agregado y DDD en general. La mayoría de los repositorios personalizados sugeridos por esta guía tienen varios métodos de actualización o transaccionales, pero solo los métodos de consulta necesarios para actualizar los datos. Por ejemplo, el repositorio BuyerRepository implementa un método FindAsync, ya que la aplicación debe saber si existe un comprador determinado antes de crear un nuevo comprador relacionado con el pedido.

Sin embargo, los métodos de consulta reales para obtener datos para enviar a la capa de presentación o las aplicaciones cliente se implementan, como se mencionó, en las consultas CQRS basadas en consultas flexibles mediante Dapper.

Uso de un repositorio personalizado frente al uso de EF DbContext directamente

La clase DbContext de Entity Framework se basa en los patrones unit of Work y Repository y se puede usar directamente desde el código, como desde un controlador MVC de ASP.NET Core. Los patrones de unidad de trabajo y repositorio dan como resultado el código más sencillo, como en el microservicio de catálogo CRUD en eShopOnContainers. En los casos en los que quiera que sea posible el código más sencillo, es posible que quiera usar directamente la clase DbContext, como hacen muchos desarrolladores.

Sin embargo, la implementación de repositorios personalizados proporciona varias ventajas al implementar microservicios o aplicaciones más complejos. Los patrones de unidad de trabajo y repositorio están diseñados para encapsular la capa de persistencia de infraestructura, por lo que se desacopla de las capas de aplicación y modelo de dominio. La implementación de estos patrones puede facilitar el uso de repositorios ficticios que simulan el acceso a la base de datos.

En la figura 7-18, puede ver las diferencias entre no usar repositorios (directamente mediante DBContext de EF) frente al uso de repositorios, lo que facilita la simulación de esos repositorios.

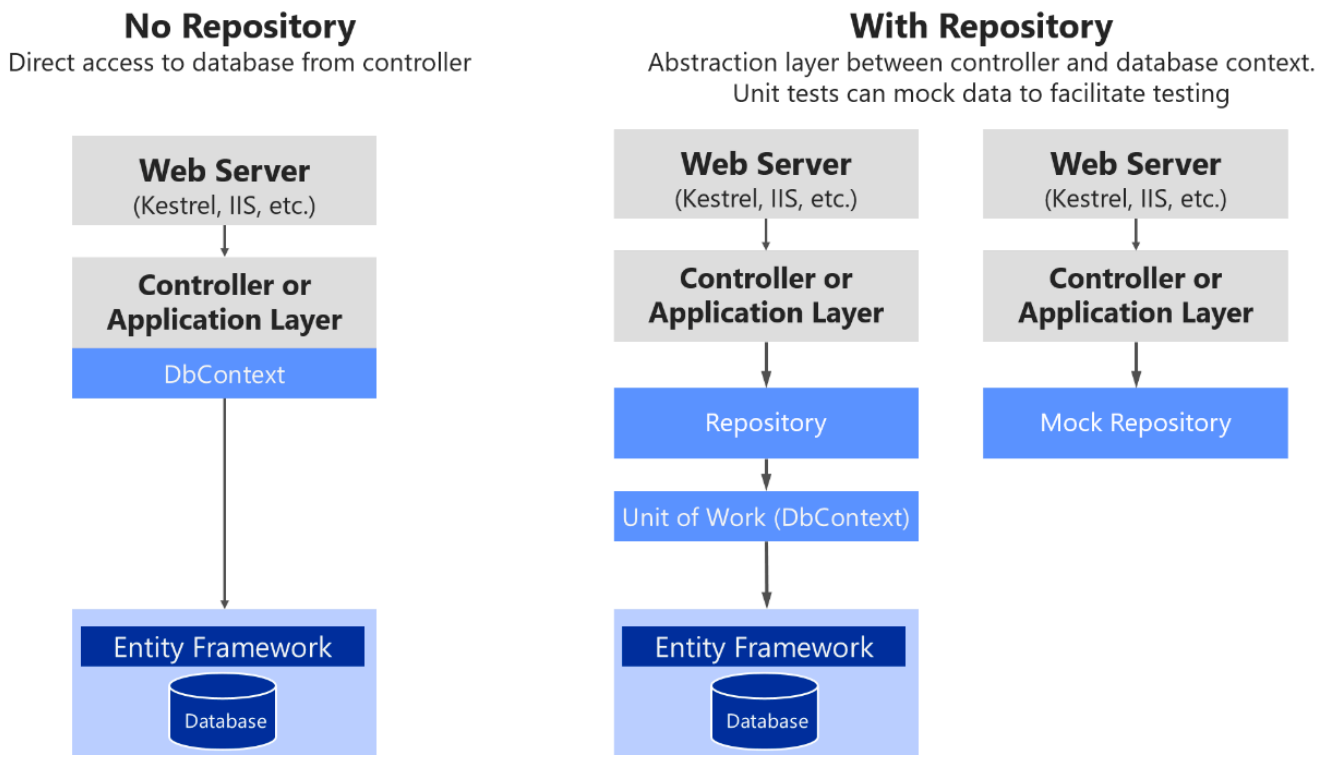


Figura 7-18. Uso de repositorios personalizados frente a DbContext sin formato

En la figura 7-18 se muestra que el uso de un repositorio personalizado agrega una capa de abstracción que se puede usar para facilitar las pruebas simulando el repositorio. Hay varias alternativas al plantear una simulación. Podría simular solo repositorios o podría simular una unidad de trabajo completa. Normalmente es suficiente con simular repositorios y no suele ser necesario pasar por la complejidad de tener que abstraer y simular una unidad de trabajo.

Más adelante, cuando nos centramos en la capa de aplicación, verá cómo funciona la inserción de dependencias en ASP.NET Core y cómo se implementa al usar repositorios.

En resumen, los repositorios personalizados permiten probar el código más fácilmente con pruebas unitarias que no se ven afectadas por el estado del nivel de datos. Si ejecuta pruebas que también acceden a la base de datos real a través de Entity Framework, no son pruebas unitarias, sino pruebas de integración, que son mucho más lentas.

Si estuviera usando DbContext directamente, tendría que simularlo o ejecutar pruebas unitarias utilizando una versión en memoria de SQL Server con datos predecibles para pruebas unitarias. Pero simular la clase DbContext o controlar datos falsos requiere más trabajo que la simulación en el nivel de repositorio. Por supuesto, siempre podría probar los controladores MVC.

Duración de la instancia de EF DbContext e IUnitOfWork en el contenedor de IoC

El `DbContext` objeto (expuesto como un `IUnitOfWork` objeto) debe compartirse entre varios repositorios dentro del mismo ámbito de solicitud HTTP. Por ejemplo, esto es cierto cuando la operación que se ejecuta debe tratar con varios agregados, o simplemente porque usa varias instancias de repositorio. También es importante mencionar que la interfaz de `IUnitOfWork` forma parte del nivel de dominio, no es un tipo de EF Core.

Para ello, la instancia del objeto `DbContext` debe tener su duración de servicio establecida en ServiceLifetime.Scoped. Esta es la duración predeterminada al registrar un `DbContext` con `builder.Services.AddDbContext` en el contenedor de IoC desde el archivo *Program.cs* en el proyecto de API web de ASP.NET Core. El código siguiente ilustra esto.

C#

```
// Add framework services.
builder.Services.AddMvc(options =>
{
    options.Filters.Add(typeof(HttpGlobalExceptionFilter));
}).AddControllersAsServices();

builder.Services.AddEntityFrameworkSqlServer()
    .AddDbContext<OrderingContext>(options =>
    {
        options.UseSqlServer(Configuration["ConnectionString"],
                             sqlOptions =>
sqlOptions.MigrationsAssembly(typeof(Startup).GetTypeInfo().

Assembly.GetName().Name));
    },
    ServiceLifetime.Scoped // Note that Scoped is the default choice
                           // in AddDbContext. It is shown here only for
                           // pedagogic purposes.
    );
```

El modo de creación de instancias de DbContext no debe configurarse como ServiceLifetime.Transient o ServiceLifetime.Singleton.

Duración de vida de la instancia del repositorio en el contenedor IoC

De forma similar, la duración del repositorio normalmente debe establecerse a nivel de ámbito (InstancePerLifetimeScope en Autofac). También puede ser transitorio (InstancePerDependency en Autofac), pero el servicio será más eficaz en lo que respecta a la memoria si se usa la duración de ámbito.

C#

```
// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
    .As<IOrderRepository>()
    .InstancePerLifetimeScope();
```

El uso de la duración de singleton para el repositorio puede causar problemas de simultaneidad graves al establecer DbContext en una duración con ámbito (InstancePerLifetimeScope) (las duraciones predeterminadas para DBContext). Siempre que las duraciones del servicio para los repositorios y DbContext tengan establecido un ámbito, evitará estos problemas.

Recursos adicionales

- **Implementación del repositorio y los patrones de unidad de trabajo en una aplicación MVC de ASP.NET**
  https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

- **Jonathan Allen. Estrategias de implementación para el patrón de repositorio con Entity Framework, Dapper y Chain**
  https://www.infoq.com/articles/repository-implementation-strategies

- **Cesar de la Torre. Comparación de los tiempos de vida del servicio del contenedor IoC de ASP.NET Core con los alcances de instancia del contenedor IoC de Autofac**
  https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Asignación de tabla

El mapeo de tablas identifica los datos de la tabla que se van a consultar y guardar en la base de datos. Anteriormente vio cómo se pueden usar las entidades de dominio (por ejemplo, un dominio de producto o pedido) para generar un esquema de base de datos relacionado. EF está fuertemente diseñado en torno al concepto de *convenciones*. Las convenciones abordan preguntas como "¿Cuál será el nombre de una tabla?" o "¿Qué propiedad es la clave principal?" Normalmente, las convenciones se basan en nombres convencionales. Por ejemplo, es habitual que la clave principal sea una propiedad que termine con `Id`.

Por convención, cada entidad se configurará para asignarse a una tabla que tenga el mismo nombre que la propiedad `DbSet<TEntity>` que expone la entidad en el contexto derivado. Si no

se proporciona ningún `DbSet<TEntity>` valor para la entidad especificada, se usa el nombre de clase.

Anotaciones de datos frente a Fluent API

Hay muchas convenciones adicionales de EF Core y la mayoría de ellas se pueden cambiar mediante anotaciones de datos o fluent API, implementadas en el método OnModelCreating.

Las anotaciones de datos se deben usar en las propias clases del modelo de entidad, que es una forma más intrusiva desde un punto de vista DDD. Esto se debe a que está contaminando el modelo con anotaciones de datos relacionadas con la base de datos de infraestructura. Por otro lado, Fluent API es una manera cómoda de cambiar la mayoría de las convenciones y asignaciones dentro del nivel de infraestructura de persistencia de datos, por lo que el modelo de entidad se limpiará y desacoplará de la infraestructura de persistencia.

Fluent API y el método OnModelCreating

Como se mencionó, para cambiar las convenciones y mapeos, puede usar el método OnModelCreating en la clase DbContext.

El microservicio de ordenación en eShopOnContainers implementa configuraciones y asignaciones explícitas, cuando es necesario, tal y como se muestra en el código siguiente.

C#

```
// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   // ...
   modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
   // Other entities' configuration ...
}

// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);

        orderConfiguration.HasKey(o => o.Id);

        orderConfiguration.Ignore(b => b.DomainEvents);

        orderConfiguration.Property(o => o.Id)
            .UseHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);
```

```
        //Address value object persisted as owned entity type supported since EF
Core 2.0
        orderConfiguration
            .OwnsOne(o => o.Address, a =>
            {
                a.WithOwner();
            });

        orderConfiguration
            .Property<int?>("_buyerId")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("BuyerId")
            .IsRequired(false);

        orderConfiguration
            .Property<DateTime>("_orderDate")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("OrderDate")
            .IsRequired();

        orderConfiguration
            .Property<int>("_orderStatusId")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("OrderStatusId")
            .IsRequired();

        orderConfiguration
            .Property<int?>("_paymentMethodId")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("PaymentMethodId")
            .IsRequired(false);

        orderConfiguration.Property<string>("Description").IsRequired(false);

        var navigation =
orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));

        // DDD Patterns comment:
        //Set as field (New since EF 1.1) to access the OrderItem collection
property through its field
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

        orderConfiguration.HasOne<PaymentMethod>()
            .WithMany()
            .HasForeignKey("_paymentMethodId")
            .IsRequired(false)
            .OnDelete(DeleteBehavior.Restrict);

        orderConfiguration.HasOne<Buyer>()
            .WithMany()
            .IsRequired(false)
            .HasForeignKey("_buyerId");

        orderConfiguration.HasOne(o => o.OrderStatus)
            .WithMany()
```

```
            .HasForeignKey("_orderStatusId");
    }
}
```

Puede establecer todas las asignaciones de la API de Fluent en el mismo `OnModelCreating` método, pero es aconsejable particionar ese código y tener varias clases de configuración, una por entidad, como se muestra en el ejemplo. Especialmente para modelos grandes, es aconsejable tener clases de configuración independientes para configurar diferentes tipos de entidad.

El código del ejemplo muestra algunas declaraciones y mapeos explícitos. Pero las convenciones de EF Core realizan muchas de esas asignaciones automáticamente, por lo que, en su caso, podría necesitar un código más pequeño.

Algoritmo Hi/Lo en EF Core

Un aspecto interesante del código en el ejemplo anterior es que usa el algoritmo Hi/Lo como estrategia de generación de claves.

El algoritmo Hi/Lo es útil cuando se necesitan claves únicas antes de confirmar los cambios. Como resumen, el algoritmo Hi-Lo asigna identificadores únicos a filas de tabla, aunque no depende de almacenar la fila en la base de datos inmediatamente. Esto le permite empezar a usar los identificadores inmediatamente, como sucede con identificadores de base de datos secuenciales normales.

El algoritmo Hi/Lo describe un mecanismo para obtener un lote de identificadores únicos de una secuencia de base de datos relacionada. Estos identificadores son seguros de usar porque la base de datos garantiza la unicidad, por lo que no habrá colisiones entre los usuarios. Este algoritmo es interesante por estas razones:

- No interrumpe el patrón de la unidad de trabajo.
- Obtiene identificadores de secuencia en lotes para minimizar los recorridos de ida y vuelta a la base de datos.
- Genera un identificador legible para humanos, a diferencia de las técnicas que usan Identificadores Únicos Globales (GUID).

EF Core admite HiLo con el `UseHiLo` método , como se muestra en el ejemplo anterior.

Mapear campos en lugar de propiedades

Con esta característica, disponible desde EF Core 1.1, puede asignar directamente columnas a campos. Es posible no usar propiedades en la clase de entidad y simplemente asignar columnas de una tabla a campos. Un uso habitual de ello serían los campos privados para cualquier estado interno, al que no sea necesario acceder desde fuera de la entidad.

Puede hacerlo con campos individuales o también con colecciones, como un `List<>` campo. Este punto se mencionó anteriormente al hablar del modelado de las clases del modelo de dominio, pero aquí puede ver cómo se realiza ese mapeo con la configuración `PropertyAccessMode.Field` destacada en el código anterior.

Uso de propiedades reemplazadas en EF Core y ocultas en el nivel de infraestructura

Las "shadow properties" en EF Core son propiedades que no existen en el modelo de clase de entidad. Los valores y estados de estas propiedades se mantienen exclusivamente en la clase ChangeTracker en el nivel de infraestructura.

Implementación del patrón de especificación de consulta

Como se introdujo anteriormente en la sección de diseño, el patrón especificación de consulta es un patrón de diseño Domain-Driven diseñado como el lugar donde puede colocar la definición de una consulta con lógica de ordenación y paginación opcionales.

El patrón Especificación de consulta define una consulta en un objeto . Por ejemplo, para encapsular una consulta paginada que busque algunos productos, puede crear una especificación PagedProduct que tome los parámetros de entrada necesarios (pageNumber, pageSize, filter, etc.). A continuación, dentro de cualquier método del Repositorio (habitualmente como una sobrecarga de List()) se aceptaría un IQuerySpecification y se ejecutaría la consulta esperada basada en esa especificación.

Un ejemplo de una interfaz de especificación genérica es el código siguiente, que es similar al código usado en la aplicación de referencia eShopOnWeb .

C#

```
// GENERIC SPECIFICATION INTERFACE
// https://github.com/dotnet-architecture/eShopOnWeb

public interface ISpecification<T>
{
    Expression<Func<T, bool>> Criteria { get; }
    List<Expression<Func<T, object>>> Includes { get; }
```

```
    List<string> IncludeStrings { get; }
}
```

A continuación, la implementación de una clase base de especificación genérica es la siguiente.

C#

```
// GENERIC SPECIFICATION IMPLEMENTATION (BASE CLASS)
// https://github.com/dotnet-architecture/eShopOnWeb

public abstract class BaseSpecification<T> : ISpecification<T>
{
    public BaseSpecification(Expression<Func<T, bool>> criteria)
    {
        Criteria = criteria;
    }
    public Expression<Func<T, bool>> Criteria { get; }

    public List<Expression<Func<T, object>>> Includes { get; } =
                                        new List<Expression<Func<T, object>>>
();

    public List<string> IncludeStrings { get; } = new List<string>();

    protected virtual void AddInclude(Expression<Func<T, object>>
includeExpression)
    {
        Includes.Add(includeExpression);
    }

    // string-based includes allow for including children of children
    // for example, Basket.Items.Product
    protected virtual void AddInclude(string includeString)
    {
        IncludeStrings.Add(includeString);
    }
}
```

La siguiente especificación carga una entidad de cesta única a partir del identificador de cesta o del identificador del comprador al que pertenece la cesta y realiza una carga diligente de la colección `Items` de la cesta.

C#

```
// SAMPLE QUERY SPECIFICATION IMPLEMENTATION

public class BasketWithItemsSpecification : BaseSpecification<Basket>
{
    public BasketWithItemsSpecification(int basketId)
        : base(b => b.Id == basketId)
    {
```

```
        AddInclude(b => b.Items);
    }

    public BasketWithItemsSpecification(string buyerId)
        : base(b => b.BuyerId == buyerId)
    {
        AddInclude(b => b.Items);
    }
}
```

Por último, puede ver cómo un repositorio EF genérico puede usar esta especificación para filtrar y cargar datos relacionados con un tipo de entidad determinado T.

C#

```
// GENERIC EF REPOSITORY WITH SPECIFICATION
// https://github.com/dotnet-architecture/eShopOnWeb

public IEnumerable<T> List(ISpecification<T> spec)
{
    // fetch a Queryable that includes all expression-based includes
    var queryableResultWithIncludes = spec.Includes
        .Aggregate(_dbContext.Set<T>().AsQueryable(),
            (current, include) => current.Include(include));

    // modify the IQueryable to include any string-based include statements
    var secondaryResult = spec.IncludeStrings
        .Aggregate(queryableResultWithIncludes,
            (current, include) => current.Include(include));

    // return the result of the query using the specification's criteria
expression
    return secondaryResult
                    .Where(spec.Criteria)
                    .AsEnumerable();
}
```

Además de encapsular la lógica de filtrado, la especificación puede especificar la forma de los datos que se van a devolver, incluidas las propiedades que se van a rellenar.

Aunque no se recomienda devolver `IQueryable` desde un repositorio, es perfectamente adecuado usarlos en el repositorio para crear un conjunto de resultados. Puede ver este enfoque usado en el método List anterior, que usa expresiones intermedias `IQueryable` para crear la lista de inclusión de la consulta antes de ejecutar la consulta con los criterios de la especificación en la última línea.

Obtenga información sobre cómo se aplica el patrón de especificación en el ejemplo eShopOnWeb.

Recursos adicionales

- **Asignación de tablas**
  https://learn.microsoft.com/ef/core/modeling/relational/tables
- **Uso de HiLo para generar claves con Entity Framework Core**
  https://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/
- **Campos de respaldo**
  https://learn.microsoft.com/ef/core/modeling/backing-field
- **Steve Smith. Colecciones encapsuladas en Entity Framework Core**
  https://ardalis.com/encapsulated-collections-in-entity-framework-core
- **Propiedades de sombra**
  https://learn.microsoft.com/ef/core/modeling/shadow-properties
- **El patrón de especificación**
  https://deviq.com/specification-pattern/

  Paquete NuGet Ardalis.Specification Usado por eShopOnWeb. \
  https://www.nuget.org/packages/Ardalis.Specification

Siguiente anterior

Uso de bases de datos NoSQL como una infraestructura de persistencia

Artículo • 10/09/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Cuando se usan bases de datos NoSQL para el nivel de datos de infraestructura, normalmente no se utiliza un ORM como Entity Framework Core. En su lugar, se utiliza la API proporcionada por el motor de NoSQL, como por ejemplo Azure Cosmos DB, MongoDB, Cassandra, RavenDB, CouchDB o tablas de Azure Storage.

Pero cuando se usa una base de datos NoSQL, especialmente una orientada a documentos como Azure Cosmos DB, CouchDB o RavenDB, la forma de diseñar el modelo con agregados DDD es parcialmente similar a cómo se puede hacer en EF Core, en lo que respecta a la identificación de las raíces agregadas, las clases de entidad secundarias y las clases de objeto de valor. Pero, en última instancia, la selección de la base de datos afectará al diseño.

Cuando utilice una base de datos orientada a documentos, implemente un agregado como un solo documento serializado en JSON o en otro formato. Pero el uso de la base de datos es transparente desde un punto de vista del código de dominio de modelo. Cuando se usa una base de datos NoSQL, también se utilizan las clases de entidad y las clases raíz de agregado, pero con más flexibilidad que cuando se usa EF Core porque la persistencia no es relacional.

La diferencia radica en cómo se persiste ese modelo. Si implementa el modelo de dominio basándose en las clases de entidad POCO, independientemente de la persistencia de la infraestructura, quizá parezca que puede cambiar a una infraestructura de persistencia diferente, incluso desde relacional a NoSQL. Pero ese no debería ser el objetivo. Siempre hay restricciones y contrapartidas en las diferentes tecnologías de bases de datos, por lo que no se podrá tener el mismo modelo para bases de datos relacionales o NoSQL. Cambiar modelos de persistencia tiene su importancia, dado que las transacciones y las operaciones de persistencia serán muy diferentes.

Por ejemplo, en una base de datos orientada a documentos, es correcto que una raíz agregada tenga varias propiedades de colección secundaria. En una base de datos relacional, consultar varias propiedades de colección secundaria no está bien optimizado, porque se recibe una instrucción UNION ALL SQL de EF. Tener el mismo modelo de dominio para bases de datos relacionales o bases de datos NoSQL no es sencillo y no debería intentarse. El modelo debe diseñarse entendiendo el uso que se va a hacer de los datos en cada base de datos en particular.

Una ventaja de utilizar las bases de datos NoSQL es que las entidades estén menos normalizadas, por lo que no se establece una asignación de tabla. El modelo de dominio puede ser más flexible que al utilizar una base de datos relacional.

Al diseñar el modelo de dominio basándose en agregados, el cambio a bases de datos NoSQL y orientadas a documentos podría ser incluso más sencillo que usar una base de datos relacional, puesto que los agregados que se diseñan son similares a documentos serializados en una base de datos orientada a documentos. Luego puede incluir en esas "bolsas" toda la información que necesite para ese agregado.

Por ejemplo, el siguiente código JSON es una implementación de ejemplo de un agregado de pedido cuando se usa una base de datos orientada a documentos. Es similar al orden agregado de pedido que implementamos en el ejemplo eShopOnContainers, pero sin utilizar EF Core debajo.

JSON

```
{
    "id": "2024001",
    "orderDate": "2/25/2024",
    "buyerId": "1234567",
    "address": [
        {
        "street": "100 One Microsoft Way",
        "city": "Redmond",
        "state": "WA",
        "zip": "98052",
        "country": "U.S."
        }
```

```
    ],
    "orderItems": [
        {"id": 20240011, "productId": "123456", "productName": ".NET T-
Shirt",
        "unitPrice": 25, "units": 2, "discount": 0},
        {"id": 20240012, "productId": "123457", "productName": ".NET Mug",
        "unitPrice": 15, "units": 1, "discount": 0}
    ]
}
```

Introducción a Azure Cosmos DB y la API Cosmos DB nativa

Azure Cosmos DB es el servicio de base de datos distribuida globalmente de Microsoft para aplicaciones críticas. Azure Cosmos DB proporciona distribución global inmediata, escalado flexible de rendimiento y almacenamiento en todo el mundo, latencias de milisegundo de un solo dígito en el percentil 99, cinco niveles de coherencia bien definidos y alta disponibilidad garantizada, todo ello respaldado por los mejores SLA del sector. Azure Cosmos DB indiza automáticamente los datos sin necesidad de administrar el esquema y el índice. Sigue varios modelos y es compatible con los modelos de datos de documento, de clave-valor, de grafo y de columnas.

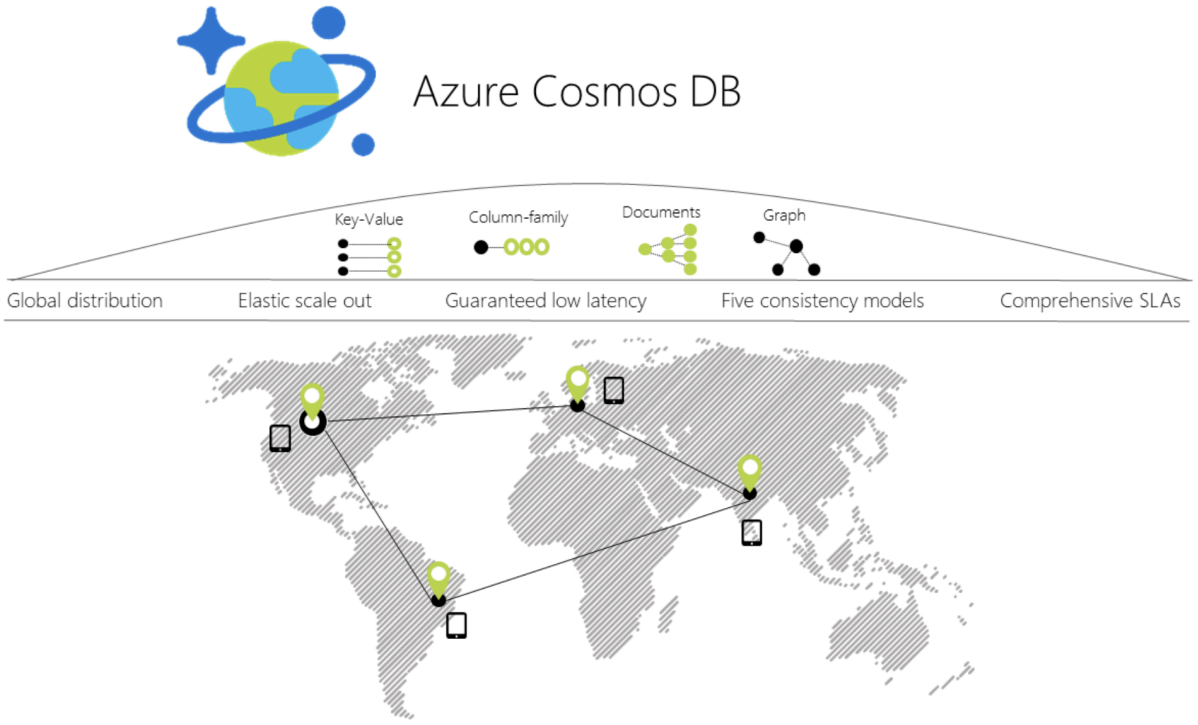


Figura 7-19. Distribución global de Azure Cosmos DB

Cuando se usa un modelo C# para implementar el agregado que va a usar la API de Azure Cosmos DB, el agregado puede ser similar a las clases POCO de C# que se usan

con EF Core. La diferencia radica en la forma de usarlos desde la aplicación y las capas de la infraestructura, como en el código siguiente:

C#

```
// C# EXAMPLE OF AN ORDER AGGREGATE BEING PERSISTED WITH AZURE COSMOS DB API
// *** Domain Model Code ***
// Aggregate: Create an Order object with its child entities and/or value
objects.
// Then, use AggregateRoot's methods to add the nested objects so invariants
and
// logic is consistent across the nested properties (value objects and
entities).

Order orderAggregate = new Order
{
    Id = "2024001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "PO18009186470"
}

Address address = new Address
{
    Street = "100 One Microsoft Way",
    City = "Redmond",
    State = "WA",
    Zip = "98052",
    Country = "U.S."
}

orderAggregate.UpdateAddress(address);

OrderItem orderItem1 = new OrderItem
{
    Id = 20240011,
    ProductId = "123456",
    ProductName = ".NET T-Shirt",
    UnitPrice = 25,
    Units = 2,
    Discount = 0;
};

//Using methods with domain logic within the entity. No anemic-domain model
orderAggregate.AddOrderItem(orderItem1);
// *** End of Domain Model Code ***

// *** Infrastructure Code using Cosmos DB Client API ***
Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName,
    collectionName);

await client.CreateDocumentAsync(collectionUri, orderAggregate);

// As your app evolves, let's say your object has a new schema. You can
```

```
insert
// OrderV2 objects without any changes to the database tier.
Order2 newOrder = GetOrderV2Sample("IdForSalesOrder2");
await client.CreateDocumentAsync(collectionUri, newOrder);
```

Puede ver que la forma de trabajar con el modelo de dominio puede ser similar a la manera en que se utiliza en la capa de modelo de dominio cuando la infraestructura es EF. Se siguen usando los mismos métodos raíz de agregación para garantizar la coherencia, las invariantes y las validaciones en el agregado.

Pero cuando se persiste el modelo en la base de datos NoSQL, el código y la API cambian drásticamente en comparación con el código de EF Core o cualquier otro código relacionado con las bases de datos relacionales.

Implementación de código de .NET destinado a MongoDB y Azure Cosmos DB

Uso de Azure Cosmos DB desde contenedores de .NET

Se puede acceder a las bases de datos de Azure Cosmos DB desde código de .NET que se ejecuta en contenedores, como en cualquier otra aplicación. NET. Por ejemplo, los microservicios Locations.API y Marketing.API de eShopOnContainers se implementan para que puedan utilizar las bases de datos de Azure Cosmos DB.

Pero hay una limitación en Azure Cosmos DB desde un punto de vista del entorno de desarrollo Docker. Aunque hay un emulador de Azure Cosmos DB local que se puede ejecutar en una máquina de desarrollo local, este solo es compatible con Windows. No se admiten Linux ni macOS.

También existe la posibilidad de ejecutar este emulador en Docker, pero solo en los contenedores de Windows, no en los de Linux. Eso es un impedimento inicial para el entorno de desarrollo si la aplicación se implementa como contenedores de Linux, puesto que actualmente no es posible implementar al mismo tiempo contenedores de Windows y Linux en Docker para Windows. Todos los contenedores que se implementen tienen que ser de Linux o de Windows.

La implementación ideal y más sencilla para una solución de desarrollo o pruebas consiste en ser capaz de implementar los sistemas de base de datos como contenedores junto con los contenedores personalizados para que sus entornos de desarrollo o pruebas sean siempre coherentes.

Uso de la API de MongoDB para contenedores locales de desarrollo o pruebas de Linux y Windows además de Azure Cosmos DB

Las bases de datos de COSMOS DB son compatibles con la API de MongoDB para. NET, además de con el protocolo de conexión de MongoDB nativo. Esto significa que, mediante los controladores existentes, la aplicación escrita para MongoDB ahora puede comunicarse con Cosmos DB y usar las bases de datos de Cosmos DB en lugar de las bases de datos de MongoDB, como se muestra en la figura 7-20.

Figura 7-20. Uso de la API de MongoDB y el protocolo para acceder Azure Cosmos DB

Esto es un método muy práctico para la prueba de conceptos en entornos de Docker con contenedores Linux porque la imagen de MongoDB Docker es una imagen multiarquitectura que admite contenedores de Docker de Linux y Windows.

Como se muestra en la siguiente imagen, mediante la API de MongoDB, eShopOnContainers admite contenedores de MongoDB de Linux y Windows para el entorno de desarrollo local. Después, puede mover a una solución de nube PaaS escalable como Azure Cosmos DB simplemente cambiando la cadena de conexión de MongoDB para que apunte a Azure Cosmos DB.

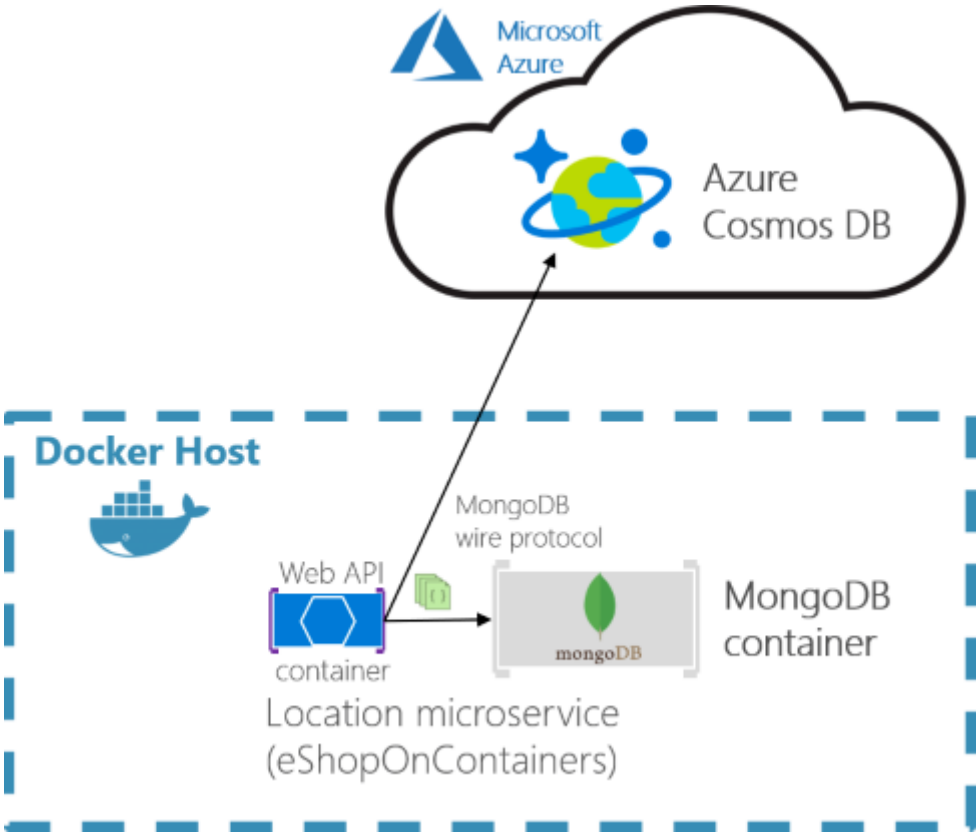


Figura 7-21. eShopOnContainers con contenedores de MongoDB para desarrollo o entorno o Azure Cosmos DB para producción

La base de datos Azure Cosmos DB de producción se ejecuta en la nube de Azure como servicio escalable y de PaaS.

Los contenedores de .NET personalizados pueden ejecutarse en un host de Docker de desarrollo local (es decir, con Docker para Windows en una máquina con Windows 10) o implementarse en un entorno de producción, como Kubernetes en Azure AKS o Azure Service Fabric. En este segundo entorno, implemente solo los contenedores personalizados de .NET, pero no el contenedor de MongoDB, ya que usaría Azure Cosmos DB en la nube para controlar los datos de producción.

Una ventaja evidente de utilizar la API de MongoDB es que la solución puede ejecutarse en dos motores de base de datos, MongoDB o Azure Cosmos DB, por lo que sería fácil migrar a otros entornos. Pero en ocasiones merece la pena usar una API nativa (es decir, la API de Cosmos DB nativa) con el fin de aprovechar al máximo las capacidades de un determinado motor de base de datos.

Para comparar el uso de MongoDB frente a Cosmos DB en la nube, consulte las ventajas de usar Azure Cosmos DB en esta página.

Análisis del enfoque para aplicaciones de producción: API de MongoDB frente a API de Cosmos DB

En eShopOnContainers, usamos API de MongoDB porque nuestra prioridad era fundamentalmente tener un entorno de desarrollo o pruebas coherente con una base de datos NoSQL que también pudiese funcionar con Azure Cosmos DB.

Pero si planea usar la API MongoDB para acceder a Azure Cosmos DB en Azure para aplicaciones de producción, debe analizar y comparar las diferencias entre las funciones y el rendimiento al usar la API MongoDB para acceder a bases de datos de Azure Cosmos DB y usar la API nativa de Azure Cosmos DB. Si el resultado es similar, se puede utilizar la API de MongoDB, con la ventaja de admitir dos motores de base de datos NoSQL al mismo tiempo.

También podría utilizar clústeres de MongoDB como base de datos de producción en la nube de Azure, con MongoDB Azure Service. Pero eso no es un servicio PaaS proporcionado por Microsoft. En este caso, Azure solo hospeda la solución procedente de MongoDB.

Básicamente, esto es simplemente una declinación de responsabilidades que indica que no debe usar siempre la API de MongoDB en Azure Cosmos DB, como hicimos en eShopOnContainers, puesto que se trataba de una opción conveniente para los contenedores de Linux. La decisión debe basarse en las necesidades específicas y las pruebas que deba hacer en la aplicación de producción.

El código: uso de la API de MongoDB en aplicaciones .NET

La API de MongoDB para .NET se basa en los paquetes NuGet que debe agregar a los proyectos, como en el proyecto Locations.API que se muestra en la siguiente imagen.

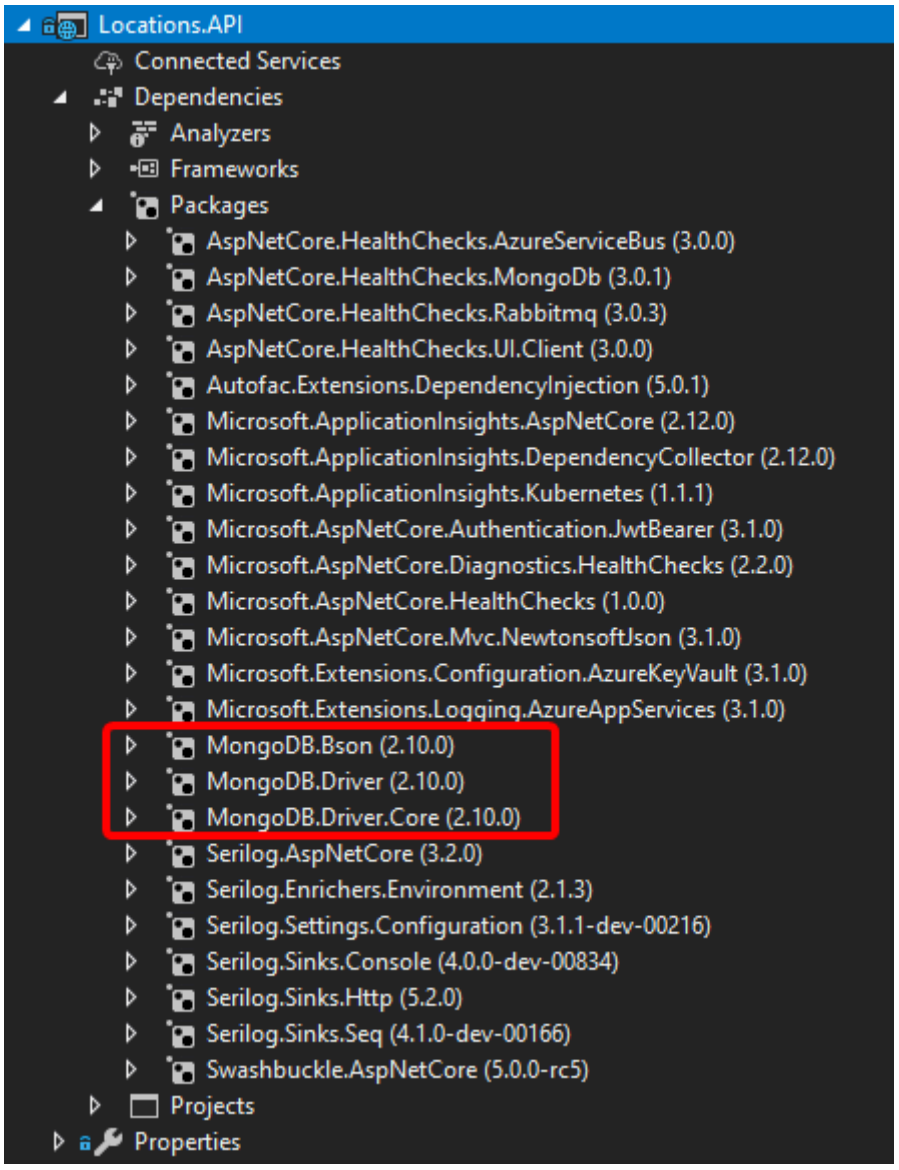


Figura 7-22. Referencias de paquetes NuGet de la API MongoDB en un proyecto de .NET

En las secciones siguientes investigaremos el código.

Un modelo usado por la API de MongoDB

En primer lugar, debe definir un modelo que contendrá los datos procedentes de la base de datos en el espacio de memoria de la aplicación. Este es un ejemplo del modelo que se usa para Locations en eShopOnContainers.

C#

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Driver.GeoJsonObjectModel;
using System.Collections.Generic;

public class Locations
{
```

```
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    public int LocationId { get; set; }
    public string Code { get; set; }
    [BsonRepresentation(BsonType.ObjectId)]
    public string Parent_Id { get; set; }
    public string Description { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public GeoJsonPoint<GeoJson2DGeographicCoordinates> Location
                                                            { get; private
set; }
    public GeoJsonPolygon<GeoJson2DGeographicCoordinates> Polygon
                                                            { get; private
set; }
    public void SetLocation(double lon, double lat) => SetPosition(lon,
lat);
    public void SetArea(List<GeoJson2DGeographicCoordinates>
coordinatesList)
                                                        =>
SetPolygon(coordinatesList);

    private void SetPosition(double lon, double lat)
    {
        Latitude = lat;
        Longitude = lon;
        Location = new GeoJsonPoint<GeoJson2DGeographicCoordinates>(
            new GeoJson2DGeographicCoordinates(lon, lat));
    }

    private void SetPolygon(List<GeoJson2DGeographicCoordinates>
coordinatesList)
    {
        Polygon = new GeoJsonPolygon<GeoJson2DGeographicCoordinates>(
                  new
GeoJsonPolygonCoordinates<GeoJson2DGeographicCoordinates>(
                  new
GeoJsonLinearRingCoordinates<GeoJson2DGeographicCoordinates>(

coordinatesList)));
    }
}
```

Puede ver que hay unos cuantos atributos y tipos procedentes de los paquetes NuGet de MongoDB.

Las bases de datos NoSQL suelen ser muy adecuadas para trabajar con datos jerárquicos no relacionales. En este ejemplo se usan tipos de MongoDB especiales para ubicaciones geográficas, como `GeoJson2DGeographicCoordinates`.

Recuperación de la base de datos y la colección

En eShopOnContainers, hemos creado un contexto de base de datos personalizado donde implementamos el código para recuperar la base de datos y MongoCollections, como se muestra en el código siguiente.

C#

```
public class LocationsContext
{
    private readonly IMongoDatabase _database = null;

    public LocationsContext(IOptions<LocationSettings> settings)
    {
        var client = new MongoClient(settings.Value.ConnectionString);
        if (client != null)
            _database = client.GetDatabase(settings.Value.Database);
    }

    public IMongoCollection<Locations> Locations
    {
        get
        {
            return _database.GetCollection<Locations>("Locations");
        }
    }
}
```

Recuperación de los datos

En código C#, al igual que con controladores de API Web o implementación de repositorios personalizada, puede escribir un código similar al siguiente al consultar a través de la API de MongoDB. Tenga en cuenta que el objeto `_context` es una instancia de la clase `LocationsContext` anterior.

C#

```
public async Task<Locations> GetAsync(int locationId)
{
    var filter = Builders<Locations>.Filter.Eq("LocationId", locationId);
    return await _context.Locations
                            .Find(filter)
                            .FirstOrDefaultAsync();
}
```

Uso de env-var en el archivo docker-compose.override.yml para la cadena de conexión de MongoDB

Al crear un objeto MongoClient, se necesita un parámetro fundamental que es precisamente el parámetro `ConnectionString` que apunta a la base de datos correcta. En el caso de eShopOnContainers, la cadena de conexión puede apuntar a un contenedor local de MongoDB Docker local o a una base de datos de "producción" de Azure Cosmos DB. Esa cadena de conexión procede de las variables de entorno definidas en los archivos `docker-compose.override.yml` que se utilizan al implementar con docker-compose o Visual Studio, como se muestra en el siguiente código yml.

yml

```yml
# docker-compose.override.yml
version: '3.4'
services:
  # Other services
  locations-api:
    environment:
      # Other settings
      - ConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}
```

> ⓘ **Importante**
>
> Microsoft recomienda usar el flujo de autenticación más seguro disponible. Si se conecta a Azure SQL, el método de autenticación recomendado es [Identidades administradas para recursos de Azure](#).

La variable de entorno `ConnectionString` se resuelve de esta manera: si la variable global `ESHOP_AZURE_COSMOSDB` está definida en el archivo `.env` con la cadena de conexión de Azure Cosmos DB, la usará para acceder a la base de datos de Azure Cosmos DB en la nube. Si no está definida, tomará el valor `mongodb://nosqldata` y usará el contenedor MongoDB de desarrollo.

El código siguiente muestra el archivo `.env` con la variable de entorno global de cadena de conexión de Azure Cosmos DB, tal y como se implementa en eShopOnContainers:

yml

```yml
# .env file, in eShopOnContainers root folder
# Other Docker environment variables

ESHOP_EXTERNAL_DNS_NAME_OR_IP=host.docker.internal
ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=<YourDockerHostIP>
```

```
#ESHOP_AZURE_COSMOSDB=<YourAzureCosmosDBConnData>

#Other environment variables for additional Azure infrastructure assets
#ESHOP_AZURE_REDIS_BASKET_DB=<YourAzureRedisBasketInfo>
#ESHOP_AZURE_STORAGE_CATALOG_URL=<YourAzureStorage_Catalog_BLOB_URL>
#ESHOP_AZURE_SERVICE_BUS=<YourAzureServiceBusInfo>
```

Quite la marca de comentario de la línea ESHOP_AZURE_COSMOSDB y actualícela con su cadena de conexión de Azure Cosmos DB obtenida en Azure Portal como se explica en Conectar una aplicación de MongoDB a Azure Cosmos DB.

Si la variable global `ESHOP_AZURE_COSMOSDB` está vacía, lo que significa que se comenta en el archivo `.env`, el contenedor usará una cadena de conexión de MongoDB predeterminada. Esta cadena de conexión apunta al contenedor de MongoDB local implementado en eShopOnContainers denominado `nosqldata` y definido en el archivo Docker-Compose, como se muestra en el siguiente código .yml:

yml

```yml
# docker-compose.yml
version: '3.4'
services:
  # ...Other services...
  nosqldata:
    image: mongo
```

Recursos adicionales

- **Modelado de datos del documento para bases de datos NoSQL**
  https://learn.microsoft.com/azure/cosmos-db/modeling-data
- **Vaughn Vernon. The Ideal Domain-Driven Design Aggregate Store? (¿El almacén de agregado ideal de diseño controlado por dominio?)**
  https://kalele.io/blog-posts/the-ideal-domain-driven-design-aggregate-store/
- **Introducción a Azure Cosmos DB: API para MongoDB**
  https://learn.microsoft.com/azure/cosmos-db/mongodb-introduction
- **Azure Cosmos DB: Compilación de una aplicación web mediante la API de MongoDB con .NET y Azure Portal**
  https://learn.microsoft.com/azure/cosmos-db/create-mongodb-dotnet
- **Uso del Emulador de Azure Cosmos DB para desarrollo y pruebas de forma local**
  https://learn.microsoft.com/azure/cosmos-db/local-emulator

- **Conectar una aplicación de MongoDB a Azure Cosmos DB**
  https://learn.microsoft.com/azure/cosmos-db/connect-mongodb-account
- **Imagen de Docker de MongoDB (contenedor Linux y Windows)**
  https://hub.docker.com/_/mongo/
- **Azure Cosmos DB: Uso de MongoChef (Studio 3T) con una cuenta de la API de MongoDB**
  https://learn.microsoft.com/azure/cosmos-db/mongodb-mongochef

Anterior Siguiente

Diseñar la capa de aplicación de microservicios y la API web

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Uso de principios SOLID e inserción de dependencias

Los principios SÓLIDOs son técnicas críticas que se usarán en cualquier aplicación moderna y crítica, como desarrollar un microservicio con patrones DDD. SOLID es un acrónimo que agrupa cinco principios fundamentales:

- Principio de responsabilidad única
- Principio abierto y cerrado
- Principio de sustitución de Liskov
- Principio de segregación de interfaz
- Principio de inversión de dependencias

SOLID es más sobre cómo diseñar las capas internas de aplicaciones o microservicios y sobre la desacoplamiento de dependencias entre ellas. No está relacionado con el dominio, sino con el

diseño técnico de la aplicación. El principio final, el principio de inversión de dependencias, permite desacoplar la capa de infraestructura del resto de las capas, lo que permite una mejor implementación desacoplada de las capas DDD.

La inyección de dependencias (DI) es una manera de implementar el principio de inversión de dependencias. Es una técnica para lograr un acoplamiento flexible entre objetos y sus dependencias. En lugar de crear instancias directas de colaboradores o usar referencias estáticas (es decir, mediante new...), los objetos que una clase necesita para realizar sus acciones se inyectan en la clase. A menudo, las clases declararán sus dependencias a través de su constructor, lo que les permitirá seguir el principio Dependencias explícitas. Normalmente, la inserción de dependencias se basa en determinados contenedores de Inversión de control (IoC). ASP.NET Core proporciona un contenedor de IoC integrado sencillo, pero también puede usar su contenedor de IoC favorito, como Autofac o Ninject.

Siguiendo los principios SOLID, las clases tienden a ser pequeñas, bien estructuradas y probadas fácilmente. ¿Pero cómo puede saber si se insertan demasiadas dependencias en las clases? Si usa di a través del constructor, será fácil detectarlo simplemente examinando el número de parámetros del constructor. Si hay demasiadas dependencias, esto suele ser una señal (una intuición de código) de que su clase está intentando hacer demasiado y de que probablemente esté infringiendo el principio de responsabilidad única.

Se necesitaría otra guía para cubrir SOLID en detalle. Por lo tanto, esta guía requiere que solo tenga un conocimiento mínimo de estos temas.

Recursos adicionales

- **SOLID: Principios fundamentales de OOP**
  https://deviq.com/solid/
- **Contenedores de inversión de control y el patrón de inyección de dependencias**
  https://martinfowler.com/articles/injection.html
- **Steve Smith. New is Glue (New es como pegamento)**
  https://ardalis.com/new-is-glue

Siguiente anterior

Implementación del nivel de aplicación de microservicios mediante la API web

Artículo • 29/03/2023

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en **Documentación de .NET** o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Uso de la inserción de dependencias para insertar objetos de la infraestructura en el nivel de aplicación

Como se ha mencionado anteriormente, el nivel de aplicación se puede implementar como parte del artefacto (ensamblado) que se está creando, por ejemplo, dentro de un proyecto de API web o de aplicación web MVC. En el caso de un microservicio compilado con ASP.NET Core, el nivel de aplicación normalmente será la biblioteca de API web. Si quiere separar lo que proviene de ASP.NET Core (su infraestructura y los controladores) del código de nivel de aplicación personalizado, también puede colocar el nivel de aplicación en una biblioteca de clases independiente, pero es algo opcional.

Por ejemplo, el código de nivel de aplicación del microservicio de pedidos se implementa directamente como parte del proyecto **Ordering.API** (un proyecto de API web de ASP.NET Core), como se muestra en la figura 7-23.

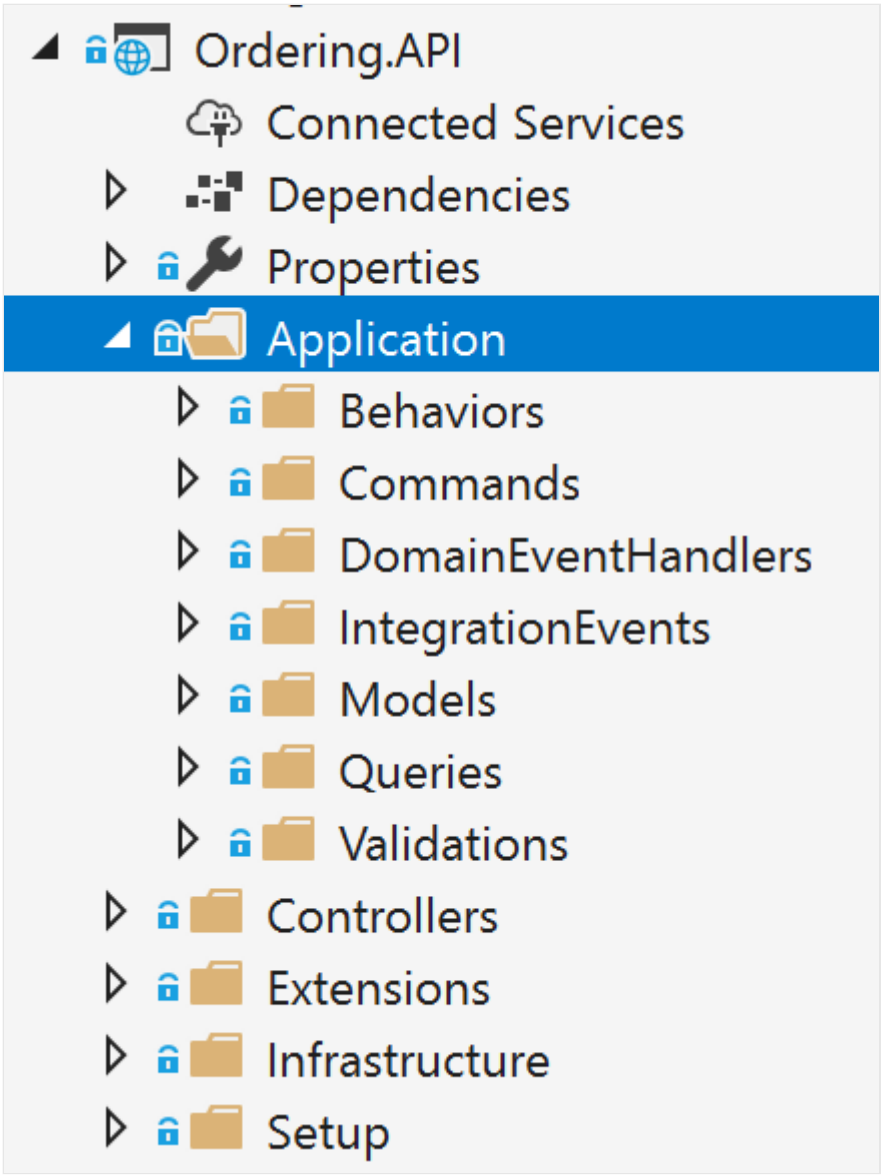


Figura 7-23. Nivel de aplicación en el proyecto de API web de ASP.NET Core Ordering.API

En ASP.NET Core se incluye un simple contenedor de IoC integrado (representado por la interfaz IServiceProvider) que admite la inserción de constructores de forma predeterminada, y ASP.NET hace que determinados servicios estén disponibles a través de DI. En ASP.NET Core se usa el término *servicio* para cualquiera de los tipos que se registran para la inserción mediante DI. Los servicios del contenedor integrado se configuran en el archivo*Program.cs* de la aplicación. Las dependencias se implementan en los servicios que un tipo necesita y que se registran en el contenedor IoC.

Normalmente, le interesará insertar dependencias que implementen objetos de infraestructura. Una dependencia habitual para insertar es un repositorio. Pero también podría insertar cualquier otra dependencia de infraestructura que pueda tener. Para las implementaciones más sencillas, también podría insertar directamente el objeto de patrón de unidades de trabajo (el objeto DbContext de EF), porque DBContext también es la implementación de los objetos de persistencia de infraestructura.

En el ejemplo siguiente, puede ver cómo .NET inserta los objetos necesarios del repositorio mediante el constructor. La clase es un controlador de comandos, que se explica en esta sección.

C#

```
public class CreateOrderCommandHandler
        : IRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;
    private readonly IOrderingIntegrationEventService
_orderingIntegrationEventService;
    private readonly ILogger<CreateOrderCommandHandler> _logger;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService identityService,
        ILogger<CreateOrderCommandHandler> logger)
    {
        _orderRepository = orderRepository ?? throw new
ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new
ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new
ArgumentNullException(nameof(mediator));
        _orderingIntegrationEventService = orderingIntegrationEventService
?? throw new ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<bool> Handle(CreateOrderCommand message,
CancellationToken cancellationToken)
    {
        // Add Integration event to clean the basket
        var orderStartedIntegrationEvent = new
OrderStartedIntegrationEvent(message.UserId);
        await
_orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegratio
nEvent);

        // Add/Update the Buyer AggregateRoot
        // DDD patterns comment: Add child entities and value-objects
through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business
logic
        // make sure that consistency is preserved across the whole
aggregate
        var address = new Address(message.Street, message.City,
message.State, message.Country, message.ZipCode);
```

```
        var order = new Order(message.UserId, message.UserName, address,
message.CardTypeId, message.CardNumber, message.CardSecurityNumber,
message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName,
item.UnitPrice, item.Discount, item.PictureUrl, item.Units);
        }

        _logger.LogInformation("----- Creating Order - Order: {@Order}",
order);

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);
    }
}
```

En la clase se usan los repositorios insertados para ejecutar la transacción y conservar los cambios de estado. No importa si esa clase es un controlador de comandos, un método de controlador de API web de ASP.NET Core, o un servicio de aplicación DDD. En última instancia, es una clase simple que usa repositorios, entidades de dominio y otra coordinación de aplicaciones de forma similar a un controlador de comandos. La inserción de dependencias funciona igual en todas las clases mencionadas, como en el ejemplo de uso de DI según el constructor.

Registro de los tipos de implementación de dependencias e interfaces o abstracciones

Antes de usar los objetos insertados mediante constructores, debe saber dónde registrar las interfaces y clases que generan los objetos que se insertan en las clases de aplicación a través de DI. (Como la inserción de dependencias basada en el constructor, tal y como se mostró anteriormente).

Uso del contenedor de IoC integrado proporcionado por ASP.NET Core

Cuando use el contenedor de IoC integrado que proporciona ASP.NET Core, debe registrar los tipos que quiera insertar en el archivo *Program.cs*, tal como se muestra en el código siguiente:

```
C#
```

```
// Register out-of-the-box framework services.
builder.Services.AddDbContext<CatalogContext>(c =>
    c.UseSqlServer(Configuration["ConnectionString"]),
    ServiceLifetime.Scoped);

builder.Services.AddMvc();
// Register custom application dependencies.
builder.Services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();
```

El modelo más común al registrar los tipos en un contenedor de IoC es registrar un par de tipos: una interfaz y su clase de implementación relacionada. Después, cuando se solicita un objeto del contenedor de IoC a través de cualquier constructor, se solicita un objeto de un tipo de interfaz determinado. En el ejemplo anterior, la última línea indica que, cuando cualquiera de los constructores tiene una dependencia de IMyCustomRepository (interfaz o abstracción), el contenedor de IoC insertará una instancia de la clase de implementación MyCustomSQLServerRepository.

Uso de la biblioteca Scrutor para el registro de tipos automático

Al usar DI en .NET, es posible que le interese poder examinar un ensamblado y registrar sus tipos de manera automática por convención. Actualmente, esta característica no está disponible en ASP.NET Core, pero puede usar la biblioteca Scrutor para hacerlo. Este enfoque resulta conveniente cuando existen docenas de tipos que deben registrarse en el contenedor de IoC.

Recursos adicionales

- **Matthew King. Registering services with Scrutor (Registro de servicios con Scrutor)**
  https://www.mking.net/blog/registering-services-with-scrutor
- **Kristian Hellang. Scrutor.** Repositorio de GitHub.
  https://github.com/khellang/Scrutor

Uso de Autofac como un contenedor de IoC

También se pueden usar contenedores de IoC adicionales y conectarlos a la canalización de ASP.NET Core, como se muestra en el microservicio de pedidos en eShopOnContainers, donde se usa Autofac. Cuando se usa Autofac normalmente los tipos se registran a través de módulos, lo que permite dividir los tipos de registro entre varios archivos, en función de dónde se encuentren los tipos, al igual que los tipos de aplicaciones podrían estar distribuidos entre varias bibliotecas de clases.

Por ejemplo, el siguiente es el módulo de aplicación de Autofac para el proyecto de API web Ordering.API con los tipos que se quieren insertar.

C#

```
public class ApplicationModule : Autofac.Module
{
    public string QueriesConnectionString { get; }
    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }

    protected override void Load(ContainerBuilder builder)
    {
        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>()
            .InstancePerLifetimeScope();
        builder.RegisterType<BuyerRepository>()
            .As<IBuyerRepository>()
            .InstancePerLifetimeScope();
        builder.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();
        builder.RegisterType<RequestManager>()
            .As<IRequestManager>()
            .InstancePerLifetimeScope();
   }
}
```

Autofac también tiene una característica para analizar ensamblados y registrar tipos por convenciones de nombre.

El proceso de registro y los conceptos son muy similares a la manera en que se pueden registrar tipos con el contenedor integrado de IoC de ASP.NET Core, pero cuando se usa Autofac la sintaxis es un poco diferente.

En el código de ejemplo, la abstracción IOrderRepository se registra junto con la clase de implementación OrderRepository. Esto significa que cada vez que un constructor declare una dependencia a través de la abstracción o la interfaz IOrderRepository, el contenedor de IoC insertará una instancia de la clase OrderRepository.

El tipo de ámbito de la instancia determina cómo se comparte una instancia entre las solicitudes del mismo servicio o dependencia. Cuando se realiza una solicitud de una dependencia, el contenedor de IoC puede devolver lo siguiente:

- Una sola instancia por ámbito de duración (denominada *con ámbito* en el contenedor de IoC de ASP.NET Core).

- Una nueva instancia por dependencia (denominada *transitoria* en el contenedor de IoC de ASP.NET Core).
- Una única instancia que se comparte entre todos los objetos que usan el contenedor de IoC (denominada *singleton* en el contenedor de IoC de ASP.NET Core).

Recursos adicionales

- **Introduction to Dependency Injection in ASP.NET Core** (Introducción a la inserción de dependencias en ASP.NET Core) https://learn.microsoft.com/aspnet/core/fundamentals/dependency-injection
- **Autofac.** Documentación oficial. https://docs.autofac.org/en/latest/
- **Comparing ASP.NET Core IoC container service lifetimes with Autofac IoC container instance scopes** (Comparación de las duraciones de servicio del contenedor IoC de ASP.NET Core con ámbitos de instancia de contenedor Autofac IoC) - Cesar de la Torre. https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Implementación de los patrones de comando y controlador de comandos

En el ejemplo de DI a través del constructor mostrado en la sección anterior, el contenedor de IoC insertaba repositorios a través de un constructor en una clase. ¿Pero exactamente dónde se insertaban? En una API web simple (por ejemplo, el microservicio de catálogo de eShopOnContainers), se insertan en el nivel de controladores de MVC, en un constructor de controlador, como parte de la canalización de solicitud de ASP.NET Core. Pero en el código inicial de esta sección (la clase CreateOrderCommandHandler del servicio Ordering.API en eShopOnContainers), la inserción de dependencias se realiza a través del constructor de un determinado controlador de comandos. Vamos a explicar qué es un controlador de comandos y por qué le interesaría usarlo.

El patrón de comandos está intrínsecamente relacionado con el patrón CQRS que se presentó anteriormente en esta guía. CQRS tiene dos lados. La primera área son las consultas, mediante consultas simplificadas con el micro-ORM Dapper, que se explicó anteriormente. La segunda área son los comandos, el punto inicial para las transacciones y el canal de entrada desde el exterior del servicio.

Como se muestra en la figura 7-24, el patrón se basa en la aceptación de comandos del lado cliente, su procesamiento según las reglas del modelo de dominio y, por último, la conservación de los estados con transacciones.

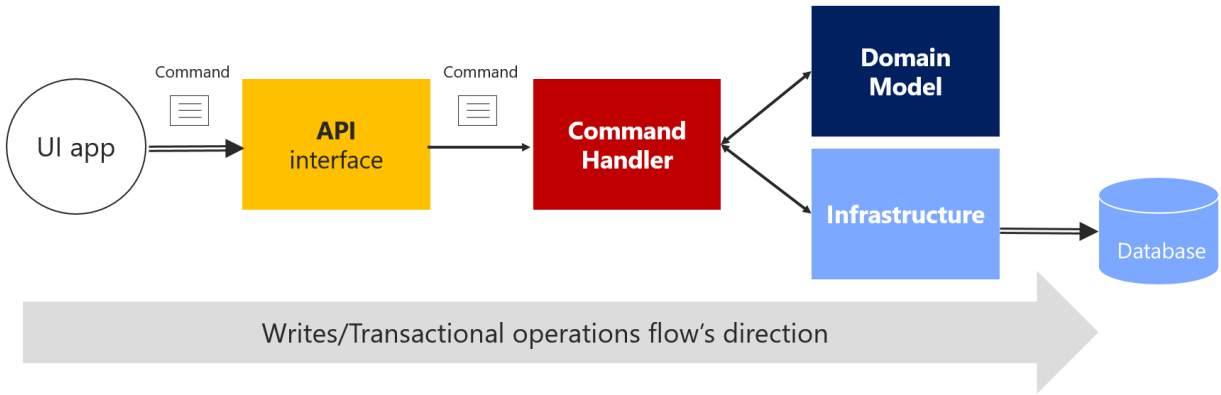


Figura 7-24. Vista general de los comandos o el "lado transaccional" en un patrón CQRS

En la figura 7-24 se muestra que la aplicación de interfaz de usuario envía un comando a través de la API que llega a un elemento `CommandHandler`, que depende del modelo de dominio y de la infraestructura para actualizar la base de datos.

La clase de comando

Un comando es una solicitud para que el sistema realice una acción que cambia el estado del sistema. Los comandos son imperativos y se deben procesar una sola vez.

Como los comandos son imperativos, normalmente se denominan con un verbo en modo imperativo (por ejemplo, "create" o "update"), y es posible que incluyan el tipo agregado, como CreateOrderCommand. A diferencia de un evento, un comando no es un hecho del pasado; es solo una solicitud y, por tanto, se puede denegar.

Los comandos se pueden originar desde la interfaz de usuario como resultado de un usuario que inicia una solicitud, o desde un administrador de procesos cuando está dirigiendo un agregado para realizar una acción.

Una característica importante de un comando es que debe procesarse una sola vez por un único receptor. Esto se debe a que un comando es una única acción o transacción que se quiere realizar en la aplicación. Por ejemplo, el mismo comando de creación de pedidos no se debe procesar más de una vez. Se trata de una diferencia importante entre los comandos y los eventos. Los eventos se pueden procesar varias veces, dado que es posible que muchos sistemas o microservicios estén interesados en el evento.

Además, es importante que un comando solo se procese una vez en caso de que no sea idempotente. Un comando es idempotente si se puede ejecutar varias veces sin cambiar el resultado, ya sea debido a la naturaleza del comando, o bien al modo en que el sistema lo controla.

Un procedimiento recomendado consiste en hacer que los comandos y las actualizaciones sean idempotentes cuando tenga sentido según las reglas de negocio y los elementos invariables del dominio. Para usar el mismo ejemplo, si por algún motivo (lógica de reintento, piratería, etc.) el mismo comando CreateOrder llega varias veces al sistema, debería poder identificarlo y asegurarse de que no se crean varios pedidos. Para ello, debe adjuntar algún tipo de identidad en las operaciones e identificar si el comando o la actualización ya se ha procesado.

Un comando se envía a un único receptor; no se publica. La publicación es para los eventos que notifican un hecho: que ha sucedido algo y que podría ser interesante para los receptores de eventos. En el caso de los eventos, al publicador no le interesa qué receptores obtienen el evento o las acciones que realizan. Pero los eventos de integración o de dominio son diferentes y ya se presentaron en secciones anteriores.

Un comando se implementa con una clase que contiene campos de datos o colecciones con toda la información necesaria para ejecutar ese comando. Un comando es un tipo especial de objeto de transferencia de datos (DTO), que se usa específicamente para solicitar cambios o transacciones. El propio comando se basa en la información exacta que se necesita para procesar el comando y nada más.

En el siguiente ejemplo se muestra la clase `CreateOrderCommand` simplificada. Se trata de un comando inmutable que se usa en el microservicio de pedidos de eShopOnContainers.

C#

```
// DDD and CQRS patterns comment: Note that it is recommended to implement
immutable Commands
// In this case, its immutability is achieved by having all the setters as
private
// plus only being able to update the data just once, when creating the
object through its constructor.
// References on Immutable Commands:
// http://cqrs.nu/Faq
// https://docs.spine3.org/motivation/immutability.html
// http://blog.gauffin.org/2012/06/griffin-container-introducing-command-
support/
// https://learn.microsoft.com/dotnet/csharp/programming-guide/classes-and-
structs/how-to-implement-a-lightweight-class-with-auto-implemented-
properties

[DataContract]
```

```
public class CreateOrderCommand
    : IRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;

    [DataMember]
    public string UserId { get; private set; }

    [DataMember]
    public string UserName { get; private set; }

    [DataMember]
    public string City { get; private set; }

    [DataMember]
    public string Street { get; private set; }

    [DataMember]
    public string State { get; private set; }

    [DataMember]
    public string Country { get; private set; }

    [DataMember]
    public string ZipCode { get; private set; }

    [DataMember]
    public string CardNumber { get; private set; }

    [DataMember]
    public string CardHolderName { get; private set; }

    [DataMember]
    public DateTime CardExpiration { get; private set; }

    [DataMember]
    public string CardSecurityNumber { get; private set; }

    [DataMember]
    public int CardTypeId { get; private set; }

    [DataMember]
    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;

    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }

    public CreateOrderCommand(List<BasketItem> basketItems, string userId,
string userName, string city, string street, string state, string country,
string zipcode,
        string cardNumber, string cardHolderName, DateTime cardExpiration,
        string cardSecurityNumber, int cardTypeId) : this()
```

```
    {
        _orderItems = basketItems.ToOrderItemsDTO().ToList();
        UserId = userId;
        UserName = userName;
        City = city;
        Street = street;
        State = state;
        Country = country;
        ZipCode = zipcode;
        CardNumber = cardNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
        CardSecurityNumber = cardSecurityNumber;
        CardTypeId = cardTypeId;
        CardExpiration = cardExpiration;
    }


    public class OrderItemDTO
    {
        public int ProductId { get; set; }

        public string ProductName { get; set; }

        public decimal UnitPrice { get; set; }

        public decimal Discount { get; set; }

        public int Units { get; set; }

        public string PictureUrl { get; set; }
    }
}
```

Básicamente, la clase de comando contiene todos los datos que se necesitan para llevar a cabo una transacción empresarial mediante los objetos de modelo de dominio. Por tanto, los comandos son simplemente las estructuras de datos que contienen datos de solo lectura y ningún comportamiento. El nombre del comando indica su propósito. En muchos lenguajes como C#, los comandos se representan como clases, pero no son verdaderas clases en el sentido real orientado a objetos.

Como una característica adicional, los comandos son inmutables, dado que el uso esperado es que el modelo de dominio los procese directamente. No deben cambiar durante su duración prevista. En una clase de C#, se puede lograr la inmutabilidad si no hay establecedores ni otros métodos que cambien el estado interno.

Tenga en cuenta que si quiere o espera que los comandos pasen por un proceso de serialización o deserialización, las propiedades deben tener un establecedor privado y el atributo `[DataMember]` (o `[JsonProperty]`). De lo contrario, el deserializador no podrá

reconstruir el objeto en el destino con los valores necesarios. También puede usar propiedades que realmente sean de solo lectura si la clase tiene un constructor con parámetros para todas las propiedades, con la convención de nomenclatura de camelCase habitual, y anotar el constructor como `[JsonConstructor]`. Sin embargo, esta opción requiere más código.

Por ejemplo, la clase de comando para crear un pedido probablemente sea similar en cuanto a los datos del pedido que se quiere crear, pero es probable que no se necesiten los mismos atributos. Por ejemplo, `CreateOrderCommand` no tiene un identificador de pedido, porque el pedido aún no se ha creado.

Muchas clases de comando pueden ser simples y requerir solo unos cuantos campos sobre algún estado que deba cambiarse. Ese sería el caso si solo se va a cambiar el estado de un pedido de "en proceso" a "pagado" o "enviado" con un comando similar al siguiente:

C#

```
[DataContract]
public class UpdateOrderStatusCommand
    :IRequest<bool>
{
    [DataMember]
    public string Status { get; private set; }

    [DataMember]
    public string OrderId { get; private set; }

    [DataMember]
    public string BuyerIdentityGuid { get; private set; }
}
```

Algunos desarrolladores separan los objetos de solicitud de interfaz de usuario de los DTO de comando, pero es solo una cuestión de preferencia. Es una separación tediosa sin demasiado valor añadido y los objetos tienen prácticamente la misma forma. Por ejemplo, en eShopOnContainers, algunos comandos proceden directamente del lado cliente.

Clase de controlador de comandos

Debe implementar una clase de controlador de comandos específica para cada comando. Ese es el funcionamiento del patrón y el lugar en el que se usarán el objeto de comando, los objetos de dominio y los objetos de repositorio de infraestructura. De hecho, el controlador de comandos es el núcleo del nivel de aplicación en lo que a

CQRS y DDD respecta. Sin embargo, toda la lógica del dominio debe incluirse en las clases de dominio, dentro de las raíces agregadas (entidades raíz), las entidades secundarias o los servicios de dominio, pero no en el controlador de comandos, que es una clase del nivel de aplicación.

La clase de controlador de comandos ofrece un punto de partida seguro en la forma de lograr el principio de responsabilidad única (SRP) mencionado en una sección anterior.

Un controlador de comandos recibe un comando y obtiene un resultado del agregado que se usa. El resultado debe ser la ejecución correcta del comando, o bien una excepción. En el caso de una excepción, el estado del sistema no debe cambiar.

Normalmente, el controlador de comandos realiza estos pasos:

- Recibe el objeto de comando, como un DTO (desde el mediador u otro objeto de infraestructura).
- Valida que el comando sea válido (si no lo hace el mediador).
- Crea una instancia de la instancia de raíz agregada que es el destino del comando actual.
- Ejecuta el método en la instancia de raíz agregada y obtiene los datos necesarios del comando.
- Conserva el nuevo estado del agregado en su base de datos relacionada. Esta última operación es la transacción real.

Normalmente, un controlador de comandos administra un único agregado controlado por su raíz agregada (la entidad raíz). Si varios agregados deben verse afectados por la recepción de un único comando, podría usar eventos de dominio para propagar los estados o las acciones entre varios agregados.

El aspecto importante aquí es que cuando se procesa un comando, toda la lógica del dominio debe incluirse en el modelo de dominio (los agregados), completamente encapsulada y lista para las pruebas unitarias. El controlador de comandos solo actúa como una manera de obtener el modelo de dominio de la base de datos y, como último paso, para indicar al nivel de infraestructura (los repositorios) que conserve los cambios cuando el modelo cambie. La ventaja de este enfoque es que se puede refactorizar la lógica del dominio en un modelo de dominio de comportamiento aislado, completamente encapsulado y enriquecido sin cambiar el código del nivel de aplicación o infraestructura, que forman el nivel de establecimiento (controladores de comandos, la API web, repositorios, etc.).

Cuando los controladores de comandos se complican, con demasiada lógica, se puede producir un problema en el código. Revíselos y, si encuentra lógica de dominio, refactorice el código para mover ese comportamiento de dominio a los métodos de los objetos de dominio (la raíz agregada y la entidad secundaria).

Como ejemplo de clase de controlador de comandos, en el código siguiente se muestra la misma clase `CreateOrderCommandHandler` que se vio al principio de este capítulo. En este caso, también se resalta el método Handle y las operaciones con los objetos de modelo de dominio y agregados.

C#

```
public class CreateOrderCommandHandler
        : IRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;
    private readonly IOrderingIntegrationEventService
_orderingIntegrationEventService;
    private readonly ILogger<CreateOrderCommandHandler> _logger;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService identityService,
        ILogger<CreateOrderCommandHandler> logger)
    {
        _orderRepository = orderRepository ?? throw new
ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new
ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new
ArgumentNullException(nameof(mediator));
        _orderingIntegrationEventService = orderingIntegrationEventService
?? throw new ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<bool> Handle(CreateOrderCommand message,
CancellationToken cancellationToken)
    {
        // Add Integration event to clean the basket
        var orderStartedIntegrationEvent = new
OrderStartedIntegrationEvent(message.UserId);
        await
_orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegratio
nEvent);

        // Add/Update the Buyer AggregateRoot
        // DDD patterns comment: Add child entities and value-objects
```

```
through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business
logic
        // make sure that consistency is preserved across the whole
aggregate
        var address = new Address(message.Street, message.City,
message.State, message.Country, message.ZipCode);
        var order = new Order(message.UserId, message.UserName, address,
message.CardTypeId, message.CardNumber, message.CardSecurityNumber,
message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName,
item.UnitPrice, item.Discount, item.PictureUrl, item.Units);
        }

        _logger.LogInformation("----- Creating Order - Order: {@Order}",
order);

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);
    }
}
```

Estos son los pasos adicionales que debe realizar un controlador de comandos:

- Usar los datos del comando para funcionar con los métodos y el comportamiento de la raíz agregada.
- Dentro de los objetos de dominio, generar eventos de dominio mientras se ejecuta la transacción, pero de forma transparente desde el punto de vista de un controlador de comandos.
- Si el resultado de la operación del agregado es correcto y una vez finalizada la transacción, generar eventos de integración. (Es posible que clases de infraestructura como repositorios también los generen).

Recursos adicionales

- **Mark Seemann. At the Boundaries, Applications are Not Object-Oriented (En los límites, las aplicaciones no están orientadas a objetos)** https://blog.ploeh.dk/2011/05/31/AttheBoundaries,ApplicationsareNotObject-Oriented/

- **Commands and events (Comandos y eventos)**
  https://cqrs.nu/faq/Command%20and%20Events
- **What does a command handler do?** (¿De qué se encarga un controlador de comandos?)
  https://cqrs.nu/faq/Command%20Handlers
- **Jimmy Bogard. Domain Command Patterns – Handlers (Patrones de comandos de dominio: controladores)**
  https://jimmybogard.com/domain-command-patterns-handlers/
- **Jimmy Bogard. Domain Command Patterns – Validation (Patrones de comandos de dominio: validación)**
  https://jimmybogard.com/domain-command-patterns-validation/

La canalización del proceso de comando: cómo desencadenar un controlador de comandos

La siguiente pregunta es cómo invocar un controlador de comandos. Se podría llamar manualmente desde cada controlador de ASP.NET Core relacionado. Pero ese enfoque sería demasiado acoplado y no es lo ideal.

Las otras dos opciones principales, que son las recomendadas, son estas:

- A través de un artefacto de patrón de mediador en memoria.
- Con una cola de mensajes asincrónicos, entre los controladores.

Uso del patrón de mediador (en memoria) en la canalización de comandos

Como se muestra en la figura 7-25, en un enfoque CQRS se usa un mediador inteligente, similar a un bus en memoria, que es lo suficientemente inteligente como para redirigir al controlador de comandos correcto según el tipo del comando o DTO que se recibe. Las flechas simples de color negro entre los componentes representan las dependencias entre los objetos (en muchos casos, insertados mediante DI) con sus interacciones relacionadas.

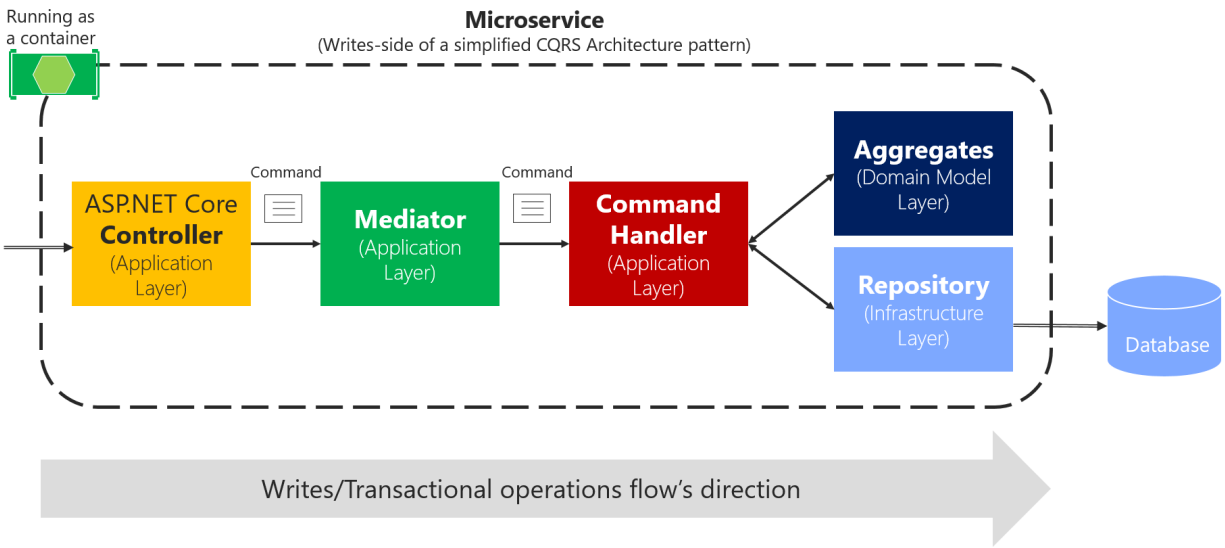


Figura 7-25. Uso del patrón de mediador en proceso en un único microservicio CQRS

En el diagrama anterior se muestra más detalle de la imagen 7-24: el controlador ASP.NET Core envía el comando a la canalización de comandos MediatR para que llegue al controlador adecuado.

El motivo por el que tiene sentido usar el patrón de mediador es que, en las aplicaciones empresariales, las solicitudes de procesamiento pueden resultar complicadas. Le interesa poder agregar un número abierto de cuestiones transversales como registro, validaciones, auditoría y seguridad. En estos casos, puede basarse en una canalización de mediador (vea Patrón de mediador) para proporcionar un medio para estos comportamientos adicionales o cuestiones transversales.

Un mediador es un objeto que encapsula el "cómo" de este proceso: coordina la ejecución en función del estado, la forma de invocar un controlador de comandos o la carga que se proporciona al controlador. Con un componente de mediador se pueden aplicar cuestiones transversales de forma centralizada y transparente aplicando elementos Decorator (o comportamientos de canalización desde MediatR 3). Para obtener más información, vea el Patrón de Decorator.

Los elementos Decorator y los comportamientos son similares a la Programación orientada a aspectos (AOP), solo se aplican a una canalización de proceso específica administrada por el componente de mediador. Los aspectos en AOP que implementan cuestiones transversales se aplican en función de *tejedores de aspectos* que se insertan en tiempo de compilación o en función de la intercepción de llamadas de objeto. En ocasiones, se dice que ambos enfoques típicos de AOP funcionan "de forma mágica", porque no es fácil ver cómo realiza AOP su trabajo. Cuando se trabaja con problemas graves o errores, AOP puede ser difícil de depurar. Por otro lado, estos elementos Decorator o comportamientos son explícitos y solo se aplican en el contexto del mediador, por lo que la depuración es mucho más sencilla y predecible.

Por ejemplo, el microservicio de pedidos de eShopOnContainers tiene implementados dos comportamientos de ejemplo, las clases LogBehavior y ValidatorBehavior. En esta sección se explica la implementación de los comportamientos mostrando cómo eShopOnContainers usa los comportamientos de MediatR.

Uso de colas de mensajes (fuera de proceso) en la canalización del comando

Otra opción consiste en usar mensajes asincrónicos basados en agentes o colas de mensajes, como se muestra en la figura 7-26. Esa opción también se podría combinar con el componente de mediador justo antes del controlador de comandos.

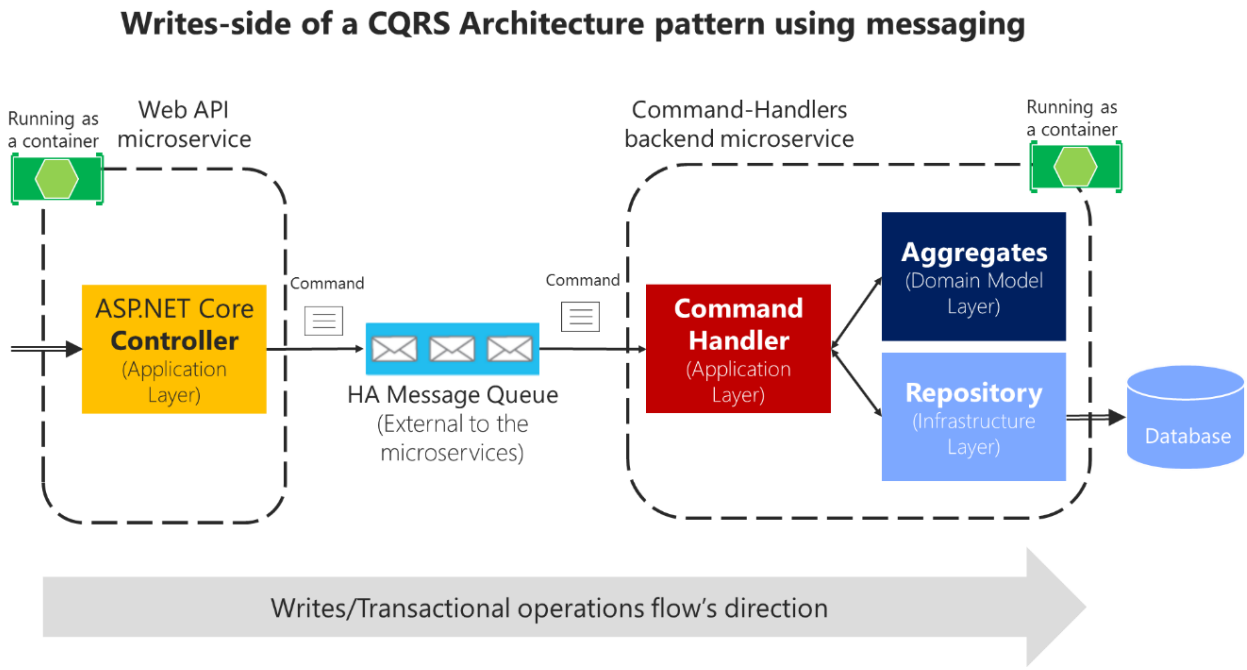


Figura 7-26. Uso de colas de mensajes (comunicación fuera de proceso y entre procesos) con comandos CQRS

La canalización del comando también puede controlarse mediante una cola de mensajes de alta disponibilidad para entregar los comandos en el controlador adecuado. El uso de colas de mensajes para aceptar los comandos puede complicar más la canalización del comando, ya que probablemente sea necesario dividir la canalización en dos procesos conectados a través de la cola de mensajes externos. Pero se debe usar si hay que ofrecer mayor escalabilidad y rendimiento según la mensajería asincrónica. Téngalo en cuenta en el caso de la figura 7-26, donde el controlador simplemente envía el mensaje de comando a la cola y vuelve. Después, los controladores de comandos procesan los mensajes a su propio ritmo. Esa es una gran ventaja de las colas: la cola de mensajes puede actuar como un búfer en casos en que se necesita hiperescalabilidad (por ejemplo, para existencias o cualquier otro escenario con un gran volumen de datos de entrada).

En cambio, debido a la naturaleza asincrónica de las colas de mensajes, debe saber cómo comunicar a la aplicación cliente si el proceso del comando se ha realizado correctamente o no. Como norma, nunca debería usar comandos "Fire and Forget" (dispare y olvídese). Cada aplicación empresarial necesita saber si un comando se ha procesado correctamente, o al menos se ha validado y aceptado.

De este modo, la capacidad de responder al cliente después de validar un mensaje de comando que se envió a una cola asincrónica agrega complejidad al sistema, en comparación con un proceso de comando en proceso que devuelve el resultado de la operación después de ejecutar la transacción. Mediante las colas, es posible que tenga que devolver el resultado del proceso de comando a través de otros mensajes de resultado de la operación, lo que requiere componentes adicionales y comunicación personalizada en el sistema.

Además, los comandos asincrónicos son unidireccionales, lo que es posible que en muchos casos no sea necesario, tal y como se explica en el siguiente e interesante intercambio entre Burtsev Alexey y Greg Young en una conversación en línea:

> [Burtsev Alexey] Veo mucho código en el que la gente usa el control de comandos asincrónicos o la mensajería de comandos unidireccionales sin ningún motivo para hacerlo (no están realizando una operación extensa, no ejecutan código asincrónico externo, ni siquiera cruzan los límites entre aplicaciones para usar bus de mensajes). ¿Por qué agregan esta complejidad innecesaria? Y en realidad, hasta ahora no he visto ningún ejemplo de código CQRS con controladores de comandos de bloqueo, aunque funcionaría correctamente en la mayoría de los casos.
>
> [Greg Young] [...] un comando asincrónico no existe; en realidad es otro evento. Si tengo que aceptar lo que se me envía y generar un evento si no estoy de acuerdo, ya no se me está pidiendo que haga algo [es decir, no es un comando]. Se me está diciendo que se ha realizado algo. Al principio puede parecer una pequeña diferencia, pero tiene muchas implicaciones.

Los comandos asincrónicos aumentan considerablemente la complejidad de un sistema, porque no hay ninguna manera sencilla de indicar los errores. Por tanto, los comandos asincrónicos no son recomendables a no ser que se necesiten requisitos de escalado o en casos especiales de comunicación de microservicios internos a través de mensajería. En esos casos, se debe diseñar un sistema independiente de informes y recuperación de errores del sistema.

En la versión inicial de eShopOnContainers, decidimos usar el procesamiento de comandos sincrónicos, iniciados desde solicitudes HTTP y controlados por el patrón de

mediador. Eso permite devolver con facilidad si el proceso se ha realizado correctamente o no, como en la implementación CreateOrderCommandHandler.

En cualquier caso, debe ser una decisión basada en los requisitos empresariales de la aplicación o el microservicio.

Implementación de la canalización del proceso de comando con un patrón de mediador (MediatR)

Como implementación de ejemplo, en esta guía se propone el uso de la canalización de proceso basada en el patrón de mediador para controlar la ingesta de comandos y enrutarlos, en memoria, a los controladores de comandos correctos. En la guía también se propone la aplicación de comportamientos para separar las cuestiones transversales.

Para la implementación en .NET, hay varias bibliotecas de código abierto disponibles que implementan el patrón de mediador. En esta guía se usa la biblioteca de código abierto MediatR (creada por Jimmy Bogard), pero puede usar otro enfoque. MediatR es una biblioteca pequeña y simple que permite procesar mensajes en memoria como un comando, mientras se aplican elementos Decorator o comportamientos.

El uso del patrón de mediador ayuda a reducir el acoplamiento y aislar los problemas del trabajo solicitado, mientras se conecta automáticamente al controlador que lleva a cabo ese trabajo, en este caso, a controladores de comandos.

En la revisión de esta guía, Jimmy Bogard explica otra buena razón para usar el patrón de mediador:

> Creo que aquí valdría la pena mencionar las pruebas: proporcionan una ventana coherente al comportamiento del sistema. Solicitud de entrada, respuesta de salida. Hemos comprobado que es un aspecto muy valioso a la hora de generar pruebas que se comporten de forma coherente.

En primer lugar, veremos un controlador WebAPI de ejemplo donde se usaría realmente el objeto de mediador. Si no se usara el objeto de mediador, sería necesario insertar todas las dependencias para ese controlador, elementos como un objeto de registrador y otros. Por tanto, el constructor sería complicado. Por otra parte, si se usa el objeto de mediador, el constructor del controlador puede ser mucho más sencillo, con solo algunas dependencias en lugar de muchas si hubiera una por cada operación transversal, como en el ejemplo siguiente:

```C#
public class MyMicroserviceController : Controller
{
    public MyMicroserviceController(IMediator mediator,
                                    IMyMicroserviceQueries
microserviceQueries)
    {
        // ...
    }
}
```

Se puede ver que el mediador proporciona un constructor de controlador de API web limpio y eficiente. Además, dentro de los métodos de controlador, el código para enviar un comando al objeto de mediador es prácticamente una línea:

```C#
[Route("new")]
[HttpPost]
public async Task<IActionResult>
ExecuteBusinessOperation([FromBody]RunOpCommand

runOperationCommand)
{
    var commandResult = await _mediator.SendAsync(runOperationCommand);

    return commandResult ? (IActionResult)Ok() :
(IActionResult)BadRequest();
}
```

Implementación de comandos idempotentes

En **eShopOnContainers**, un ejemplo más avanzado que el anterior es el envío de un objeto CreateOrderCommand desde el microservicio Ordering. Pero como el proceso empresarial Ordering es un poco más complejo y, en nuestro caso, se inicia realmente en el microservicio Basket, esta acción de enviar el objeto CreateOrderCommand se realiza desde un controlador de eventos de integración denominado UserCheckoutAcceptedIntegrationEventHandler, en lugar de un controlador WebAPI sencillo al que se llama desde la aplicación cliente, como ocurre en el ejemplo anterior más sencillo.

Pero la acción de enviar el comando a MediatR es bastante similar, como se muestra en el código siguiente.

```C#
```

```
var createOrderCommand = new CreateOrderCommand(eventMsg.Basket.Items,
                                                eventMsg.UserId,
eventMsg.City,
                                                eventMsg.Street,
eventMsg.State,
                                                eventMsg.Country,
eventMsg.ZipCode,
                                                eventMsg.CardNumber,
                                                eventMsg.CardHolderName,
                                                eventMsg.CardExpiration,
                                                eventMsg.CardSecurityNumber,
                                                eventMsg.CardTypeId);

var requestCreateOrder = new IdentifiedCommand<CreateOrderCommand,bool>
(createOrderCommand,

eventMsg.RequestId);
result = await _mediator.Send(requestCreateOrder);
```

Sin embargo, este caso también es ligeramente más avanzado porque también se implementan comandos idempotentes. El proceso CreateOrderCommand debe ser idempotente, por lo que si el mismo mensaje procede duplicado a través de la red, por cualquier motivo, como un reintento, el mismo pedido se procesará una sola vez.

Esto se implementa mediante la encapsulación del comando de negocio (en este caso, CreateOrderCommand) y su inserción en un IdentifiedCommand genérico, cuyo seguimiento se realiza a través de un identificador de todos los mensajes que lleguen a través de la red que tienen que ser idempotentes.

En el código siguiente, puede ver que el IdentifiedCommand no es más que un DTO con un identificador junto con el objeto de comando de negocio insertado.

C#

```
public class IdentifiedCommand<T, R> : IRequest<R>
    where T : IRequest<R>
{
    public T Command { get; }
    public Guid Id { get; }
    public IdentifiedCommand(T command, Guid id)
    {
        Command = command;
        Id = id;
    }
}
```

Después, el CommandHandler para el IdentifiedCommand denominado IdentifiedCommandHandler.cs básicamente comprobará si el identificador que

procede como parte del mensaje ya existe en una tabla. Si ya existe, ese comando no se volverá a procesar, por lo que se comporta como un comando idempotente. Ese código de infraestructura se ejecuta mediante la llamada al método `_requestManager.ExistAsync` siguiente.

C#

```
// IdentifiedCommandHandler.cs
public class IdentifiedCommandHandler<T, R> :
IRequestHandler<IdentifiedCommand<T, R>, R>
        where T : IRequest<R>
{
    private readonly IMediator _mediator;
    private readonly IRequestManager _requestManager;
    private readonly ILogger<IdentifiedCommandHandler<T, R>> _logger;

    public IdentifiedCommandHandler(
        IMediator mediator,
        IRequestManager requestManager,
        ILogger<IdentifiedCommandHandler<T, R>> logger)
    {
        _mediator = mediator;
        _requestManager = requestManager;
        _logger = logger ?? throw new
System.ArgumentNullException(nameof(logger));
    }

    /// <summary>
    /// Creates the result value to return if a previous request was found
    /// </summary>
    /// <returns></returns>
    protected virtual R CreateResultForDuplicateRequest()
    {
        return default(R);
    }

    /// <summary>
    /// This method handles the command. It just ensures that no other
request exists with the same ID, and if this is the case
    /// just enqueues the original inner command.
    /// </summary>
    /// <param name="message">IdentifiedCommand which contains both original
command & request ID</param>
    /// <returns>Return value of inner command or default value if request
same ID was found</returns>
    public async Task<R> Handle(IdentifiedCommand<T, R> message,
CancellationToken cancellationToken)
    {
        var alreadyExists = await _requestManager.ExistAsync(message.Id);
        if (alreadyExists)
        {
            return CreateResultForDuplicateRequest();
        }
```

```
        else
        {
            await _requestManager.CreateRequestForCommandAsync<T>
(message.Id);
            try
            {
                var command = message.Command;
                var commandName = command.GetGenericTypeName();
                var idProperty = string.Empty;
                var commandId = string.Empty;

                switch (command)
                {
                    case CreateOrderCommand createOrderCommand:
                        idProperty = nameof(createOrderCommand.UserId);
                        commandId = createOrderCommand.UserId;
                        break;

                    case CancelOrderCommand cancelOrderCommand:
                        idProperty = nameof(cancelOrderCommand.OrderNumber);
                        commandId = $"{cancelOrderCommand.OrderNumber}";
                        break;

                    case ShipOrderCommand shipOrderCommand:
                        idProperty = nameof(shipOrderCommand.OrderNumber);
                        commandId = $"{shipOrderCommand.OrderNumber}";
                        break;

                    default:
                        idProperty = "Id?";
                        commandId = "n/a";
                        break;
                }

                _logger.LogInformation(
                    "----- Sending command: {CommandName} - {IdProperty}:
{CommandId} ({@Command})",
                    commandName,
                    idProperty,
                    commandId,
                    command);

                // Send the embedded business command to mediator so it runs
its related CommandHandler
                var result = await _mediator.Send(command,
cancellationToken);

                _logger.LogInformation(
                    "----- Command result: {@Result} - {CommandName} -
{IdProperty}: {CommandId} ({@Command})",
                    result,
                    commandName,
                    idProperty,
                    commandId,
                    command);
```

```
                return result;
            }
            catch
            {
                return default(R);
            }
        }
    }
}
```

Dado que IdentifiedCommand actúa como sobre de un comando de negocios, cuando el comando de negocios se debe procesar porque no es un identificador repetido, toma ese comando de negocios interno y lo vuelve a enviar al mediador, como se muestra en la última parte del código anterior al ejecutar `_mediator.Send(message.Command)` desde IdentifiedCommandHandler.cs.

Al hacerlo, se vincula y ejecuta el controlador de comandos de negocios, en este caso, CreateOrderCommandHandler, que ejecuta transacciones con la base de datos Ordering, como se muestra en el código siguiente.

C#

```
// CreateOrderCommandHandler.cs
public class CreateOrderCommandHandler
        : IRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;
    private readonly IOrderingIntegrationEventService
_orderingIntegrationEventService;
    private readonly ILogger<CreateOrderCommandHandler> _logger;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService identityService,
        ILogger<CreateOrderCommandHandler> logger)
    {
        _orderRepository = orderRepository ?? throw new
ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new
ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new
ArgumentNullException(nameof(mediator));
        _orderingIntegrationEventService = orderingIntegrationEventService
?? throw new ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }
```

```
    public async Task<bool> Handle(CreateOrderCommand message,
CancellationToken cancellationToken)
    {
        // Add Integration event to clean the basket
        var orderStartedIntegrationEvent = new
OrderStartedIntegrationEvent(message.UserId);
        await
_orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegratio
nEvent);

        // Add/Update the Buyer AggregateRoot
        // DDD patterns comment: Add child entities and value-objects
through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business
logic
        // make sure that consistency is preserved across the whole
aggregate
        var address = new Address(message.Street, message.City,
message.State, message.Country, message.ZipCode);
        var order = new Order(message.UserId, message.UserName, address,
message.CardTypeId, message.CardNumber, message.CardSecurityNumber,
message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName,
item.UnitPrice, item.Discount, item.PictureUrl, item.Units);
        }

        _logger.LogInformation("----- Creating Order - Order: {@Order}",
order);

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);
    }
}
```

Registro de los tipos usados por MediatR

Para que MediatR sea consciente de las clases de controlador de comandos, debe registrar las clases de mediador y las de controlador de comandos en el contenedor de IoC. De forma predeterminada, MediatR usa Autofac como el contenedor de IoC, pero también se puede usar el contenedor de IoC integrado de ASP.NET Core o cualquier otro contenedor compatible con MediatR.

En el código siguiente se muestra cómo registrar los tipos y comandos del mediador al usar módulos de Autofac.

```C#
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {

builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IRequestHandler)
        // in assembly holding the Commands

builder.RegisterAssemblyTypes(typeof(CreateOrderCommand).GetTypeInfo().Assem
bly)
                .AsClosedTypesOf(typeof(IRequestHandler<,>));
        // Other types registration
        //...
    }
}
```

Aquí es donde "ocurre la magia" con MediatR.

Como cada controlador de comandos implementa la interfaz de `IRequestHandler<T>` genérica, cuando se registran los ensamblados mediante el método `RegisteredAssemblyTypes`, todos los tipos marcados como `IRequestHandler` también se registran con sus `Commands`. Por ejemplo:

```C#
public class CreateOrderCommandHandler
  : IRequestHandler<CreateOrderCommand, bool>
{
```

Ese es el código que pone en correlación los comandos y los controladores de comandos. El controlador es simplemente una clase, pero hereda de `RequestHandler<T>`, donde T es el tipo de comando, y MediatR se asegura de que se invoque con la carga correcta (el comando).

Aplicación de cuestiones transversales al procesar comandos con los comportamientos de MediatR

Hay otro aspecto: la capacidad de aplicar cuestiones transversales a la canalización de mediador. También puede ver al final del código del módulo de registro de Autofac

cómo registra un tipo de comportamiento, en concreto una clase LoggingBehavior personalizada y una clase ValidatorBehavior. Pero también se podrían agregar otros comportamientos personalizados.

C#

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {

builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(

typeof(CreateOrderCommand).GetTypeInfo().Assembly).

AsClosedTypesOf(typeof(IRequestHandler<,>));
        // Other types registration
        //...
        builder.RegisterGeneric(typeof(LoggingBehavior<,>)).

As(typeof(IPipelineBehavior<,>));
        builder.RegisterGeneric(typeof(ValidatorBehavior<,>)).

As(typeof(IPipelineBehavior<,>));
    }
}
```

Esa clase LoggingBehavior se puede implementar como el código siguiente, que registra información sobre el controlador de comandos que se está ejecutando y si se ha realizado correctamente o no.

C#

```
public class LoggingBehavior<TRequest, TResponse>
         : IPipelineBehavior<TRequest, TResponse>
{
    private readonly ILogger<LoggingBehavior<TRequest, TResponse>> _logger;
    public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>>
logger) =>
                                                                    _logger =
logger;

    public async Task<TResponse> Handle(TRequest request,
                                        RequestHandlerDelegate<TResponse>
next)
    {
```

```
        _logger.LogInformation($"Handling {typeof(TRequest).Name}");
        var response = await next();
        _logger.LogInformation($"Handled {typeof(TResponse).Name}");
        return response;
    }
}
```

Con la simple implementación de esta clase de comportamiento y su registro en la canalización (en el MediatorModule anterior), todos los comandos que se procesan a través de MediatR registrarán información sobre la ejecución.

El microservicio de pedidos de eShopOnContainers también aplica un segundo comportamiento para validaciones básicas, la clase ValidatorBehavior que se basa en la biblioteca FluentValidation, como se muestra en el código siguiente:

C#

```
public class ValidatorBehavior<TRequest, TResponse>
         : IPipelineBehavior<TRequest, TResponse>
{
    private readonly IValidator<TRequest>[] _validators;
    public ValidatorBehavior(IValidator<TRequest>[] validators) =>
                                                        _validators =
validators;

    public async Task<TResponse> Handle(TRequest request,
                                        RequestHandlerDelegate<TResponse>
next)
    {
        var failures = _validators
            .Select(v => v.Validate(request))
            .SelectMany(result => result.Errors)
            .Where(error => error != null)
            .ToList();

        if (failures.Any())
        {
            throw new OrderingDomainException(
                $"Command Validation Errors for type
{typeof(TRequest).Name}",
                        new ValidationException("Validation exception",
failures));
        }

        var response = await next();
        return response;
    }
}
```

Aquí, el comportamiento está generando una excepción si se produce un error de validación, pero también podría devolver un objeto de resultado, que contiene el resultado del comando si se realiza correctamente o la validación de mensajes en caso de que no lo hiciese. Esto probablemente facilitaría mostrar los resultados de validación al usuario.

Después, en función de la biblioteca FluentValidation, se crea la validación de los datos pasados con CreateOrderCommand, como se muestra en el código siguiente:

C#

```
public class CreateOrderCommandValidator :
AbstractValidator<CreateOrderCommand>
{
    public CreateOrderCommandValidator()
    {
        RuleFor(command => command.City).NotEmpty();
        RuleFor(command => command.Street).NotEmpty();
        RuleFor(command => command.State).NotEmpty();
        RuleFor(command => command.Country).NotEmpty();
        RuleFor(command => command.ZipCode).NotEmpty();
        RuleFor(command => command.CardNumber).NotEmpty().Length(12, 19);
        RuleFor(command => command.CardHolderName).NotEmpty();
        RuleFor(command =>
command.CardExpiration).NotEmpty().Must(BeValidExpirationDate).WithMessage("
Please specify a valid card expiration date");
        RuleFor(command => command.CardSecurityNumber).NotEmpty().Length(3);
        RuleFor(command => command.CardTypeId).NotEmpty();
        RuleFor(command =>
command.OrderItems).Must(ContainOrderItems).WithMessage("No order items
found");
    }

    private bool BeValidExpirationDate(DateTime dateTime)
    {
        return dateTime >= DateTime.UtcNow;
    }

    private bool ContainOrderItems(IEnumerable<OrderItemDTO> orderItems)
    {
        return orderItems.Any();
    }
}
```

Podría crear validaciones adicionales. Se trata de una forma muy limpia y elegante de implementar las validaciones de comandos.

De forma similar, podría implementar otros comportamientos para aspectos adicionales o cuestiones transversales que quiera aplicar a los comandos cuando los administre.

Recursos adicionales

El patrón de mediador

- **Patrón de mediador**
  https://en.wikipedia.org/wiki/Mediator_pattern

El patrón Decorator

- **Patrón Decorator**
  https://en.wikipedia.org/wiki/Decorator_pattern

MediatR (Jimmy Bogard)

- **MediatR.** Repositorio de GitHub.
  https://github.com/jbogard/MediatR
- **CQRS with MediatR and AutoMapper (CQRS con MediatR y AutoMapper)**
  https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/
- **Put your controllers on a diet: POSTs and commands** (Poner los controladores a dieta: POST y comandos).
  https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/
- **Tackling cross-cutting concerns with a mediator pipeline (Abordar cuestiones transversales con una canalización de mediador)**
  https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/
- **CQRS and REST: the perfect match (CQRS y REST: la combinación perfecta)**
  https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/
- **MediatR Pipeline Examples (Ejemplos de canalización de MediatR)**
  https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/
- **Vertical Slice Test Fixtures for MediatR and ASP.NET Core (Accesorios de prueba de segmentos verticales para MediatR y ASP.NET Core)**
  https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/

- **MediatR Extensions for Microsoft Dependency Injection Released (Extensiones de MediatR para el lanzamiento de inserciones de dependencias de Microsoft)** https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/

Validación fluida

- **Jeremy Skinner. Validación fluida.** Repositorio de GitHub. https://github.com/JeremySkinner/FluentValidation

Anterior Siguiente

Implementación de aplicaciones resistentes

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Los microservicios y las aplicaciones basadas en la nube deben adoptar los errores parciales que se producirán finalmente. Debe diseñar la aplicación para que sea resistente a esos errores parciales.

Resistencia es la capacidad de recuperarse de los errores y seguir funcionando. No se trata de evitar errores, sino de aceptar el hecho de que se producirán errores y de responder a ellos de una manera que evite el tiempo de inactividad o la pérdida de datos. El objetivo de la resistencia es devolver la aplicación a un estado de funcionamiento completo después de un error.

Es lo suficientemente difícil diseñar e implementar una aplicación basada en microservicios. Pero también debe mantener la aplicación funcionando en un entorno donde algún tipo de fallo es inevitable. Por lo tanto, la aplicación debe ser resistente. Debe diseñarse para hacer frente a errores parciales, como interrupciones de red o nodos o máquinas virtuales que se bloquean en la nube. Incluso los microservicios (contenedores) que se mueven a un nodo diferente dentro de un clúster pueden provocar errores cortos intermitentes dentro de la aplicación.

Los muchos componentes individuales de la aplicación también deben incorporar características de supervisión de estado. Siguiendo las instrucciones de este capítulo, puede crear una aplicación que funcione sin problemas a pesar del tiempo de inactividad transitorio o

de los problemas normales que se producen en implementaciones complejas y basadas en la nube.

> **Importante**
>
> eShopOnContainer ha estado usando la biblioteca de Polly para implementar resistencia mediante clientes con tipo hasta la versión 3.0.0. A partir de la versión 3.0.0, la resistencia de las llamadas HTTP se implementa mediante una malla Linkerd, que controla los reintentos de forma transparente y configurable dentro de un clúster de Kubernetes, sin tener que controlar esos problemas en el código.
>
> La biblioteca Polly todavía se usa para agregar resistencia a las conexiones de base de datos, especialmente al iniciar los servicios.

> **Advertencia**
>
> Todos los ejemplos de código e imágenes de esta sección eran válidos antes de usar Linkerd y no se actualizan para reflejar el código real actual. Por lo tanto, tienen sentido en el contexto de esta sección.

Siguiente anterior

Controlar errores parciales

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

En sistemas distribuidos, como las aplicaciones basadas en microservicios, existe un riesgo de error parcial. Por ejemplo, un único microservicio o contenedor puede producir un error o no estar disponible para responder durante un breve tiempo, o una sola máquina virtual o servidor puede bloquearse. Dado que los clientes y los servicios son procesos independientes, es posible que un servicio no pueda responder de forma oportuna a la solicitud de un cliente. Es posible que el servicio se sobrecargue y responda muy lentamente a las solicitudes o que simplemente no sea accesible durante un breve tiempo debido a problemas de red.

Por ejemplo, considere la página de detalles del pedido de la aplicación de ejemplo eShopOnContainers. Si el microservicio de ordenación no responde cuando el usuario intenta enviar un pedido, una implementación incorrecta del proceso de cliente (la aplicación web MVC), por ejemplo, si el código de cliente usara RPCs sincrónicas sin tiempo de espera, bloquearía los subprocesos indefinidamente esperando una respuesta. Además de crear una mala experiencia del usuario, cada espera que no responde consume o bloquea un subproceso, y los subprocesos son extremadamente valiosos en aplicaciones altamente escalables. Si hay muchos subprocesos bloqueados, eventualmente el tiempo de ejecución de la aplicación puede quedarse sin subprocesos. En ese caso, la aplicación puede dejar de responder globalmente en lugar de simplemente no responder parcialmente, como se muestra en la figura 8-1.

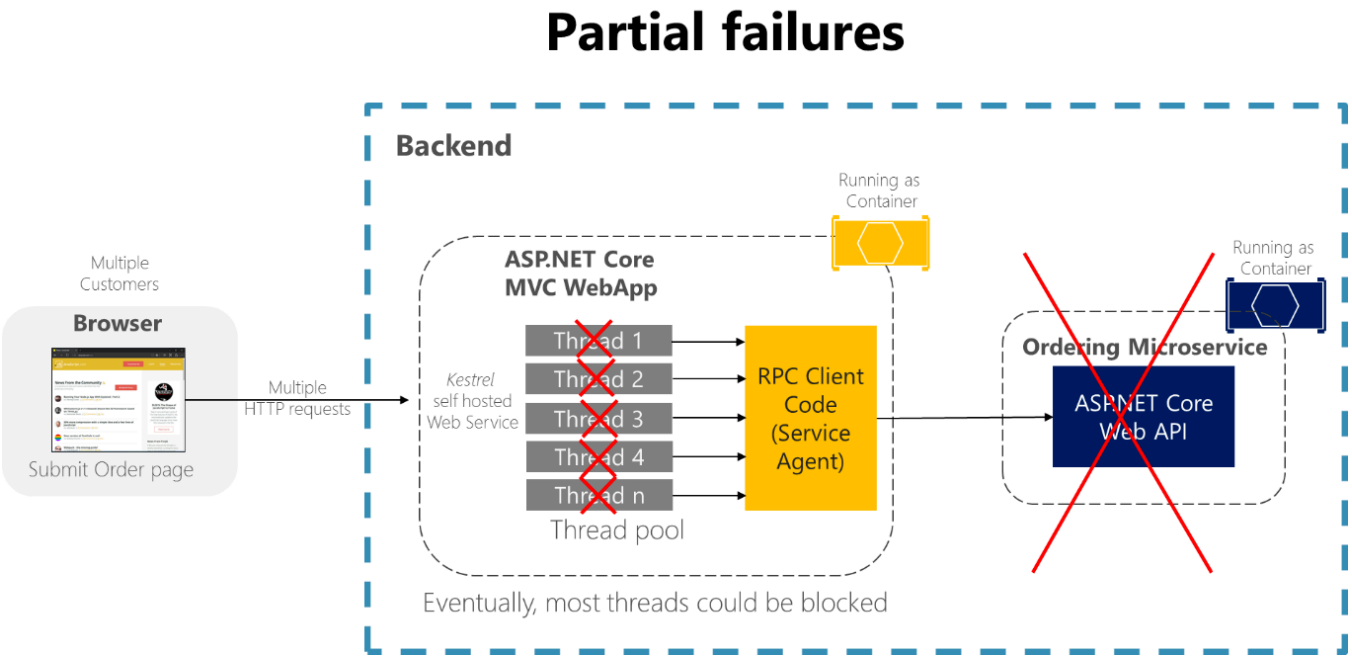


Figura 8-1. Errores parciales debido a dependencias que afectan a la disponibilidad del subproceso de servicio

En una aplicación basada en microservicios de gran tamaño, se puede amplificar cualquier error parcial, especialmente si la mayoría de la interacción interna de microservicios se basa en llamadas HTTP sincrónicas (que se considera un antipatrón). Piense en un sistema que recibe millones de llamadas entrantes al día. Si el sistema tiene un diseño incorrecto basado en cadenas largas de llamadas HTTP sincrónicas, estas llamadas entrantes podrían dar lugar a muchos más millones de llamadas salientes (supongamos una relación de 1:4) a docenas de microservicios internos como dependencias sincrónicas. Esta situación se muestra en la figura 8-2, especialmente la dependencia 3, que inicia una cadena, llamando a dependency #4, que luego llama a #5.

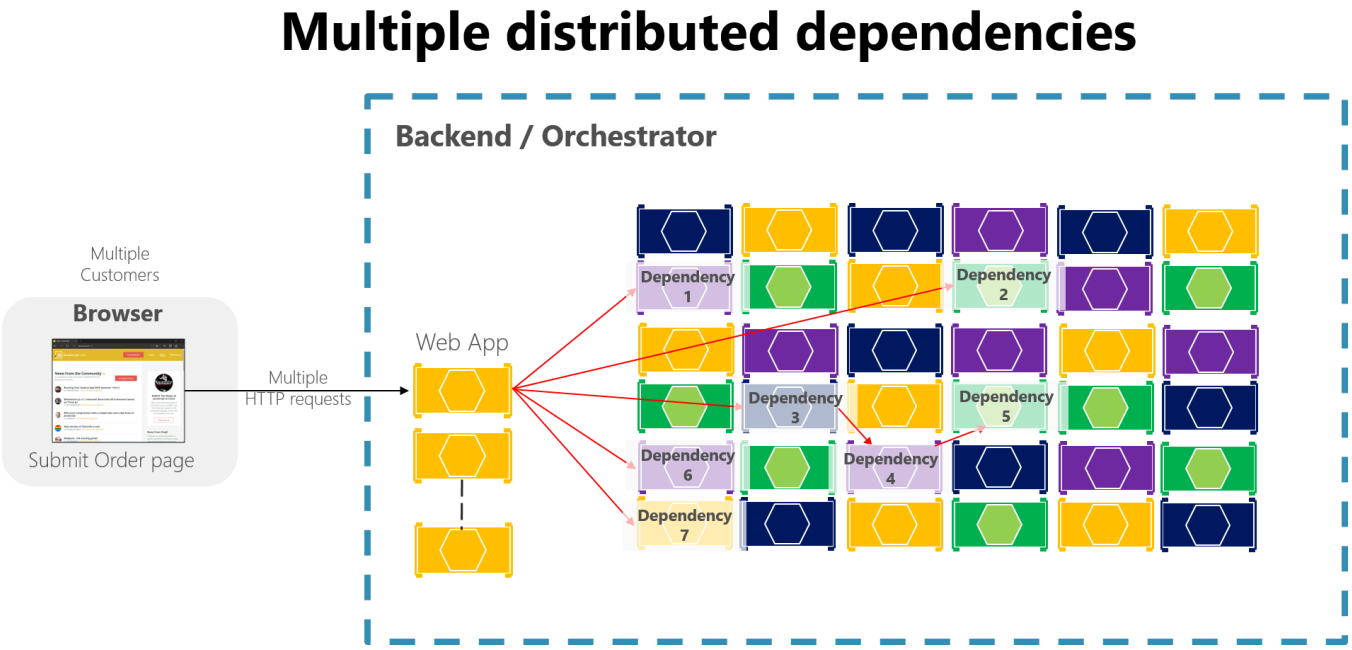


Figura 8-2. El impacto de tener un diseño incorrecto que incluye cadenas largas de solicitudes HTTP

Se garantiza un error intermitente en un sistema distribuido y basado en la nube, incluso si cada dependencia tiene una excelente disponibilidad. Es un hecho que debes tener en cuenta.

Si no diseña ni implementa técnicas para asegurar la tolerancia a errores, incluso se pueden magnificar los pequeños tiempos de inactividad. Por ejemplo, 50 dependencias cada una con 99,99% de disponibilidad daría como resultado varias horas de tiempo de inactividad cada mes debido a este efecto de onda. Cuando se produce un error en una dependencia de microservicio al controlar un gran volumen de solicitudes, ese error puede saturar rápidamente todos los subprocesos de solicitud disponibles en cada servicio y bloquear toda la aplicación.

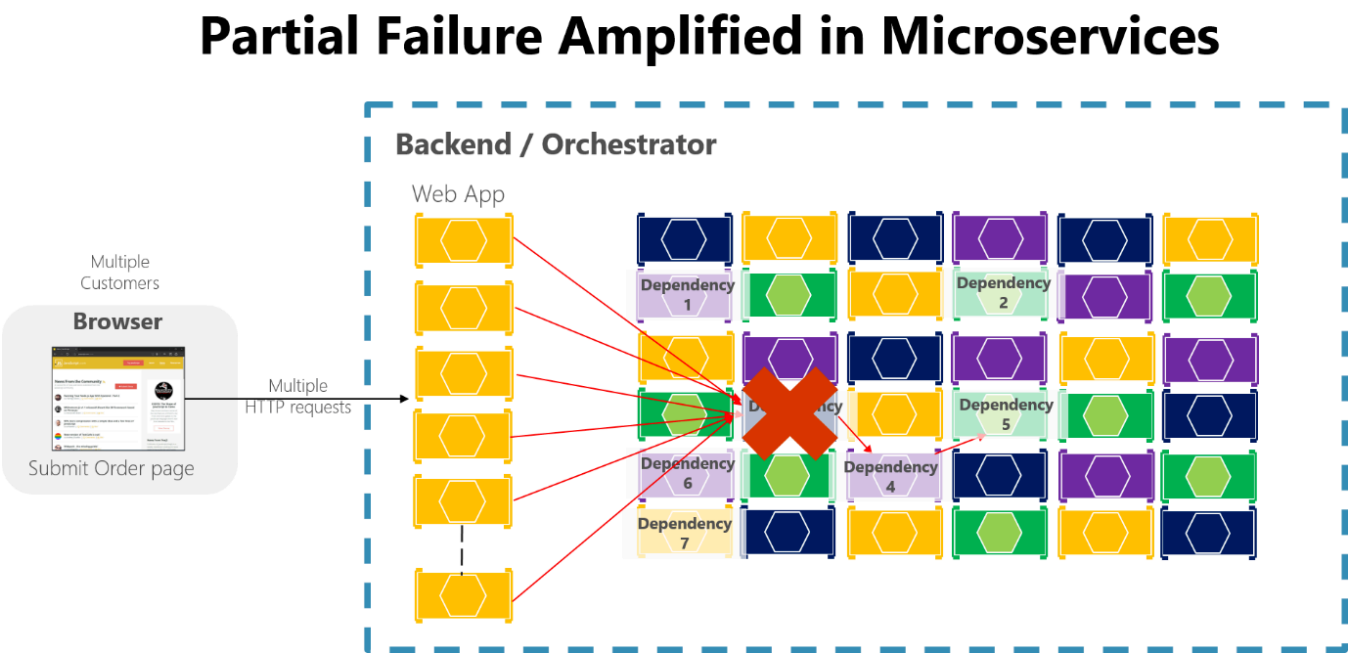


Figura 8-3. Error parcial amplificado por microservicios con cadenas largas de llamadas HTTP sincrónicas

Para minimizar este problema, en la sección Integración asincrónica de microservicios exige la autonomía del microservicio, esta guía le anima a usar la comunicación asincrónica entre los microservicios internos.

Además, es esencial diseñar los microservicios y las aplicaciones cliente para controlar errores parciales, es decir, crear microservicios resistentes y aplicaciones cliente.

Siguiente anterior

Estrategias para controlar errores parciales

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Para tratar los errores parciales, use una de las estrategias que se describen aquí.

Use la comunicación asincrónica (por ejemplo, comunicación basada en mensajes) en microservicios internos. Es muy recomendable no crear cadenas largas de llamadas HTTP sincrónicas en los microservicios internos, ya que ese diseño incorrecto se convertirá finalmente en la causa principal de interrupciones incorrectas. Por el contrario, excepto para las comunicaciones front-end entre las aplicaciones cliente y el primer nivel de microservicios o puertas de enlace de API específicas, se recomienda usar solo la comunicación asincrónica (basada en mensajes) una vez pasado el ciclo inicial de solicitud/respuesta, en los microservicios internos. La coherencia final y las arquitecturas controladas por eventos ayudarán a minimizar los efectos de onda. Estos enfoques aplican un mayor nivel de autonomía de microservicios y, por tanto, impiden el problema que se indica aquí.

Usar reintentos con retroceso exponencial. Esta técnica ayuda a evitar errores cortos e intermitentes realizando reintentos de llamada un número determinado de veces, en caso de que el servicio no estuviera disponible solo durante un breve tiempo. Esto puede producirse debido a problemas de red intermitentes o cuando un microservicio o contenedor se mueve a un nodo diferente de un clúster. Pero si estos intentos no se diseñan correctamente con interruptores, pueden agravar el efecto dominó e incluso pueden llegar a producir un ataque por denegación de servicio (DoS).

Solucionar los tiempos de expiración de red. En general, los clientes deben diseñarse para que no se bloqueen indefinidamente y para que usen siempre los tiempos de expiración cuando esperen una respuesta. El uso de tiempos de espera garantiza que los recursos nunca estén vinculados indefinidamente.

Utiliza el patrón Circuit Breaker. En este enfoque, el proceso de cliente realiza un seguimiento del número de solicitudes con error. Si la tasa de errores supera el límite establecido, se activa un "interruptor" para que los intentos adicionales fallen de inmediato. (Si se produce un error en un gran número de solicitudes, esto sugiere que el servicio no está disponible y que el envío de solicitudes no tiene sentido). Después de un período de tiempo de espera, el cliente debe intentarlo de nuevo y, si las nuevas solicitudes se realizan correctamente, cierre el disyuntor.

Proporcione alternativas. En este enfoque, el proceso de cliente realiza la lógica de reserva cuando se produce un error en una solicitud, como devolver datos almacenados en caché o un valor predeterminado. Se trata de un enfoque adecuado para las consultas y es más complejo para las actualizaciones o comandos.

Limite el número de solicitudes en cola. Los clientes también deben imponer un límite superior en el número de solicitudes pendientes que un microservicio de cliente puede enviar a un servicio determinado. Si se ha alcanzado el límite, es probable que no tenga sentido realizar solicitudes adicionales y esos intentos deben producir un error inmediatamente. En cuanto a la implementación, la directiva Aislamiento compartimentado de Polly se puede usar para cumplir este requisito. Este enfoque es básicamente una limitación en paralelo con SemaphoreSlim como implementación. También admite una "cola" fuera de la mampara. Puede perder proactivamente una carga excesiva incluso antes de la ejecución (por ejemplo, porque se considera que ha llegado al límite de su capacidad). Esto hace que su respuesta a determinados escenarios de error sea más rápida que la de un disyuntor, ya que este espera a los fallos. El objeto BulkheadPolicy de Polly expone hasta qué punto están llenos el espacio limitado por la mampara y la cola, y ofrece eventos sobre desbordamiento para que también se puedan utilizar para administrar un escalado horizontal automatizado.

Recursos adicionales

- **Patrones de resistencia**
  https://learn.microsoft.com/azure/architecture/framework/resiliency/reliability-patterns
- **Adición de resistencia y optimización del rendimiento**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591574(v=pandp.10)
- **Bulkhead (Mampara).** Repositorio de GitHub. Implementación con la directiva de Polly.
  https://github.com/App-vNext/Polly/wiki/Bulkhead

- **Diseño de aplicaciones resistentes de Azure**
  https://learn.microsoft.com/azure/architecture/framework/resiliency/app-design

- **Control de errores transitorios**
  https://learn.microsoft.com/azure/architecture/best-practices/transient-faults

Siguiente anterior

Implementar reintentos con retroceso exponencial

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Los reintentos con retroceso exponencial es una técnica que vuelve a intentar una operación, con un tiempo de espera exponencialmente creciente, hasta un número máximo de reintentos se ha alcanzado (el retroceso exponencial). Esta técnica adopta el hecho de que los recursos en la nube podrían no estar disponibles intermitentemente durante más de unos segundos por cualquier motivo. Por ejemplo, un orquestador podría mover un contenedor a otro nodo de un clúster para el equilibrio de carga. Durante ese tiempo, es posible que se produzca un error en algunas solicitudes. Otro ejemplo podría ser una base de datos como SQL Azure, donde una base de datos se puede mover a otro servidor para el equilibrio de carga, lo que hace que la base de datos no esté disponible durante unos segundos.

Existen muchos enfoques para implementar la lógica de reintentos con retroceso exponencial.

Siguiente anterior

Implementación de conexiones SQL resistentes de Entity Framework Core

Artículo • 15/10/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Para Azure SQL DB, Entity Framework (EF) Core ya proporciona la lógica de reintento y resistencia de conexión de base de datos interna. Pero debe habilitar la estrategia de ejecución de Entity Framework para cada conexión de DbContext si quiere tener conexiones resistentes de EF Core.

Por ejemplo, el código siguiente en el nivel de conexión de EF Core permite conexiones resistentes de SQL que se vuelven a intentar si se produce un error en la conexión.

C#

```
// Program.cs from any ASP.NET Core Web API
// Other code ...
builder.Services.AddDbContext<CatalogContext>(options =>
    {
        options.UseSqlServer(
            builder.Configuration["ConnectionString"],
            sqlServerOptionsAction: sqlOptions =>
            {
                sqlOptions.EnableRetryOnFailure(
                maxRetryCount: 10,
                maxRetryDelay: TimeSpan.FromSeconds(30),
                errorNumbersToAdd: null);
```

```
            });
    });
```

> **Importante**
>
> Microsoft recomienda usar el flujo de autenticación más seguro disponible. Si se conecta a Azure SQL, el método de autenticación recomendado es Identidades administradas para recursos de Azure.

Estrategias de ejecución y transacciones explícitas mediante BeginTransaction y varios DbContexts

Cuando se habilitan los reintentos en las conexiones de EF Core, cada operación que se realiza mediante EF Core se convierte en su propia operación que se puede reintentar. Cada consulta y cada llamada a `SaveChanges` se reintentará como una unidad si se produce un error transitorio.

Sin embargo, si su código inicia una transición con `BeginTransaction`, define su propio grupo de operaciones que deben tratarse como unidad. Todo el contenido de la transacción debe revertirse si se produce un error.

Si intenta ejecutar esa transacción cuando se usa una estrategia de ejecución de EF (directiva de reintentos) y llama a `SaveChanges` de varios DbContext, obtendrá una excepción como esta:

> System.InvalidOperationException: la estrategia de ejecución configurada "SqlServerRetryingExecutionStrategy" no es compatible con las transacciones que el usuario inicie. Use la estrategia de ejecución que devuelve "DbContext.Database.CreateExecutionStrategy()" para ejecutar todas las operaciones en la transacción como una unidad que se puede reintentar.

La solución consiste en invocar manualmente la estrategia de ejecución de EF con un delegado que representa a todos los elementos que se deben ejecutar. Si se produce un error transitorio, la estrategia de ejecución vuelve a invocar al delegado. Por ejemplo, el código siguiente muestra cómo se implementa en eShopOnContainers con dos DbContexts múltiples (_catalogContext y el IntegrationEventLogContext) al actualizar un producto y, después, guardar el objeto ProductPriceChangedIntegrationEvent, que debe usar un DbContext diferente.

C#

```
public async Task<IActionResult> UpdateProduct(
    [FromBody]CatalogItem productToUpdate)
{
    // Other code ...

    var oldPrice = catalogItem.Price;
    var raiseProductPriceChangedEvent = oldPrice != productToUpdate.Price;

    // Update current product
    catalogItem = productToUpdate;

    // Save product's data and publish integration event through the Event
Bus
    // if price has changed
    if (raiseProductPriceChangedEvent)
    {
        //Create Integration Event to be published through the Event Bus
        var priceChangedEvent = new ProductPriceChangedIntegrationEvent(
          catalogItem.Id, productToUpdate.Price, oldPrice);

       // Achieving atomicity between original Catalog database operation
and the
       // IntegrationEventLog thanks to a local transaction
       await
_catalogIntegrationEventService.SaveEventAndCatalogContextChangesAsync(
           priceChangedEvent);

       // Publish through the Event Bus and mark the saved event as
published
       await _catalogIntegrationEventService.PublishThroughEventBusAsync(
           priceChangedEvent);
    }
    // Just save the updated product because the Product's Price hasn't
changed.
    else
    {
        await _catalogContext.SaveChangesAsync();
    }
}
```

El primer contexto DbContext es `_catalogContext` y el segundo contexto `DbContext` está dentro del objeto `_catalogIntegrationEventService`. La acción Commit se realiza a través de todos los objetos `DbContext` mediante una estrategia de ejecución de EF.

Para lograr esta confirmación `DbContext` múltiple, el elemento `SaveEventAndCatalogContextChangesAsync` usa una clase `ResilientTransaction`, como se muestra en el siguiente código:

C#

```
public class CatalogIntegrationEventService :
ICatalogIntegrationEventService
{
    //…
    public async Task SaveEventAndCatalogContextChangesAsync(
        IntegrationEvent evt)
    {
        // Use of an EF Core resiliency strategy when using multiple
DbContexts
        // within an explicit BeginTransaction():
        // https://learn.microsoft.com/ef/core/miscellaneous/connection-
resiliency
        await ResilientTransaction.New(_catalogContext).ExecuteAsync(async
() =>
        {
            // Achieving atomicity between original catalog database
            // operation and the IntegrationEventLog thanks to a local
transaction
            await _catalogContext.SaveChangesAsync();
            await _eventLogService.SaveEventAsync(evt,

_catalogContext.Database.CurrentTransaction.GetDbTransaction());
        });
    }
}
```

El método `ResilientTransaction.ExecuteAsync` básicamente comienza una transacción desde el contexto `DbContext` pasado (`_catalogContext`) y, a continuación, hace que el servicio `EventLogService` use dicha transacción para guardar los cambios del contexto `IntegrationEventLogContext` y, después, confirma toda la transacción.

C#

```
public class ResilientTransaction
{
    private DbContext _context;
    private ResilientTransaction(DbContext context) =>
        _context = context ?? throw new
ArgumentNullException(nameof(context));

    public static ResilientTransaction New (DbContext context) =>
        new ResilientTransaction(context);

    public async Task ExecuteAsync(Func<Task> action)
    {
        // Use of an EF Core resiliency strategy when using multiple
DbContexts
        // within an explicit BeginTransaction():
        // https://learn.microsoft.com/ef/core/miscellaneous/connection-
resiliency
        var strategy = _context.Database.CreateExecutionStrategy();
```

```
        await strategy.ExecuteAsync(async () =>
        {
            await using var transaction = await
_context.Database.BeginTransactionAsync();
            await action();
            await transaction.CommitAsync();
        });
    }
}
```

Recursos adicionales

- **Connection Resiliency and Command Interception with EF in an ASP.NET MVC Application (Resistencia de la conexión e intercepción de comandos con EF en una aplicación de ASP.NET MVC)**
  https://learn.microsoft.com/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/connection-resiliency-and-command-interception-with-the-entity-framework-in-an-asp-net-mvc-application

- **Cesar de la Torre. Using Resilient Entity Framework Core SQL Connections and Transactions (Usar conexiones y transacciones SQL resistentes de Entity Framework Core)**
  https://devblogs.microsoft.com/cesardelatorre/using-resilient-entity-framework-core-sql-connections-and-transactions-retries-with-exponential-backoff/

Anterior Siguiente

Uso de IHttpClientFactory para implementar solicitudes HTTP resistentes

Artículo • 31/01/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

IHttpClientFactory es un contrato implementado por `DefaultHttpClientFactory`, una fábrica bien fundamentada disponible desde .NET Core 2.1 para crear instancias de HttpClient con el fin de usarlas en las aplicaciones.

Problemas con la clase HttpClient original disponible en .NET

La clase de HttpClient original y conocida se puede usar fácilmente, pero en algunos casos, muchos desarrolladores no lo usan correctamente.

Aunque esta clase implementa `IDisposable`, no se aconseja declarar y crear instancias de ella en una instrucción `using` porque, cuando el objeto `HttpClient` se desecha, el socket subyacente no se libera inmediatamente, lo que puede conducir a un problema de *agotamiento del socket*. Para obtener más información sobre este problema, consulte la entrada de blog Está usando HttpClient incorrecto y está desestabilizando el software.

Por lo tanto, `HttpClient` está pensado para ser instanciado una vez y reutilizarse durante toda la vida útil de una aplicación. Crear una instancia de una clase `HttpClient` para cada solicitud agotará el número de sockets disponibles bajo cargas pesadas. Ese problema generará errores `SocketException`. Los posibles enfoques para resolver ese problema se basan en la creación del objeto `HttpClient` como singleton o estático, como se explica en este artículo de Microsoft sobre el uso de HttpClient. Puede ser una buena solución para aplicaciones de consola de corta duración o similares, que se ejecutan varias veces al día.

Otro problema con el que se encuentran los desarrolladores es al utilizar una instancia compartida de `HttpClient` en procesos de ejecución prolongada. En una situación en la que se crean instancias del HttpClient como un singleton o un objeto estático, los cambios de DNS no se pueden controlar, tal y como se describe en esta incidencia del repositorio de GitHub sobre dotnet/runtime.

Sin embargo, el problema no está realmente con `HttpClient` por se, pero con el constructor predeterminado para HttpClient, ya que crea una nueva instancia concreta de HttpMessageHandler, que es la que tiene *problemas de agotamiento de sockets* y cambios de DNS mencionados anteriormente.

Para solucionar los problemas mencionados anteriormente y para que las instancias de `HttpClient` puedan administrarse, .NET Core 2.1 introdujo dos enfoques, uno de ellos se IHttpClientFactory. Se trata de una interfaz que se usa para configurar y crear instancias de `HttpClient` en una aplicación mediante la inserción de dependencias (DI). También proporciona extensiones para el middleware basado en Polly para aprovechar las ventajas de delegar controladores en HttpClient.

La alternativa es usar `SocketsHttpHandler` con `PooledConnectionLifetime` configurado. Este enfoque se aplica a instancias de larga duración , `static` o `HttpClient` de singleton. Para obtener más información sobre las distintas estrategias, vea Directrices de HttpClient para .NET.

Polly es una biblioteca de manejo de fallos transitorios que ayuda a los desarrolladores a agregar resiliencia a sus aplicaciones mediante el uso de algunas directivas predefinidas de forma fluida y segura para subprocesos.

Ventajas de usar IHttpClientFactory

La implementación actual de IHttpClientFactory, que también implementa IHttpMessageHandlerFactory, ofrece las siguientes ventajas:

- Proporciona una ubicación central para asignar nombres y configurar objetos de `HttpClient` lógicos. Por ejemplo, puede configurar un cliente (Agente de servicio) preconfigurado para acceder a un microservicio específico.
- Codifique el concepto de middleware saliente mediante la delegación de manejadores en `HttpClient` e implemente middleware basado en Polly para aprovechar las políticas de Polly y lograr resiliencia.
- `HttpClient` ya posee el concepto de controladores de delegación, que se pueden vincular entre sí para las solicitudes HTTP salientes. Los clientes HTTP se pueden registrar en la fábrica y se puede usar un controlador de Polly que permite utilizar directivas de Polly para el reintento, interruptores, etc.
- Administre la duración de HttpMessageHandler para evitar los problemas mencionados y los que se puedan producir al administrar las duraciones de `HttpClient` usted mismo.

Sugerencia

Las instancias de `HttpClient` insertadas mediante DI se pueden eliminar de forma segura, porque el elemento `HttpMessageHandler` asociado lo administra la fábrica. Las instancias de `HttpClient` insertadas son *transitorias* desde punto de vista de DI, mientras que las de `HttpMessageHandler` se pueden considerar como *con ámbito*. Las instancias de `HttpMessageHandler` tienen sus propios ámbitos de DI, **independientes** de los ámbitos de la aplicación (por ejemplo, ámbitos de solicitud de entrada de ASP.NET). Para obtener más información, vea Using HttpClientFactory in .NET.

Nota

La implementación de `IHttpClientFactory` (`DefaultHttpClientFactory`) está estrechamente vinculada a la implementación de DI en el paquete NuGet de `Microsoft.Extensions.DependencyInjection`. Si necesita usar `HttpClient` sin DI o con otras implementaciones de DI, considere la posibilidad de usar un elemento `static` o una singleton `HttpClient` con la configuración `PooledConnectionLifetime`. A fin de obtener más información, vea Directrices de HttpClient para .NET.

Varias maneras de usar IHttpClientFactory

Hay varias maneras de usar `IHttpClientFactory` en la aplicación:

- Uso básico
- Usar clientes con nombre.
- Usar clientes con tipo.
- Uso de clientes generados

Por motivos de brevedad, esta guía muestra la manera más estructurada de usar `IHttpClientFactory`, que consiste en utilizar clientes tipados (patrón de agente de servicios). Pero todas las opciones están documentadas e incluidas actualmente en este artículo que trata sobre el uso de HttpClientFactoryIHttpClientFactory.

> ⓘ **Nota**
>
> Si la aplicación requiere cookies, es posible que sea mejor evitar el uso de IHttpClientFactory en la aplicación. Para obtener formas alternativas de administrar clientes, consulte Directrices para usar clientes HTTP.

Cómo usar Clientes tipados con IHttpClientFactory

Así pues, ¿qué es un "Cliente con tipo"? Es solo un elemento `HttpClient` que está preconfigurado para un uso específico. Esta configuración puede incluir valores específicos, como el servidor base, los encabezados HTTP o los tiempos de espera.

En el diagrama siguiente se muestra cómo se usan los clientes con tipo con `IHttpClientFactory`:

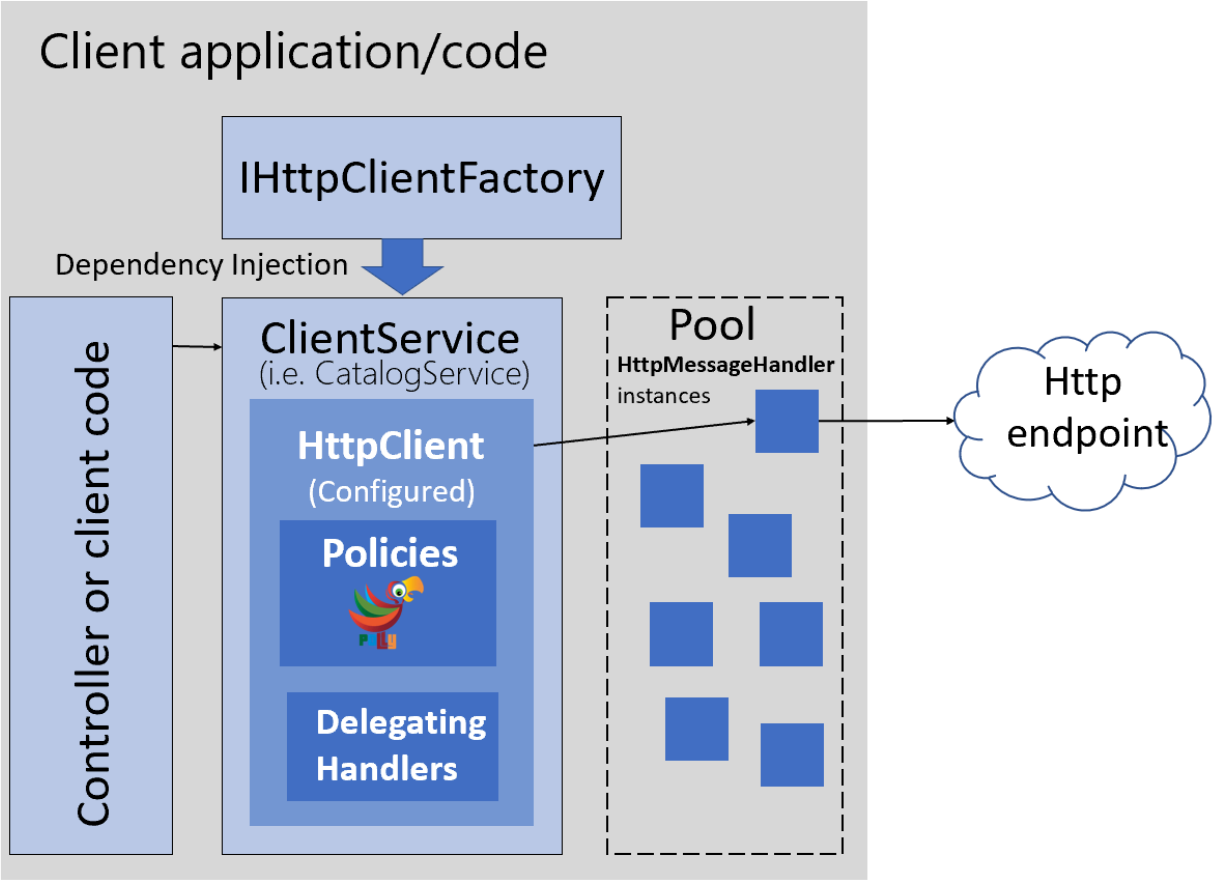


Figura 8-4. Uso de `IHttpClientFactory` con clases de cliente con tipo.

En la imagen anterior, un servicio `ClientService` (usado por un controlador o código de cliente) utiliza un cliente `HttpClient` creado por la fábrica `IHttpClientFactory` registrada. Este generador asigna a `HttpClient` un elemento `HttpMessageHandler` desde un grupo. El cliente `HttpClient` se puede configurar con las directivas de Polly al registrar la fábrica `IHttpClientFactory` en el contenedor de DI con el método de extensión AddHttpClient.

Para configurar la estructura anterior, agregue IHttpClientFactory en la aplicación instalando el paquete NuGet de `Microsoft.Extensions.Http` que incluye el método de extensión AddHttpClient para IServiceCollection. Este método de extensión registra la clase `DefaultHttpClientFactory` interna que se usará como singleton para la interfaz `IHttpClientFactory`. Define una configuración transitoria para el HttpMessageHandlerBuilder. Este controlador de mensajes (el objeto HttpMessageHandler), tomado de un grupo, lo usa el `HttpClient` devuelto desde la fábrica.

En el siguiente fragmento de código, puede ver cómo se puede usar `AddHttpClient()` para registrar clientes tipados (agentes de servicio) que deben usar `HttpClient`.

```
C#
```

```
// Program.cs
//Add http client services at ConfigureServices(IServiceCollection services)
builder.Services.AddHttpClient<ICatalogService, CatalogService>();
builder.Services.AddHttpClient<IBasketService, BasketService>();
builder.Services.AddHttpClient<IOrderingService, OrderingService>();
```

Al registrar los servicios de cliente como se muestra en el fragmento de código anterior, el `DefaultClientFactory` crea un `HttpClient` estándar para cada servicio. El cliente con tipo se registra como transitorio con contenedor DI (de inserción con dependencia). En el código anterior, `AddHttpClient()` registra *CatalogService*, *BasketService*, *OrderingService* como servicios transitorios para que se puedan insertar y consumir directamente sin necesidad de registros adicionales.

También puede agregar una configuración específica de instancia en el registro para, por ejemplo, configurar la dirección base y agregar algunas directivas de resistencia, como se muestra en lo siguiente:

C#

```
builder.Services.AddHttpClient<ICatalogService, CatalogService>(client =>
{
    client.BaseAddress = new Uri(builder.Configuration["BaseUrl"]);
})
    .AddPolicyHandler(GetRetryPolicy())
    .AddPolicyHandler(GetCircuitBreakerPolicy());
```

En este ejemplo siguiente, puede ver la configuración de una de las directivas anteriores:

C#

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode ==
System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt =>
TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)));
}
```

Puede encontrar más detalles sobre el uso de Polly en el artículo siguiente.

Vigencias de HttpClient

Cada vez que obtiene un objeto `HttpClient` del `IHttpClientFactory`, recibe una nueva instancia. Sin embargo, cada `HttpClient` usa una `HttpMessageHandler` agrupada y reutilizada por el `IHttpClientFactory` para reducir el consumo de recursos, siempre y cuando la vida útil del `HttpMessageHandler` no haya expirado.

La agrupación de controladores es deseable, ya que cada controlador normalmente administra sus propias conexiones HTTP subyacentes; la creación de más controladores de los necesarios puede dar lugar a retrasos de conexión. Algunos controladores también mantienen abiertas las conexiones indefinidamente, lo que puede impedir que el controlador reaccione a los cambios de DNS.

Los objetos `HttpMessageHandler` del grupo tienen una duración que es el período de tiempo que se puede reutilizar una instancia de `HttpMessageHandler` en el grupo. El valor predeterminado es de dos minutos, pero se puede invalidar por cada cliente con tipo. Para ello, llame a `SetHandlerLifetime()` en el IHttpClientBuilder que se devuelve cuando se crea el cliente, como se muestra en el siguiente código:

C#

```
//Set 5 min as the lifetime for the HttpMessageHandler objects in the pool
used for the Catalog Typed Client
builder.Services.AddHttpClient<ICatalogService, CatalogService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

Cada cliente con tipo puede tener configurado su propio valor de duración de controlador. Establezca la duración en `InfiniteTimeSpan` para deshabilitar la expiración del controlador.

Implemente las clases de cliente tipadas que usan el HttpClient inyectado y configurado.

Como paso anterior, debe tener definidas las clases de cliente con tipo, como las clases del código de ejemplo, como "BasketService", "CatalogService", "OrderingService", etc. – Un cliente con tipo es una clase que acepta un objeto `HttpClient` (insertado a través de su constructor) y lo usa para llamar a algún servicio HTTP remoto. Por ejemplo:

C#

```
public class CatalogService : ICatalogService
{
    private readonly HttpClient _httpClient;
    private readonly string _remoteServiceBaseUrl;
```

```
    public CatalogService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<Catalog> GetCatalogItems(int page, int take,
                                               int? brand, int? type)
    {
        var uri = API.Catalog.GetAllCatalogItems(_remoteServiceBaseUrl,
                                                 page, take, brand, type);

        var responseString = await _httpClient.GetStringAsync(uri);

        var catalog = JsonConvert.DeserializeObject<Catalog>
(responseString);
        return catalog;
    }
}
```

El Typed Client (`CatalogService` en el ejemplo) se activa mediante DI (Dependency Injection), lo que significa que puede aceptar cualquier servicio registrado en su constructor, además de `HttpClient`.

Un cliente tipado es efectivamente un objeto transitorio, lo que significa que se crea una nueva instancia cada vez que se necesita. Recibe una nueva instancia de `HttpClient` cada vez que se construye. Pero los objetos `HttpMessageHandler` del grupo son los objetos que varias instancias de `HttpClient` reutilizan.

Usar las clases de cliente con tipo

Por último, una vez que haya implementado las clases tipadas, puede registrarlas y configurarlas con `AddHttpClient()`. Después de eso, puede usarlos dondequiera que los servicios se inserten mediante DI, como en el código de la página de Razor o en un controlador de aplicación web MVC, que se muestra en el código siguiente de eShopOnContainers:

C#

```
namespace Microsoft.eShopOnContainers.WebMVC.Controllers
{
    public class CatalogController : Controller
    {
        private ICatalogService _catalogSvc;

        public CatalogController(ICatalogService catalogSvc) =>
                                                            _catalogSvc =
catalogSvc;
```

```
        public async Task<IActionResult> Index(int? BrandFilterApplied,
                                               int? TypesFilterApplied,
                                               int? page,
                                               [FromQuery]string errorMsg)
        {
            var itemsPage = 10;
            var catalog = await _catalogSvc.GetCatalogItems(page ?? 0,
                                                            itemsPage,

BrandFilterApplied,

TypesFilterApplied);
            //… Additional code
        }

        }
}
```

Hasta este punto, el fragmento de código anterior solo muestra el ejemplo de realizar solicitudes HTTP normales. Pero la "magia" se encuentra en las siguientes secciones, donde muestra cómo todas las solicitudes HTTP realizadas por `HttpClient` pueden tener políticas resilientes, como reintentos con retroceso exponencial, interruptores automáticos, características de seguridad mediante tokens de autenticación o incluso cualquier otra característica personalizada. Y todo esto se puede hacer simplemente agregando directivas y delegando controladores a los clientes con tipo registrados.

Recursos adicionales

- **Directrices de HttpClient para .NET**https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/http/httpclient-guidelines
- **Uso de HttpClientFactory en .NET**https://learn.microsoft.com/en-us/dotnet/core/extensions/httpclient-factory
- **Uso de HttpClientFactory en ASP.NET Core**https://learn.microsoft.com/aspnet/core/fundamentals/http-requests
- **Código fuente de HttpClientFactory en el repositorio de GitHub** `dotnet/runtime`https://github.com/dotnet/runtime/tree/release/7.0/src/libraries/Microsoft.Extensions.Http/
- **Polly (biblioteca de resiliencia y manejo de errores transitorios de .NET)**https://thepollyproject.azurewebsites.net/

siguiente

Implementación de reintentos de llamada HTTP con retroceso exponencial con IHttpClientFactory y las directivas de Polly

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El enfoque recomendado para los reintentos con retroceso exponencial es aprovechar las bibliotecas de .NET más avanzadas, como la biblioteca Polly de código abierto.

Polly es una biblioteca de .NET que proporciona funcionalidades de control de errores transitorios y resistencia. Puede implementar esas funcionalidades aplicando directivas de Polly, como Retry, Circuit Breaker, Bulkhead Isolation, Timeout y Fallback. Polly tiene como destino .NET Framework 4.x y .NET Standard 1.0, 1.1 y 2.0 (que admite .NET Core y versiones posteriores).

En los pasos siguientes se muestra cómo usar reintentos HTTP con Polly integrados en `IHttpClientFactory`, que se explica en la sección anterior.

Instalación de paquetes .NET

En primer lugar, deberá instalar el `Microsoft.Extensions.Http.Polly` paquete.

- Instalación con Visual Studio
- Instalación con la CLI de dotnet

- Instala con nuget.exe CLI
- Instalación con la consola del Administrador de paquetes (PowerShell)

Referencia a los paquetes de .NET 8

`IHttpClientFactory` está disponible desde .NET Core 2.1, pero se recomienda usar los paquetes de .NET 8 más recientes de NuGet en el proyecto. Normalmente, también debe hacer referencia al paquete de extensión `Microsoft.Extensions.Http.Polly`.

Configura un cliente con la política de reintento de Polly al iniciar la aplicación

El método **AddPolicyHandler()** es lo que agrega directivas a los `HttpClient` objetos que usará. En este caso, se agrega una directiva de Polly para reintentos HTTP con retroceso exponencial.

Para tener un enfoque más modular, la directiva de reintento http se puede definir en un método independiente dentro del archivo *Program.cs* , como se muestra en el código siguiente:

C#

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,

retryAttempt)));
}
```

Tal como se muestra en las secciones anteriores, debe definir una configuración HttpClient de cliente con nombre o con tipo en la configuración estándar de la aplicación *Program.cs*. Ahora agregamos código incremental que especifica la directiva para los reintentos HTTP con retroceso exponencial, tal como se indica a continuación:

C#

```
// Program.cs
builder.Services.AddHttpClient<IBasketService, BasketService>()
        .SetHandlerLifetime(TimeSpan.FromMinutes(5))  //Set lifetime to five
minutes
        .AddPolicyHandler(GetRetryPolicy());
```

Con Polly, puede definir una directiva de reintento con el número de reintentos, la configuración de retroceso exponencial y las acciones que se deben realizar cuando haya una

excepción HTTP, como registrar el error. En este caso, la política está configurada para intentarlo seis veces con un reintento exponencial, iniciando en dos segundos.

Agregar una estrategia de vibración a la directiva de reintentos

Una directiva de reintentos normal puede afectar a su sistema en casos de escalabilidad y simultaneidad altas y de gran contención. Para gestionar los picos de reintentos similares procedentes de diferentes clientes en interrupciones parciales, una buena solución es agregar una estrategia de vibración a la directiva o algoritmo de reintento. Esta estrategia puede mejorar el rendimiento general del sistema de un extremo a otro. Tal y como se recomienda en Polly: reintento con vibración, se puede implementar una buena estrategia de vibración mediante intervalos de reintentos distribuidos y uniformes, aplicados con un intervalo de reintento inicial medio bien controlado en un retroceso exponencial. Este enfoque ayuda a distribuir los picos cuando surge el problema. El principio se muestra en el ejemplo siguiente:

C#

```
var delay = Backoff.DecorrelatedJitterBackoffV2(medianFirstRetryDelay:
TimeSpan.FromSeconds(1), retryCount: 5);

var retryPolicy = Policy
    .Handle<FooException>()
    .WaitAndRetryAsync(delay);
```

Recursos adicionales

- **Patrón de reintento**https://learn.microsoft.com/azure/architecture/patterns/retry
- **Polly e IHttpClientFactory**https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory
- **Polly (biblioteca de resiliencia y manejo de errores transitorios de .NET)**https://github.com/App-vNext/Polly
- **Polly: reintentar con vibración**https://github.com/App-vNext/Polly/wiki/Retry-with-jitter
- **Marc Brooker. Jitter: Making Things Better With Randomness (Vibración: hacer mejor las cosas gracias a la aleatoriedad)** https://brooker.co.za/blog/2015/03/21/backoff.html

Siguiente anterior

Implementación del patrón de interruptor

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Como se indicó anteriormente, debe controlar los errores que pueden tardar una cantidad variable de tiempo en recuperarse, como podría ocurrir al intentar conectarse a un servicio remoto o recurso. Controlar este tipo de error puede mejorar la estabilidad y la resistencia de una aplicación.

En un entorno distribuido, las llamadas a servicios y recursos remotos pueden producir errores debido a errores transitorios, como conexiones de red lentas y tiempos de espera, o si los recursos responden lentamente o no están disponibles temporalmente. Estos errores suelen corregirse solos pasado un tiempo, y una aplicación en la nube sólida debería estar preparada para controlarlos mediante el uso de una estrategia como el "Patrón de reintento".

Sin embargo, también puede haber situaciones en las que los errores se deben a eventos imprevistos que pueden tardar mucho más tiempo en corregirse. La gravedad de estos errores puede abarcar desde una pérdida parcial de la conectividad hasta la total detención de un servicio. En estas situaciones, podría ser inútil que una aplicación vuelva a intentar continuamente una operación que es poco probable que se realice correctamente.

En su lugar, la aplicación debe codificarse para aceptar que se ha producido un error en la operación y controlar el error en consecuencia.

El uso de los reintentos HTTP de forma descuidada podría crear ataques por denegación de servicio (DoS) dentro de su propio software. Dado que se produce un error en un microservicio o se realiza lentamente, varios clientes pueden reintentar repetidamente las solicitudes con errores. Esto crea un riesgo peligroso de un aumento exponencial del tráfico dirigido al servicio fallido.

Por lo tanto, necesita algún tipo de barrera de defensa para que las solicitudes excesivas se detengan cuando no vale la pena seguir intentando. Esa barrera de defensa es precisamente el interruptor.

El patrón de interruptor tiene una finalidad distinta a la del "patrón de reintento". El "patrón de reintento" permite a una aplicación reintentar una operación con la expectativa de que la operación se realizará correctamente. El patrón Circuit Breaker impide que una aplicación realice una operación que probablemente produzca un error. Una aplicación puede combinar estos dos patrones. Sin embargo, la lógica de reintento debe ser sensible a cualquier excepción devuelta por el disyuntor y debe abandonar los reintentos si el disyuntor indica que un error no es transitorio.

Implementación de un patrón de interruptor con `IHttpClientFactory` y Polly

Como sucede al implementar los reintentos, el enfoque recomendado para los interruptores es aprovechar las bibliotecas .NET de eficacia probada como Polly y su integración nativa con `IHttpClientFactory`.

Agregar una directiva de interruptor a la canalización de software intermedio saliente de `IHttpClientFactory` es tan sencillo como agregar un único fragmento de código incremental a lo que ya tiene cuando se usa `IHttpClientFactory`.

La única adición aquí al código usado para los reintentos de llamada HTTP es el código donde se agrega la directiva Circuit Breaker a la lista de directivas que se van a usar, como se muestra en el siguiente código incremental.

C#

```
// Program.cs
var retryPolicy = GetRetryPolicy();
var circuitBreakerPolicy = GetCircuitBreakerPolicy();

builder.Services.AddHttpClient<IBasketService, BasketService>()
        .SetHandlerLifetime(TimeSpan.FromMinutes(5))  // Sample: default lifetime
is 2 minutes
        .AddHttpMessageHandler<HttpClientAuthorizationDelegatingHandler>()
```

```
        .AddPolicyHandler(retryPolicy)
        .AddPolicyHandler(circuitBreakerPolicy);
```

El `AddPolicyHandler()` método es lo que agrega directivas a los `HttpClient` objetos que usará. En este caso, se agrega una directiva de Polly para un interruptor.

Para tener un enfoque más modular, la directiva circuit breaker se define en un método independiente denominado `GetCircuitBreakerPolicy()`, como se muestra en el código siguiente:

```C#
// also in Program.cs
static IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(5, TimeSpan.FromSeconds(30));
}
```

En el ejemplo de código anterior, la directiva de interruptor se configura para que interrumpa o abra el circuito cuando se hayan producido cinco fallos consecutivos al reintentar las solicitudes HTTP. Cuando esto ocurre, el circuito se interrumpirá durante 30 segundos. En ese período, las llamadas no se podrán realizar debido al interruptor del circuito. La directiva interpreta automáticamente las excepciones pertinentes y los códigos de estado HTTP como errores.

Los interruptores también se deben usar para redirigir las solicitudes a una infraestructura de reserva siempre que haya tenido problemas en un recurso concreto implementado en otro entorno que no sea el de la aplicación cliente o del servicio que realiza la llamada HTTP. De este modo, si se produce una interrupción en el centro de datos que afecta solo a los microservicios de back-end, pero no a las aplicaciones cliente, estas aplicaciones pueden redirigir a los servicios de reserva. Polly está creando una directiva nueva para automatizar este escenario de directiva de conmutación por error.

Todas estas características sirven para los casos en los que se administra la conmutación por error desde el código .NET, y no cuando Azure lo hace de forma automática, con la transparencia de ubicación.

Desde un punto de vista de uso, al usar HttpClient, no es necesario agregar nada nuevo aquí porque el código es el mismo que cuando se usa `HttpClient` con `IHttpClientFactory`, como se muestra en las secciones anteriores.

Prueba de reintentos HTTP e interruptores en eShopOnContainers

Siempre que inicie la solución eShopOnContainers en un host de Docker, debe iniciar varios contenedores. Algunos de los contenedores son más lentos para iniciarse e inicializarse, como el contenedor de SQL Server. Esto es especialmente cierto la primera vez que implementa la aplicación eShopOnContainers en Docker porque necesita configurar las imágenes y la base de datos. El hecho de que algunos contenedores se inicien más lentamente que otros pueden hacer que el resto de los servicios inicie inicialmente excepciones HTTP, incluso si establece dependencias entre contenedores en el nivel docker-compose, como se explicó en las secciones anteriores. Esas dependencias de docker-compose entre contenedores solo están en el nivel de proceso. Es posible que se inicie el proceso de punto de entrada del contenedor, pero es posible que SQL Server no esté listo para las consultas. El resultado puede ser una cascada de errores y la aplicación puede obtener una excepción al intentar consumir ese contenedor determinado.

También puede ver este tipo de error al iniciarse cuando la aplicación se implementa en la nube. En ese caso, los orquestadores pueden mover contenedores de un nodo o máquina virtual a otro (es decir, iniciar nuevas instancias) al equilibrar el número de contenedores entre los nodos del clúster.

La forma en que "eShopOnContainers" resuelve esos problemas al iniciar todos los contenedores es mediante el patrón Retry mostrado anteriormente.

Prueba del disyuntor en eShopOnContainers

Hay varias maneras de romper o abrir el circuito y probarlo con eShopOnContainers.

Una opción es reducir el número permitido de reintentos a 1 en la directiva del disyuntor y volver a implementar toda la solución en Docker. Con un solo reintento, hay una gran probabilidad de que una solicitud HTTP falle durante la implementación, el interruptor se abra y se produzca un error.

Otra opción es usar middleware personalizado que se implementa en el microservicio **Basket** . Cuando este middleware está habilitado, detecta todas las solicitudes HTTP y devuelve el código de estado 500. Para habilitar el middleware, envíe una solicitud GET al URI que falla, de forma similar a esta:

- `GET http://localhost:5103/failing`
  Esta solicitud devuelve el estado actual del middleware. Si el middleware está habilitado, el código de estado de la solicitud será devuelto como 500. Si el middleware está deshabilitado, no hay respuesta.

- `GET http://localhost:5103/failing?enable`
  Esta solicitud habilita el middleware.
- `GET http://localhost:5103/failing?disable`
  Esta solicitud deshabilita el middleware.

Por ejemplo, una vez que se ejecuta la aplicación, puede habilitar el middleware realizando una solicitud mediante el siguiente URI en cualquier explorador. Tenga en cuenta que el microservicio de pedidos usa el puerto 5103.

`http://localhost:5103/failing?enable`

A continuación, puede comprobar el estado mediante el URI `http://localhost:5103/failing`, como se muestra en la figura 8-5.

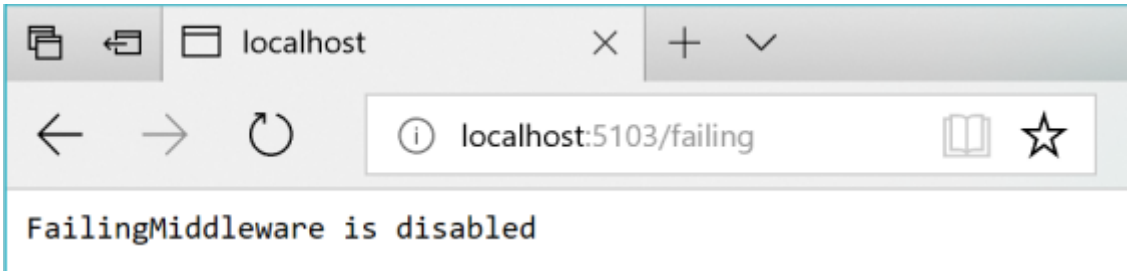


Figura 8-5. Comprobación del estado del middleware ASP.NET "con errores": en este caso, deshabilitado.

En este momento, el microservicio Basket responde con el código de estado 500 cada vez que se llama a invocarlo.

Una vez que se ejecuta el middleware, puede intentar realizar un pedido desde la aplicación web MVC. Como se produce un error en las solicitudes, el circuito se abre.

En el ejemplo siguiente, la aplicación web MVC presenta un bloque catch en la lógica para realizar un pedido. Si el código detecta una excepción de circuito abierto, muestra al usuario un mensaje descriptivo que les indica que esperen.

C#

```
public class CartController : Controller
{
    //…
    public async Task<IActionResult> Index()
    {
        try
        {
            var user = _appUserParser.Parse(HttpContext.User);
            //Http requests using the Typed Client (Service Agent)
            var vm = await _basketSvc.GetBasket(user);
```

```
            return View(vm);
        }
        catch (BrokenCircuitException)
        {
            // Catches error when Basket.api is in circuit-opened mode
            HandleBrokenCircuitException();
        }
        return View();
    }

    private void HandleBrokenCircuitException()
    {
        TempData["BasketInoperativeMsg"] = "Basket Service is inoperative, please
try later on. (Business message due to Circuit-Breaker)";
    }
}
```

Este es un resumen. La política de reintento intenta varias veces realizar la solicitud HTTP y encuentra errores HTTP. Cuando el número de reintentos alcanza el número máximo establecido para la directiva del interruptor (en este caso, 5), la aplicación genera una excepción BrokenCircuitException. El resultado es un mensaje amigable, como se muestra en la Figura 8-6.

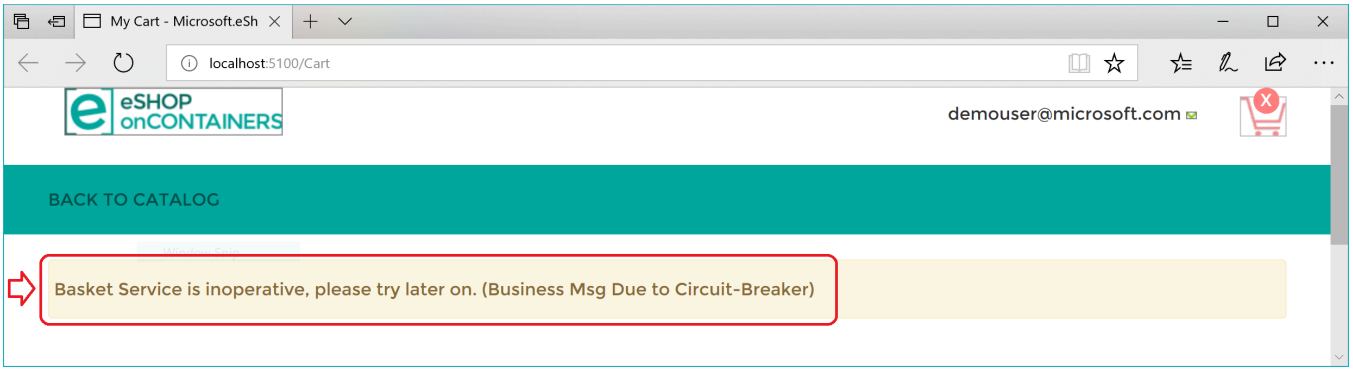


Figura 8-6. Interruptor que devuelve un error en la interfaz de usuario

Puede implementar una lógica diferente para cuándo abrir o interrumpir el circuito. También puede probar una solicitud HTTP en un microservicio de back-end distinto si se dispone de un centro de datos de reserva o un sistema back-end redundante.

Finalmente, otra opción para `CircuitBreakerPolicy` es usar `Isolate` (que fuerza la apertura y mantiene abierto el circuito) y `Reset` (que lo cierra de nuevo). Se pueden usar para crear un punto de conexión HTTP utilitario que invoque Isolate y Reset directamente en la política. Este punto de conexión HTTP se podría utilizar, debidamente protegido, en producción para aislar temporalmente un sistema aguas abajo, como cuando quieras actualizarlo. También puede activar el circuito manualmente para proteger un sistema de nivel inferior que le parezca que está fallando.

Recursos adicionales

- **Patrón Circuit Breaker**
  https://learn.microsoft.com/azure/architecture/patterns/circuit-breaker

Siguiente anterior

Supervisión del estado

Artículo • 10/09/2024

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

El seguimiento de estado puede permitir información prácticamente en tiempo real sobre el estado de los contenedores y los microservicios. El seguimiento de estado es fundamental para varios aspectos del funcionamiento de los microservicios y es especialmente importante cuando los orquestadores realizan actualizaciones de aplicación parcial en fases, tal como se describe más adelante.

Las aplicaciones basadas en microservicios suelen usar latidos o comprobaciones de estado para que sus monitores de rendimiento, programadores y orquestadores puedan realizar el seguimiento de gran cantidad de servicios. Si los servicios no pueden enviar algún tipo de señal "estoy activo", ya sea a petición o siguiendo una programación, la aplicación podría correr riesgos al implementar las actualizaciones, o podría simplemente detectar los errores demasiado tarde y no poder detener errores en cascada que pueden dar lugar a interrupciones importantes.

En el modelo típico, los servicios envían informes sobre su estado. Esa información se agrega para proporcionar una visión general del estado de la aplicación. Si se utiliza un orquestador, se puede proporcionar información de estado al clúster del orquestador a fin de que el clúster pueda actuar en consecuencia. Si se invierte en informes de estado de alta calidad personalizados para la aplicación, se pueden detectar y corregir mucho más fácilmente los problemas de la aplicación que se está ejecutando.

Implementación de comprobaciones de estado en servicios de ASP.NET Core

Al desarrollar una aplicación web o microservicio de ASP.NET Core, puede usar la característica de comprobaciones de estado integrada que se lanzó en ASP.NET Core 2.2 (Microsoft.Extensions.Diagnostics.HealthChecks). Al igual que muchas características de ASP.NET Core, las comprobaciones de estado incluyen un conjunto de servicios y un middleware.

Los servicios de comprobación de estado y middleware son fáciles de usar y proporcionan características que permiten validar el funcionamiento correcto de cualquier recurso externo necesario para la aplicación (por ejemplo, una base de datos de SQL Server o API remota). Cuando se utiliza esta característica, también se puede decidir lo que significa que el estado del recurso sea correcto, tal y como se explica más adelante.

Para usar esta característica con eficacia, primero debe configurar servicios en sus microservicios. En segundo lugar, necesita una aplicación front-end que realice consultas para los informes de estado. La aplicación front-end podría ser una aplicación de informes personalizada, o podría ser un orquestador que reaccione en consecuencia a los estados.

Uso de la característica HealthChecks en los microservicios ASP.NET de back-end

En esta sección, aprenderá a implementar la característica HealthChecks en una aplicación de ASP.NET Core 8.0 Web API de ejemplo al usar el paquete Microsoft.Extensions.Diagnostics.HealthChecks. La implementación de esta característica en un microservicio a gran escala como eShopOnContainers se explica en la siguiente sección.

Para empezar, debe definir qué constituye un estado correcto en cada microservicio. En la aplicación de ejemplo, definiremos que el estado del microservicio es correcto si se puede acceder a su API a través de HTTP y si su base de datos de SQL Server relacionada también está disponible.

En .NET 8, con las API integradas, así puede configurar los servicios, y agregar una comprobación de estado para el microservicio y su base de datos de SQL Server dependiente:

```
C#
```

```
// Program.cs from .NET 8 Web API sample

//...
// Registers required services for health checks
builder.Services.AddHealthChecks()
    // Add a health check for a SQL Server database
    .AddCheck(
        "OrderingDB-check",
        new
SqlConnectionHealthCheck(builder.Configuration["ConnectionString"]),
        HealthStatus.Unhealthy,
        new string[] { "orderingdb" });
```

En el código anterior, el método `services.AddHealthChecks()` configura una comprobación HTTP básica que devuelve un código de estado **200** con "Correcto". Además, el método de extensión `AddCheck()` configura una `SqlConnectionHealthCheck` personalizada que comprueba el estado de la base de datos SQL Database relacionado.

El método `AddCheck()` agrega una nueva comprobación de estado con un nombre especificado y la implementación de tipo `IHealthCheck`. Puede agregar varias comprobaciones de estado mediante el método AddCheck, por lo que un microservicio no proporcionará un estado "correcto" hasta que el estado de todas sus comprobaciones sea correcto.

`SqlConnectionHealthCheck` es una clase personalizada que implementa `IHealthCheck`, que toma una cadena de conexión como parámetro del constructor y ejecuta una consulta sencilla que se va a comprobar si la conexión a la base de datos SQL es correcta. Devuelve `HealthCheckResult.Healthy()` si la consulta se ejecutó correctamente y un `FailureStatus` con la excepción real si hay errores.

C#

```
// Sample SQL Connection Health Check
public class SqlConnectionHealthCheck : IHealthCheck
{
    private const string DefaultTestQuery = "Select 1";

    public string ConnectionString { get; }

    public string TestQuery { get; }

    public SqlConnectionHealthCheck(string connectionString)
        : this(connectionString, testQuery: DefaultTestQuery)
    {
    }
```

```
    public SqlConnectionHealthCheck(string connectionString, string
testQuery)
    {
        ConnectionString = connectionString ?? throw new
ArgumentNullException(nameof(connectionString));
        TestQuery = testQuery;
    }

    public async Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext
context, CancellationToken cancellationToken = default(CancellationToken))
    {
        using (var connection = new SqlConnection(ConnectionString))
        {
            try
            {
                await connection.OpenAsync(cancellationToken);

                if (TestQuery != null)
                {
                    var command = connection.CreateCommand();
                    command.CommandText = TestQuery;

                    await command.ExecuteNonQueryAsync(cancellationToken);
                }
            }
            catch (DbException ex)
            {
                return new HealthCheckResult(status:
context.Registration.FailureStatus, exception: ex);
            }
        }

        return HealthCheckResult.Healthy();
    }
}
```

> ⓘ **Importante**
>
> Microsoft recomienda usar el flujo de autenticación más seguro disponible. Si se conecta a Azure SQL, el método de autenticación recomendado es [Identidades administradas para recursos de Azure](#).

Tenga en cuenta que en el código anterior, `Select 1` es la consulta usada para comprobar el estado de la base de datos. Para supervisar la disponibilidad de los microservicios, orquestadores como Kubernetes realizan periódicamente comprobaciones de estado mediante el envío de solicitudes para probar los microservicios. Es importante mantener la eficacia de sus consultas de base de datos para que estas operaciones sean rápidas y no den lugar a una mayor utilización de recursos.

Por último, agregue un middleware que responda a la dirección URL `/hc`:

```
C#

// Program.cs from .NET 8 Web Api sample

app.MapHealthChecks("/hc");
```

Cuando se invoca, el punto de conexión `<yourmicroservice>/hc` ejecuta todas las comprobaciones de estado que están configuradas en el método `AddHealthChecks()` de la clase Startup y muestra el resultado.

Implementación de HealthChecks en eShopOnContainers

Los microservicios de eShopOnContainers se basan en varios servicios para realizar su tarea. Por ejemplo, el microservicio `Catalog.API` de eShopOnContainers depende de muchos servicios, como Azure Blob Storage, SQL Server y RabbitMQ. Por lo tanto, tiene varias comprobaciones de estado agregadas mediante el método `AddCheck()`. En todos los servicios dependientes, se debería agregar una implementación `IHealthCheck` que defina su estado de mantenimiento correspondiente.

El proyecto de código abierto AspNetCore.Diagnostics.HealthChecks resuelve este problema mediante la realización de implementaciones de comprobación de estado personalizadas en todos y cada uno de estos servicios empresariales basados en .NET 8. Cada comprobación de estado está disponible como paquete NuGet individual que se puede agregar fácilmente al proyecto. eShopOnContainers los usa mayoritariamente en todos sus microservicios.

Por ejemplo, en el microservicio `Catalog.API`, se agregaron los siguientes paquetes NuGet:

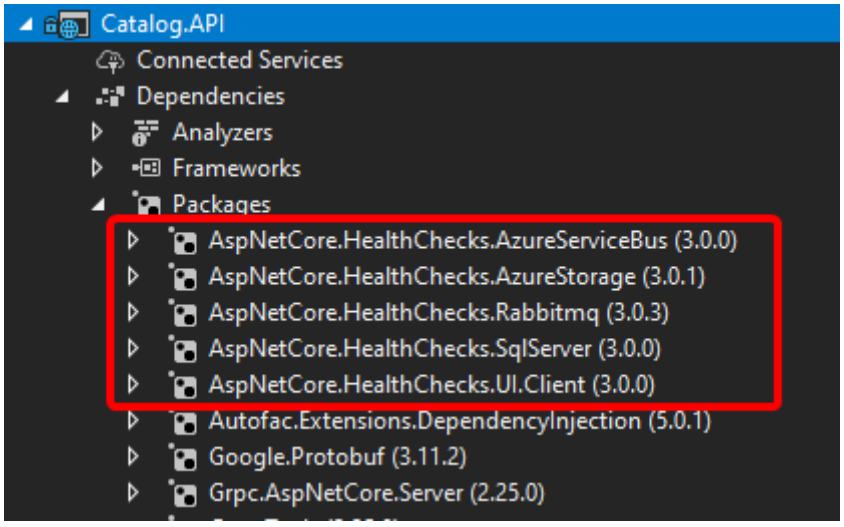


Figura 8-7. Comprobaciones de estado personalizadas implementadas en Catalog.API mediante AspNetCore.Diagnostics.HealthChecks

En el siguiente código, las implementaciones de comprobación de estado se agregan para cada servicio dependiente y, a continuación, se configura el middleware:

C#

```
// Extension method from Catalog.api microservice
//
public static IServiceCollection AddCustomHealthCheck(this
IServiceCollection services, IConfiguration configuration)
{
    var accountName = configuration.GetValue<string>
("AzureStorageAccountName");
    var accountKey = configuration.GetValue<string>
("AzureStorageAccountKey");

    var hcBuilder = services.AddHealthChecks();

    hcBuilder
        .AddSqlServer(
            configuration["ConnectionString"],
            name: "CatalogDB-check",
            tags: new string[] { "catalogdb" });

    if (!string.IsNullOrEmpty(accountName) &&
!string.IsNullOrEmpty(accountKey))
    {
        hcBuilder
            .AddAzureBlobStorage(
                $"DefaultEndpointsProtocol=https;AccountName=
{accountName};AccountKey={accountKey};EndpointSuffix=core.windows.net",
                name: "catalog-storage-check",
                tags: new string[] { "catalogstorage" });
    }
    if (configuration.GetValue<bool>("AzureServiceBusEnabled"))
    {
        hcBuilder
            .AddAzureServiceBusTopic(
                configuration["EventBusConnection"],
                topicName: "eshop_event_bus",
                name: "catalog-servicebus-check",
                tags: new string[] { "servicebus" });
    }
    else
    {
        hcBuilder
            .AddRabbitMQ(
                $"amqp://{configuration["EventBusConnection"]}",
                name: "catalog-rabbitmqbus-check",
                tags: new string[] { "rabbitmqbus" });
    }
```

```
    return services;
}
```

Por último, agregue el middleware HealthCheck que se va a escuchar al punto de conexión "/hc":

```
C#

// HealthCheck middleware
app.UseHealthChecks("/hc", new HealthCheckOptions()
{
    Predicate = _ => true,
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
```

Consulta de los microservicios para informar de su estado

Cuando haya configurado las comprobaciones de estado como se describe en este artículo, y una vez que el microservicio se esté ejecutando en Docker, puede comprobar directamente desde un explorador si su estado es correcto. Debe publicar el puerto de contenedor en el host de Docker, para así poder acceder al contenedor a través de la dirección IP del host de Docker externa host local o de `host.docker.internal`, como se muestra en la figura 8-8.

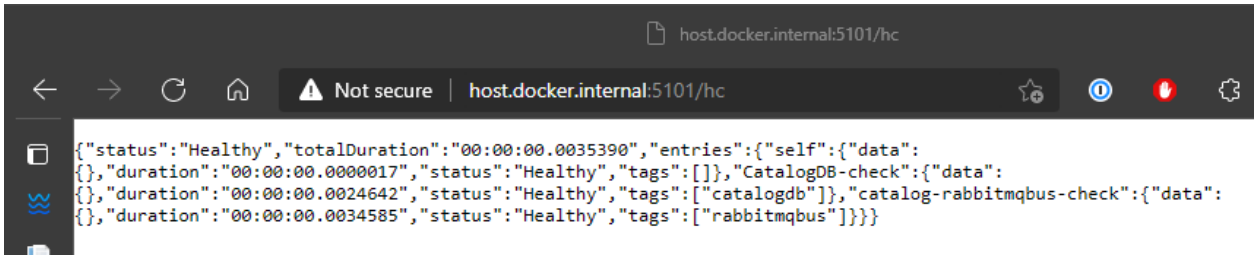


Figura 8-8. Comprobación del estado de un único servicio desde un explorador

En esa prueba, puede ver que el estado del microservicio `Catalog.API` (que se ejecuta en el puerto 5101) es correcto. Se devuelve el código de estado HTTP 200 e información de estado en JSON. El servicio también comprobó el estado de su dependencia de la base de datos de SQL Server y RabbitMQ, por lo que el estado se notificó como correcto.

Uso de guardianes

Un guardián es un servicio independiente que puede observar el estado y la carga en varios servicios, e informar del estado de los microservicios con una consulta con la

biblioteca `HealthChecks` vista anteriormente. Esto puede ayudar a evitar errores que no se detectarían si se observase un único servicio. Los guardianes también son un buen lugar para hospedar código que lleve a cabo acciones correctoras para condiciones conocidas sin la intervención del usuario.

El ejemplo de eShopOnContainers contiene una página web que muestra informes de comprobación de estado de ejemplo, como se muestra en la figura 8-9. Se trata del guardián más sencillo que se puede tener, dado que lo único que hace es mostrar el estado de las aplicaciones web y los microservicios en eShopOnContainers. Normalmente, un guardián también realiza acciones cuando detecta estados no correctos.

Afortunadamente, AspNetCore.Diagnostics.HealthChecks también proporciona el paquete NuGet AspNetCore.HealthChecks.UI que se puede usar para mostrar los resultados de comprobación de estado de los URI configurados.

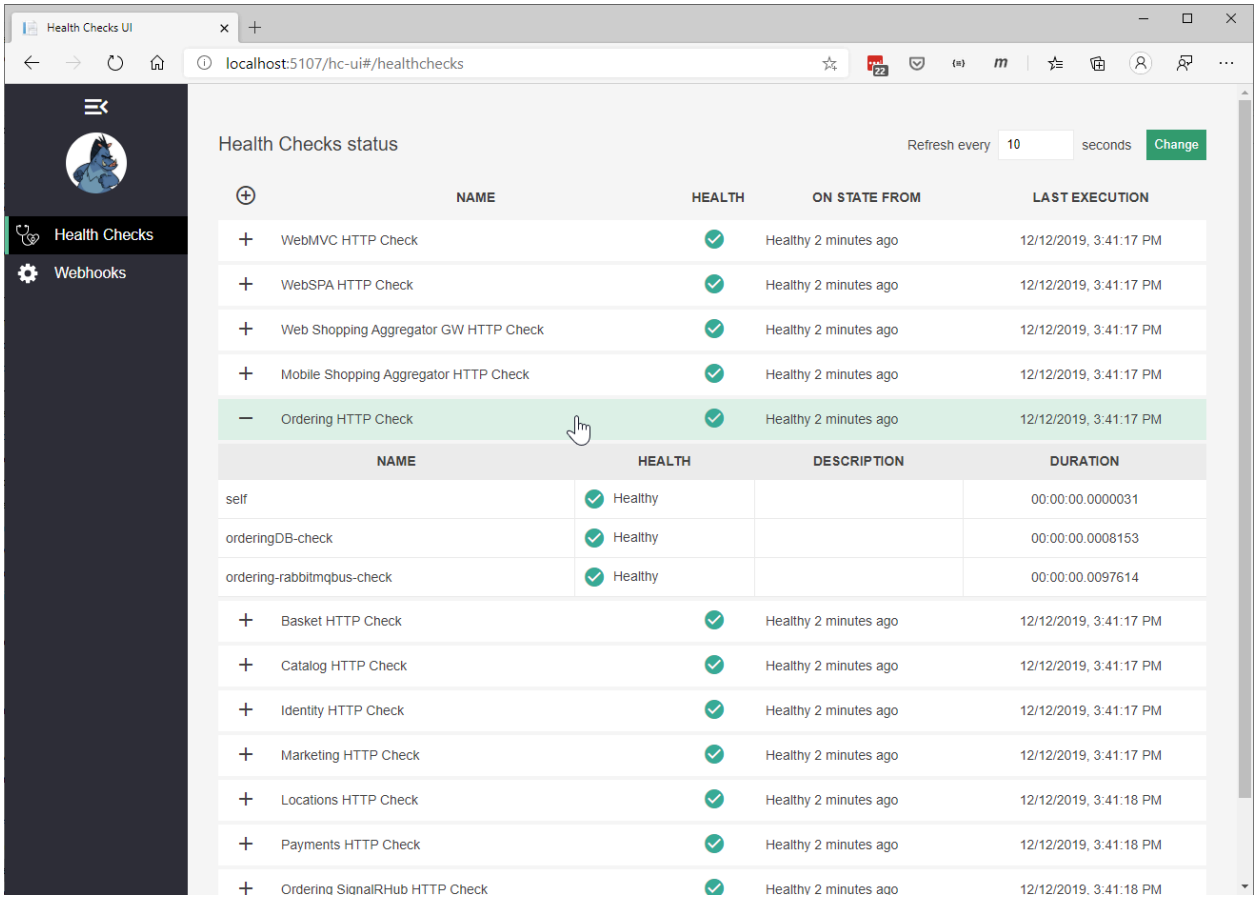


Figura 8-9. Informe de comprobación de estado de ejemplo en eShopOnContainers

En resumen, este servicio de vigilancia consulta cada uno de los puntos de conexión "/hc" del microservicio. El middleware ejecutará todas las comprobaciones de estado definidas en él y devolverá un estado general que dependerá de todas esas comprobaciones. HealthChecksUI es fácil de usar con algunas entradas de configuración y dos líneas de código que deben agregarse en el archivo *Startup.cs* del servicio de inspección.

Archivo de configuración de ejemplo para la interfaz de usuario de comprobación de estado:

JSON

```
// Configuration
{
  "HealthChecksUI": {
    "HealthChecks": [
      {
        "Name": "Ordering HTTP Check",
        "Uri": "http://host.docker.internal:5102/hc"
      },
      {
        "Name": "Ordering HTTP Background Check",
        "Uri": "http://host.docker.internal:5111/hc"
      },
      //...
    ]}
}
```

Archivo *Program.cs* que agrega HealthChecksUI:

C#

```
// Program.cs from WebStatus(Watch Dog) service
//
// Registers required services for health checks
builder.Services.AddHealthChecksUI();
// build the app, register other middleware
app.UseHealthChecksUI(config => config.UIPath = "/hc-ui");
```

Comprobaciones de estado con orquestadores

Para supervisar la disponibilidad de los microservicios, orquestadores como Kubernetes y Service Fabric realizan periódicamente comprobaciones de estado mediante el envío de solicitudes para probar los microservicios. Cuando un orquestador determina que el estado de un contenedor o servicio no es correcto, deja de enrutar las solicitudes a esa instancia. Normalmente también crea una nueva instancia de ese contenedor.

Por ejemplo, la mayoría de los orquestadores pueden utilizar comprobaciones de estado para administrar implementaciones sin tiempos de inactividad. Solo cuando el estado de un servicio o contenedor cambia a correcto, el orquestador empieza a enrutar el tráfico a las instancias de servicio o contenedor.

El seguimiento de estado es especialmente importante cuando un orquestador lleva a cabo una actualización de la aplicación. Algunos orquestadores (por ejemplo, Azure Service Fabric) actualizan los servicios en fases: por ejemplo, pueden actualizar una quinta parte de la superficie del clúster para cada actualización de la aplicación. El conjunto de nodos que se actualiza al mismo tiempo se conoce como *dominio de actualización*. Después de que cada dominio de actualización se haya actualizado y esté disponible para los usuarios, el dominio de actualización debe pasar las comprobaciones de estado antes de que la implementación se mueva al siguiente dominio de actualización.

Otro aspecto del estado del servicio es informar de las métricas del servicio. Se trata de una característica avanzada del modelo de estado de algunos orquestadores, como Service Fabric. Las métricas son importantes cuando se usa un orquestador porque se usan para equilibrar el uso de recursos. Las métricas también pueden ser un indicador del estado del sistema. Pongamos por ejemplo una aplicación que tenga muchos microservicios, y cada instancia informa sobre una métrica de solicitudes por segundo (RPS). Si un servicio está utilizando más recursos (memoria, procesador, etc.) que otro servicio, el orquestador puede mover las instancias del servicio en el clúster para intentar equilibrar el uso de los recursos.

Tenga en cuenta que Azure Service Fabric proporciona su propio modelo de seguimiento de estado, que es más avanzado que las comprobaciones de estado simples.

Supervisión avanzada: visualización, análisis y alertas

La parte final de la supervisión es visualizar la secuencia de eventos, generar informes sobre rendimiento de los servicios y emitir alertas cuando se detecta un problema. Para este aspecto de la supervisión se pueden usar diferentes soluciones.

Se pueden utilizar aplicaciones personalizadas simples que muestren el estado de los servicios, como la página personalizada mostrada al explicar AspNetCore.Diagnostics.HealthChecks. O bien, podría usar herramientas más avanzadas como Azure Monitor para generar alertas basadas en el flujo de eventos.

Por último, si almacena todos los flujos de eventos, se puede utilizar Microsoft Power BI u otras soluciones como Kibana o Splunk para visualizar los datos.

Recursos adicionales

- **HealthChecks and HealthChecks UI for ASP.NET Core (HealthChecks e interfaz de usuario de HealthChecks para ASP.NET Core)**
  https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks
- **Introduction to Service Fabric health monitoring (Introducción al seguimiento de estado de Service Fabric)**
  https://learn.microsoft.com/azure/service-fabric/service-fabric-health-introduction
- **Azure Monitor**
  https://azure.microsoft.com/services/monitor/

Anterior Siguiente

Protección de microservicios y aplicaciones web de .NET

Artículo • 21/05/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Hay tantos aspectos sobre la seguridad en microservicios y aplicaciones web que el tema podría tomar fácilmente varios libros como este. Por lo tanto, en esta sección, nos centraremos en la autenticación, autorización y secretos de aplicación.

Implementación de la autenticación en microservicios y aplicaciones web de .NET

A menudo es necesario que los recursos y las API publicados por un servicio se limite a determinados usuarios o clientes de confianza. El primer paso para tomar estas decisiones de confianza de nivel de API es la autenticación. La autenticación es el proceso de comprobar de forma confiable la identidad de un usuario.

Cuando hay microservicios, la autenticación normalmente se controla de forma centralizada. Si usa una puerta de enlace de API, la puerta de enlace es un buen lugar para autenticarse, como se muestra en la figura 9-1. Si usa este enfoque, asegúrese de que no se puede acceder directamente a los microservicios individuales (sin la puerta de enlace de API), a menos que haya seguridad adicional para autenticar mensajes tanto si proceden de la puerta de enlace como si no.

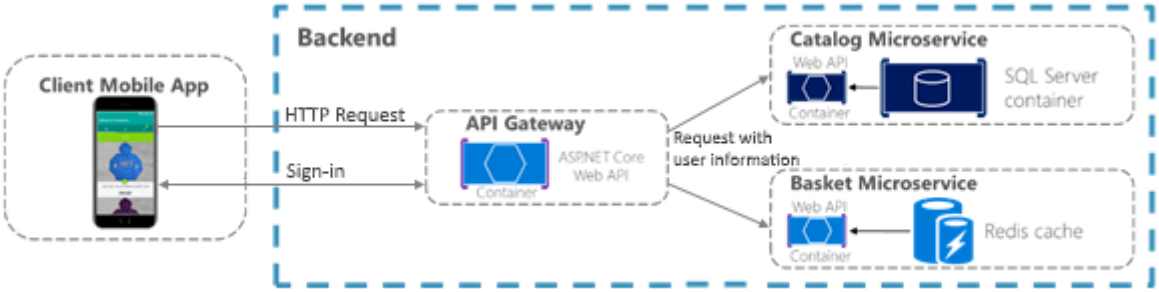


Figura 9-1. Autenticación centralizada con una puerta de enlace de API

Cuando la puerta de enlace de API centraliza la autenticación, agrega información de usuario al reenviar solicitudes a los microservicios. Si se puede acceder directamente a los servicios, se puede usar un servicio de autenticación como Azure Active Directory o un microservicio de autenticación dedicado que actúe como servicio de token de seguridad (STS) para autenticar a los usuarios. Las decisiones de confianza se comparten entre los servicios con tokens de seguridad o cookies. (Estos tokens se pueden compartir entre ASP.NET aplicaciones core, si es necesario, mediante la implementación de uso compartido de cookies). Este patrón se muestra en la figura 9-2.

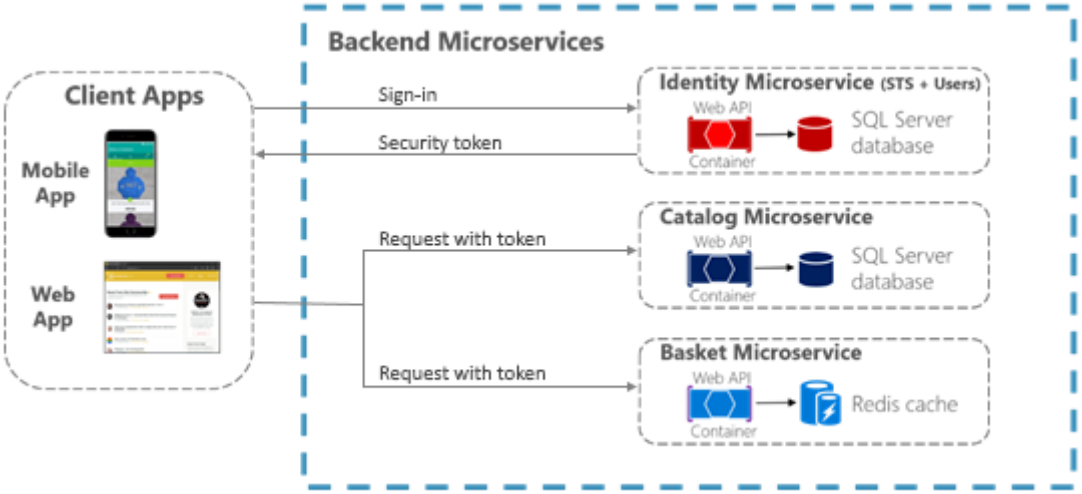


Figura 9-2. Autenticación por microservicio de identidad; trust se comparte mediante un token de autorización

Cuando se accede directamente a los microservicios, la confianza, que incluye la autenticación y la autorización, se controla mediante un token de seguridad emitido por un microservicio dedicado, compartido entre microservicios.

Autenticación con ASP.NET Core Identity

El mecanismo principal de ASP.NET Core para identificar a los usuarios de una aplicación es el sistema de pertenencia ASP.NET Core Identity . Identity de ASP.NET Core almacena información de usuario (incluida la información de inicio de sesión, roles y notificaciones) en un almacén de datos configurado por el desarrollador. Normalmente, el almacén de datos de

ASP.NET Core Identity es un almacén de Entity Framework proporcionado en el `Microsoft.AspNetCore.Identity.EntityFrameworkCore` paquete. Sin embargo, los almacenes personalizados u otros paquetes de terceros se pueden usar para almacenar información de identidad en Azure Table Storage, CosmosDB u otras ubicaciones.

> **Sugerencia**
>
> ASP.NET Core 2.1 y versiones posteriores proporciona ASP.NET Core Identity como biblioteca de clases de Razor, por lo que no verá gran parte del código necesario en el proyecto, como era el caso de las versiones anteriores. Para más información sobre cómo personalizar el código de identidad para satisfacer sus necesidades, consulte Scaffold Identity in ASP.NET Core projects.

El código siguiente se toma de la plantilla de proyecto ASP.NET Core Web Application MVC con la autenticación de cuenta de usuario individual seleccionada. Muestra cómo configurar ASP.NET Core Identity mediante Entity Framework Core en el archivo *Program.cs* .

C#

```
//...
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection")));

builder.Services.AddDefaultIdentity<IdentityUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();
//...
```

Una vez configurada ASP.NET Core Identity, habilítela agregando los `app.UseAuthentication()` y `endpoints.MapRazorPages()` como se muestra en el código siguiente en el archivo *Program.cs* del servicio:

C#

```
//...
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
```

```
});
//...
```

> **Importante**
>
> Las líneas del código anterior **DEBEN ESTAR EN EL ORDEN MOSTRADO** para que la identidad funcione correctamente.

El uso de ASP.NET Core Identity habilita varios escenarios:

- Cree nueva información de usuario utilizando el tipo UserManager (userManager.CreateAsync).
- Autentíquese a los usuarios mediante el tipo SignInManager. Puede usar `signInManager.SignInAsync` para iniciar sesión directamente o `signInManager.PasswordSignInAsync` para confirmar que la contraseña del usuario es correcta y, a continuación, iniciar sesión.
- Identifique a un usuario en función de la información almacenada en una cookie (que es leída por el middleware de ASP.NET Core Identity) para que las solicitudes posteriores de un navegador incluyan la identidad y las reclamaciones del usuario que ha iniciado sesión.

ASP.NET Core Identity también admite la autenticación en dos fases.

En escenarios de autenticación que usan un almacén de datos de usuario local y que conservan la identidad entre solicitudes mediante cookies (como es habitual para las aplicaciones web de MVC), ASP.NET Core Identity es una solución recomendada.

Autenticación con proveedores externos

ASP.NET Core también admite el uso de proveedores de autenticación externos para permitir que los usuarios inicien sesión a través de flujos de OAuth 2.0 . Esto significa que los usuarios pueden iniciar sesión con procesos de autenticación existentes de proveedores como Microsoft, Google, Facebook o Twitter y asociar esas identidades a una identidad de ASP.NET Core en la aplicación.

Para usar la autenticación externa, además de incluir el middleware de autenticación como se mencionó anteriormente utilizando el método `app.UseAuthentication()`, también tiene que registrar el proveedor externo en *Program.cs* como se muestra en el ejemplo siguiente.

```
C#
```

```
//...
services.AddDefaultIdentity<IdentityUser>(options =>
options.SignIn.RequireConfirmedAccount = true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions =>
    {
        microsoftOptions.ClientId =
builder.Configuration["Authentication:Microsoft:ClientId"];
        microsoftOptions.ClientSecret =
builder.Configuration["Authentication:Microsoft:ClientSecret"];
    })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
//...
```

Los proveedores de autenticación externos populares y sus paquetes NuGet asociados se muestran en la tabla siguiente:

Expandir tabla

proveedor	Paquete
Microsoft	Microsoft.AspNetCore.Authentication.MicrosoftAccount
Google	Microsoft.AspNetCore.Authentication.Google
Facebook	Microsoft.AspNetCore.Authentication.Facebook
Twitter	Microsoft.AspNetCore.Authentication.Twitter

En todos los casos, debe completar un procedimiento de registro de aplicación dependiente del proveedor y que normalmente implica:

1. Obtención de un identificador de aplicación cliente.
2. Obtener un secreto de aplicación cliente.
3. Configuración de la URL de redireccionamiento que está controlada por el middleware de autorización y el proveedor registrado.
4. Opcionalmente, configurar una dirección URL de cierre de sesión para controlar correctamente el cierre de sesión en un escenario de inicio de sesión único (SSO).

Para obtener más información sobre cómo configurar la aplicación para un proveedor externo, consulte autenticación del proveedor externo en la documentación de ASP.NET Core).

Sugerencia

Todos los detalles se controlan mediante el middleware de autorización y los servicios mencionados anteriormente. Por lo tanto, solo tiene que elegir la opción de autenticación **cuenta de usuario individual** al crear el proyecto de aplicación web ASP.NET Core en Visual Studio, como se muestra en la figura 9-3, además de registrar los proveedores de autenticación mencionados anteriormente.

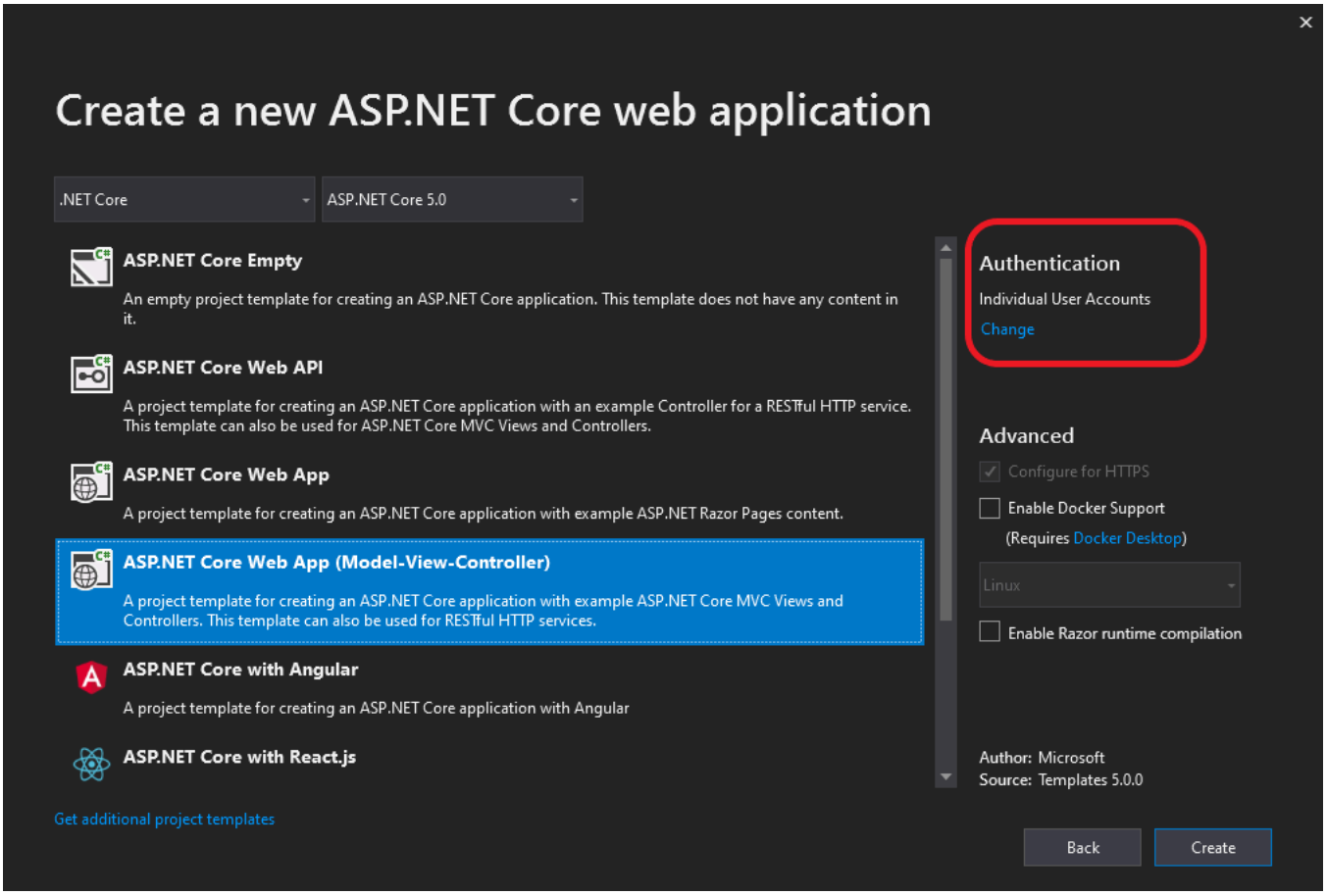


Figura 9-3. Al crear un proyecto de aplicación web en Visual Studio 2019, seleccione la opción Cuentas de usuario individuales para usar la autenticación externa.

Además de los proveedores de autenticación externos enumerados anteriormente, hay paquetes de terceros disponibles que proporcionan middleware para usar muchos más proveedores de autenticación externos. Para obtener una lista, consulte el repositorio AspNet.Security.OAuth.Providers en GitHub.

También puede crear su propio middleware de autenticación externa para resolver algunas necesidades especiales.

Autenticación con tokens de portador

La autenticación con ASP.NET Core Identity (o Identity más proveedores de autenticación externos) funciona bien para muchos escenarios de aplicaciones web en los que el

almacenamiento de información del usuario en una cookie es adecuado. Sin embargo, en otros escenarios, las cookies no son un medio natural de conservar y transmitir datos.

Por ejemplo, en una API web de ASP.NET Core que expone puntos de conexión RESTful a los que pueden acceder aplicaciones de página única (SPA), clientes nativos o incluso otras API web, se prefiere usar la autenticación de token de portador en cambio. Estos tipos de aplicaciones no funcionan con cookies, pero pueden recuperar fácilmente un token de portador e incluirlo en el encabezado de autorización de las solicitudes posteriores. Para habilitar la autenticación de tokens, ASP.NET Core admite varias opciones para usar OAuth 2.0 y OpenID Connect.

Autenticación con un proveedor de identidades de OpenID Connect o OAuth 2.0

Si la información del usuario se almacena en Azure Active Directory u otra solución de identidad que admita OpenID Connect o OAuth 2.0, puede usar el paquete **Microsoft.AspNetCore.Authentication.OpenIdConnect** para autenticarse mediante el flujo de trabajo de OpenID Connect. Por ejemplo, para autenticarse en el microservicio Identity.Api en eShopOnContainers, una aplicación web ASP.NET Core puede usar middleware desde ese paquete, como se muestra en el ejemplo simplificado siguiente en *Program.cs*:

C#

```
// Program.cs

var identityUrl = builder.Configuration.GetValue<string>("IdentityUrl");
var callBackUrl = builder.Configuration.GetValue<string>("CallBackUrl");
var sessionCookieLifetime =
builder.Configuration.GetValue("SessionCookieLifetimeMinutes", 60);

// Add Authentication services

services.AddAuthentication(options =>
{
    options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
})
.AddCookie(setup => setup.ExpireTimeSpan =
TimeSpan.FromMinutes(sessionCookieLifetime))
.AddOpenIdConnect(options =>
{
    options.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
    options.Authority = identityUrl.ToString();
    options.SignedOutRedirectUri = callBackUrl.ToString();
    options.ClientId = useLoadTest ? "mvctest" : "mvc";
    options.ClientSecret = "secret";
    options.ResponseType = useLoadTest ? "code id_token token" : "code id_token";
    options.SaveTokens = true;
```

```
    options.GetClaimsFromUserInfoEndpoint = true;
    options.RequireHttpsMetadata = false;
    options.Scope.Add("openid");
    options.Scope.Add("profile");
    options.Scope.Add("orders");
    options.Scope.Add("basket");
    options.Scope.Add("marketing");
    options.Scope.Add("locations");
    options.Scope.Add("webshoppingagg");
    options.Scope.Add("orders.signalrhub");
});

// Build the app
//…
app.UseAuthentication();
//…
app.UseEndpoints(endpoints =>
{
    //...
});
```

Cuando se usa este flujo de trabajo, no se necesita el middleware de ASP.NET Core Identity, ya que el servicio Identity controla todo el almacenamiento de información del usuario y la autenticación.

Emisión de tokens de seguridad desde un servicio ASP.NET Core

Si prefiere emitir tokens de seguridad para usuarios locales de ASP.NET Core Identity en lugar de usar un proveedor de identidades externo, puede aprovechar algunas bibliotecas de terceros adecuadas.

IdentityServer4 y OpenIddict son proveedores de OpenID Connect que se integran fácilmente con ASP.NET Core Identity para permitirle emitir tokens de seguridad desde un servicio ASP.NET Core. La documentación de IdentityServer4 tiene instrucciones detalladas para usar la biblioteca. Sin embargo, los pasos básicos para usar IdentityServer4 para emitir tokens son los siguientes.

1. Para configurar IdentityServer4 en *Program.cs* , realice una llamada al generador. Services.AddIdentityServer.

2. Llamas al método app.UseIdentityServer en *Program.cs* para agregar IdentityServer4 a la canalización de procesamiento de solicitudes HTTP de la aplicación. Esto permite a la biblioteca atender solicitudes a los puntos de conexión de OpenID Connect y OAuth2, como /connect/token.

3. Para configurar el servidor de identidades, establezca los datos siguientes:

- Credenciales que se van a usar para firmar.
- Los recursos de identidad y API a los que los usuarios pueden solicitar acceso:
  - Los recursos de API representan datos protegidos o funcionalidad a los que un usuario puede acceder con un token de acceso. Un ejemplo de un recurso de API sería una API web (o un conjunto de API) que requiere autorización.
  - Los recursos de identidad representan información (declaraciones) que se proporciona a un cliente para identificar a un usuario. Las afirmaciones pueden incluir el nombre de usuario, la dirección de correo electrónico, etcétera.
- Los clientes que se conectarán para solicitar tokens.
- Mecanismo de almacenamiento para la información del usuario, como ASP.NET Core Identity o una alternativa.

Al especificar clientes y recursos para su uso por IdentityServer4, puede pasar una IEnumerable<T> colección del tipo adecuado a los métodos que utilizan almacenes en memoria de clientes o recursos. O bien, para escenarios más complejos, puede proporcionar tipos de cliente o de proveedor de recursos mediante inyección de dependencias.

Una configuración de ejemplo para IdentityServer4 para usar los clientes y recursos en memoria proporcionados por un tipo IClientStore personalizado podría ser similar al ejemplo siguiente:

C#

```
// Program.cs

builder.Services.AddSingleton<IClientStore, CustomClientStore>();
builder.Services.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<ApplicationUser>();
//...
```

Consumir tokens de seguridad

La autenticación en un punto de conexión de OpenID Connect o la emisión de sus propios tokens de seguridad cubre algunos escenarios. ¿Pero qué ocurre con un servicio que simplemente necesita limitar el acceso a los usuarios que tienen tokens de seguridad válidos proporcionados por un servicio diferente?

En ese escenario, el middleware de autenticación que controla los tokens JWT está disponible en el paquete **Microsoft.AspNetCore.Authentication.JwtBearer** . JWT significa "JSON Web Token" y es un formato de token de seguridad común (definido por RFC 7519) para comunicar notificaciones de seguridad. Un ejemplo simplificado de cómo usar middleware para consumir estos tokens podría parecerse a este fragmento de código, tomado del microservicio Ordering.Api de eShopOnContainers.

C#

```
// Program.cs

var identityUrl = builder.Configuration.GetValue<string>("IdentityUrl");

// Add Authentication services

builder.Services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme =
AspNetCore.Authentication.JwtBearer.JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme =
AspNetCore.Authentication.JwtBearer.JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>
{
    options.Authority = identityUrl;
    options.RequireHttpsMetadata = false;
    options.Audience = "orders";
});

// Build the app

app.UseAuthentication();
//…
app.UseEndpoints(endpoints =>
{
    //...
});
```

Los parámetros de este uso son:

- `Audience` representa el receptor del token entrante o el recurso al que el token concede acceso. Si el valor especificado en este parámetro no coincide con el parámetro del token, se rechazará el token.
- `Authority` es la dirección del servidor de autenticación emisora de tokens. El middleware de autenticación de portador JWT usa este URI para obtener la clave pública que se puede usar para validar la firma del token. El middleware también confirma que el `iss` parámetro del token coincide con este URI.

Otro parámetro, `RequireHttpsMetadata`, es útil para realizar pruebas; establezca este parámetro en false para que pueda probar en entornos en los que no tenga certificados. En las implementaciones reales, los tokens de portador JWT siempre deben pasarse solo a través de HTTPS.

Con este middleware implementado, los tokens JWT se extraen automáticamente de los encabezados de autorización. A continuación, se deserializan, validan (mediante los valores de los `Audience` parámetros y `Authority`) y se almacenan como información de usuario a la que se hace referencia más adelante mediante las acciones de MVC o los filtros de autorización.

El middleware de autenticación de portador JWT también puede admitir escenarios más avanzados, como el uso de un certificado local para validar un token si la entidad no está disponible. En este escenario, puede especificar un `TokenValidationParameters` objeto en el `JwtBearerOptions` objeto .

Recursos adicionales

- **Uso compartido de cookies entre aplicaciones**
  https://learn.microsoft.com/aspnet/core/security/cookie-sharing
- **Introducción a la identidad**
  https://learn.microsoft.com/aspnet/core/security/authentication/identity
- **Autenticación en dos fases con SMS**
  https://learn.microsoft.com/aspnet/core/security/authentication/2fa
- **Habilitación de la autenticación mediante Facebook, Google y otros proveedores externos**
  https://learn.microsoft.com/aspnet/core/security/authentication/social/
- **Michell Anicas. Introducción a OAuth 2**
  https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2
- **AspNet.Security.OAuth.Providers** (repositorio de GitHub para proveedores de OAuth de ASP.NET)
  https://github.com/aspnet-contrib/AspNet.Security.OAuth.Providers/tree/dev/src
- **IdentityServer4. Documentación oficial**
  https://identityserver4.readthedocs.io/en/latest/

Siguiente anterior

Acerca de la autorización en microservicios y aplicaciones web de .NET

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Después de la autenticación, ASP.NET Core Web API deben autorizar el acceso. Este proceso permite que un servicio ponga las API a disposición de algunos usuarios autenticados, pero no de todos. La autorización se puede realizar en función de los roles de los usuarios o en función de una política personalizada, lo que puede incluir inspeccionar reclamaciones u otras heurísticas.

Restringir el acceso a una ruta de ASP.NET Core MVC es tan fácil como aplicar un atributo Authorize al método de acción (o a la clase del controlador si todas las acciones del controlador requieren autorización), como se muestra en el ejemplo siguiente:

C#

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
```

```
    }
}
```

De forma predeterminada, agregar un atributo Authorize sin parámetros limitará el acceso a los usuarios autenticados para ese controlador o acción. Para restringir aún más la disponibilidad de una API solo para usuarios específicos, el atributo se puede expandir para especificar roles o directivas necesarios que los usuarios deben satisfacer.

Implementación de la autorización basada en roles

ASP.NET Core Identity tiene un concepto integrado de roles. Además de los usuarios, ASP.NET Core Identity almacena información sobre los distintos roles usados por la aplicación y realiza un seguimiento de qué usuarios se asignan a qué roles. Estas asignaciones se pueden cambiar programáticamente con el tipo `RoleManager` que actualiza roles en el almacenamiento persistente, y el tipo `UserManager` que puede conceder o revocar roles a los usuarios.

Si está realizando la autenticación con tokens de portador JWT, el middleware de autenticación de portador JWT de ASP.NET Core rellenará los roles de un usuario según las notificaciones de rol que se encuentren en el token. Para limitar el acceso a una acción o controlador de MVC a los usuarios de roles específicos, puede incluir un parámetro Roles en la anotación Authorize (atributo), como se muestra en el siguiente fragmento de código:

C#

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

En este ejemplo, solo los usuarios de los roles Administrador o PowerUser pueden acceder a las API en el controlador ControlPanel (por ejemplo, la ejecución de la acción SetTime). La API ShutDown está restringida aún más para permitir el acceso solo a los usuarios del rol Administrador.

Para requerir que un usuario esté en varios roles, use varios atributos Authorize, como se muestra en el ejemplo siguiente:

```C#
[Authorize(Roles = "Administrator, PowerUser")]
[Authorize(Roles = "RemoteEmployee ")]
[Authorize(Policy = "CustomPolicy")]
public ActionResult API1 ()
{
}
```

En este ejemplo, para llamar a API1, un usuario debe:

- Tener el rol Administrator *o* PowerUser *y*
- Tener el rol RemoteEmployee *y*
- Satisfacer a un controlador personalizado para la autorización de CustomPolicy.

Implementación de la autorización basada en directivas

Las reglas de autorización personalizadas también se pueden escribir mediante directivas de autorización. En esta sección se proporciona información general. Para obtener más información, consulte el taller de autorización de ASP.NET.

Las políticas de autorización personalizadas se registran en el método Startup.ConfigureServices mediante el método service.AddAuthorization. Este método recibe un delegado que configura un argumento AuthorizationOptions.

```C#
services.AddAuthorization(options =>
{
    options.AddPolicy("AdministratorsOnly", policy =>
        policy.RequireRole("Administrator"));

    options.AddPolicy("EmployeesOnly", policy =>
        policy.RequireClaim("EmployeeNumber"));

    options.AddPolicy("Over21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});
```

Como se muestra en el ejemplo, las directivas se pueden asociar a distintos tipos de requisitos. Una vez registradas las directivas, se pueden aplicar a una acción o controlador pasando el nombre de la directiva como argumento Policy del atributo Authorize (por ejemplo,

`[Authorize(Policy="EmployeesOnly")]`) Las directivas pueden tener varios requisitos, no solo uno (como se muestra en estos ejemplos).

En el ejemplo anterior, la primera llamada a AddPolicy es simplemente una forma alternativa de autorizar por rol. Si `[Authorize(Policy="AdministratorsOnly")]` se aplica a una API, solo los usuarios del rol Administrador podrán acceder a ella.

La segunda AddPolicy llamada demuestra una manera sencilla de requerir que una reclamación determinada deba estar presente para el usuario. El RequireClaim método también toma opcionalmente los valores esperados para la reclamación. Si se especifican valores, el requisito solo se cumple si el usuario tiene una reclamación del tipo correcto y uno de los valores especificados. Si usa el middleware de autenticación de portador JWT, todas las propiedades JWT estarán disponibles como notificaciones de usuario.

La directiva más interesante que se muestra aquí está en el tercer `AddPolicy` método, ya que usa un requisito de autorización personalizado. Mediante el uso de requisitos de autorización personalizados, puede tener un gran control sobre cómo se realiza la autorización. Para que esto funcione, debe implementar estos tipos:

- Tipo requirements que deriva de IAuthorizationRequirement y que contiene campos que especifican los detalles del requisito. En el ejemplo, se trata de un campo de edad para el tipo de ejemplo `MinimumAgeRequirement` .

- Controlador que implementa AuthorizationHandler<TRequirement>, donde T es el tipo de IAuthorizationRequirement que el controlador puede satisfacer. El controlador debe implementar el HandleRequirementAsync método , que comprueba si un contexto especificado que contiene información sobre el usuario cumple el requisito.

Si el usuario cumple el requisito, una llamada a `context.Succeed` indicará que el usuario está autorizado. Si hay varias maneras de que un usuario pueda satisfacer un requisito de autorización, se pueden crear varios controladores.

Además de registrar los requisitos de directiva personalizada con llamadas `AddPolicy`, también debe registrar los controladores de requisito personalizado mediante la inserción de dependencias (`services.AddTransient<IAuthorizationHandler, MinimumAgeHandler>()`).

Un ejemplo de un requisito de autorización personalizado y un controlador para comprobar la antigüedad de un usuario (en función de una `DateOfBirth` notificación) está disponible en la documentación de autorización de ASP.NET Core.

Autorización y API mínimas

ASP.NET admite api mínimas como alternativa a las API basadas en controlador. Las directivas de autorización son la manera recomendada de configurar la autorización para las API mínimas, como se muestra en este ejemplo:

```C#
// Program.cs
builder.Services.AddAuthorizationBuilder()
  .AddPolicy("admin_greetings", policy =>
        policy
            .RequireRole("admin")
            .RequireScope("greetings_api"));

// build the app

app.MapGet("/hello", () => "Hello world!")
  .RequireAuthorization("admin_greetings");
```

Recursos adicionales

- **autenticación principal de ASP.NET**
  https://learn.microsoft.com/aspnet/core/security/authentication/identity
- **autorización principal de ASP.NET**
  https://learn.microsoft.com/aspnet/core/security/authorization/introduction
- **Autorización basada en roles**
  https://learn.microsoft.com/aspnet/core/security/authorization/roles
- **Autorización personalizada basada en directivas**
  https://learn.microsoft.com/aspnet/core/security/authorization/policies
- **Autenticación y autorización en API mínimas **
  https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis/security

Siguiente anterior

Almacenamiento de secretos de aplicación de forma segura durante el desarrollo

17/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Para conectarse con recursos protegidos y otros servicios, las aplicaciones ASP.NET Core suelen necesitar usar cadenas de conexión, contraseñas u otras credenciales que contengan información confidencial. Estos fragmentos confidenciales de información se denominan *secretos*. Se recomienda no incluir secretos en el código fuente y asegurarse de no almacenar secretos en el control de código fuente. En su lugar, debe utilizar el modelo de configuración de ASP.NET Core para leer los secretos desde ubicaciones más seguras.

Debe separar los secretos para acceder a los recursos de desarrollo y almacenamiento provisional de los que se usan para acceder a los recursos de producción, ya que diferentes individuos necesitarán acceso a esos distintos conjuntos de secretos. Para almacenar secretos usados durante el desarrollo, los enfoques comunes son almacenar secretos en variables de entorno o mediante la herramienta Administrador de secretos principales de ASP.NET. Para un almacenamiento más seguro en entornos de producción, los microservicios pueden almacenar secretos en una instancia de Azure Key Vault.

Almacenamiento de secretos en variables de entorno

Una manera de mantener los secretos fuera del código fuente es que los desarrolladores establezcan secretos basados en cadenas como variables de entorno en sus máquinas de desarrollo. Al usar variables de entorno para almacenar secretos con nombres jerárquicos, como los anidados en las secciones de configuración, debe asignar un nombre a las variables para incluir la jerarquía completa de sus secciones, delimitada con dos puntos (:).

Por ejemplo, establecer una variable de entorno `Logging:LogLevel:Default` con el valor `Debug` sería equivalente a un valor de configuración en el siguiente archivo de JSON.

JSON

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Debug"
        }
    }
}
```

Para acceder a estos valores desde las variables de entorno, la aplicación solo tiene que llamar a `AddEnvironmentVariables` en su `ConfigurationBuilder` al construir un objeto `IConfigurationRoot`.

> **Nota**
>
> Las variables de entorno se almacenan normalmente como texto sin formato, por lo que si la máquina o el proceso con las variables de entorno están en peligro, los valores de las variables de entorno serán visibles.

Almacenamiento de secretos con ASP.NET Core Secret Manager

La herramienta ASP.NET Core Secret Manager proporciona otro método para mantener los secretos fuera del código fuente **durante el desarrollo**. Para usar la herramienta Administrador de secretos, instale el paquete **Microsoft.Extensions.Configuration.UserSecrets** en el archivo del proyecto. Una vez que esa dependencia está presente y se ha restaurado, el `dotnet user-secrets` comando se puede usar para establecer el valor de secretos desde la línea de comandos. Estos secretos se almacenarán en un archivo JSON en el directorio de perfiles del usuario (los detalles varían según el sistema operativo), lejos del código fuente.

Los secretos establecidos con la herramienta Secret Manager se organizan según la `UserSecretsId` propiedad del proyecto que usa los secretos. Por lo tanto, debe asegurarse de establecer la propiedad UserSecretsId en el archivo de proyecto, como se muestra en el fragmento de código siguiente. El valor predeterminado es un Identificador Único Global (GUID) asignado por Visual Studio, pero la cadena real no es importante siempre que sea única en tu equipo.

XML

```xml
<PropertyGroup>
    <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

El uso de secretos almacenados con Secret Manager en una aplicación se realiza mediante una llamada a `AddUserSecrets<T>` en la `ConfigurationBuilder` instancia para incluir secretos para la aplicación en su configuración. El parámetro `T` genérico debe ser un tipo del ensamblado al que se aplicó UserSecretId. Por lo general, el uso `AddUserSecrets<Startup>` es correcto.

El `AddUserSecrets<Startup>()` está incluido en las opciones predeterminadas del entorno de desarrollo cuando se utiliza el método `CreateDefaultBuilder` en *Program.cs*.

Siguiente anterior

Uso de Azure Key Vault para proteger secretos en tiempo de producción

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Los secretos almacenados como variables de entorno o almacenados por la herramienta Secret Manager siguen almacenados localmente y sin cifrar en el equipo. Una opción más segura para almacenar secretos es Azure Key Vault, que proporciona una ubicación segura y central para almacenar claves y secretos.

El paquete **Azure.Extensions.AspNetCore.Configuration.Secrets** permite que una aplicación ASP.NET Core lea la información de configuración de Azure Key Vault. Para empezar a usar secretos desde una instancia de Azure Key Vault, siga estos pasos:

1. Registre la aplicación como una aplicación de Azure AD. (Azure AD administra el acceso a los almacenes de claves). Esto se puede hacer a través del Portal de administración de Azure.\

   Como alternativa, si quiere que la aplicación se autentique mediante un certificado en lugar de una contraseña o un secreto de cliente, puede usar el cmdlet new-AzADApplication de PowerShell. El certificado que se registra en Azure Key Vault solo necesita la clave pública. La aplicación usará la clave privada.

2. Proporcione a la aplicación registrada acceso a la bóveda de claves mediante la creación de un nuevo principal de servicio. Puede hacerlo con los siguientes comandos de

PowerShell:

```
PowerShell

$sp = New-AzADServicePrincipal -ApplicationId "<Application ID guid>"
Set-AzKeyVaultAccessPolicy -VaultName "<VaultName>" -ServicePrincipalName
$sp.ServicePrincipalNames[0] -PermissionsToSecrets all -ResourceGroupName "
<KeyVault Resource Group>"
```

3. Incluya el almacén de claves como origen de configuración en la aplicación mediante una llamada al método de extensión AzureKeyVaultConfigurationExtensions.AddAzureKeyVault al crear una IConfigurationRoot instancia.

Tenga en cuenta que, al llamar a `AddAzureKeyVault` necesitará el Id. de la aplicación registrada y a la que se concedió acceso al almacén de claves en los pasos anteriores. O bien, primero puede ejecutar el comando de la CLI de Azure `az login` y, después, usar una sobrecarga de `AddAzureKeyVault` que toma un objeto DefaultAzureCredential en lugar del cliente.

> **Importante**
>
> Se recomienda registrar Azure Key Vault como último proveedor de configuración, por lo que puede invalidar los valores de configuración de los proveedores anteriores.

Recursos adicionales

- **Uso de Azure Key Vault para proteger los secretos de aplicación**
  https://learn.microsoft.com/azure/architecture/multitenant-identity
- **Almacenamiento seguro de secretos de aplicación durante el desarrollo**
  https://learn.microsoft.com/aspnet/core/security/app-secrets
- **Configuración de la protección de datos**
  https://learn.microsoft.com/aspnet/core/security/data-protection/configuration/overview
- **Administración y duración de claves de protección de datos en ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/security/data-protection/configuration/default-settings

Siguiente anterior

Conclusiones clave de arquitectura de microservicios de .NET

18/06/2025

Sugerencia

Este contenido es un extracto del libro electrónico, ".NET Microservices Architecture for Containerized .NET Applications" (Arquitectura de microservicios de .NET para aplicaciones de .NET contenedorizadas), disponible en Documentación de .NET o como un PDF descargable y gratuito que se puede leer sin conexión.

Descargar PDF

Como resumen y conclusiones clave, las siguientes son las conclusiones más importantes de esta guía.

Ventajas de usar contenedores. Las soluciones basadas en contenedores proporcionan importantes ahorros de costos porque ayudan a reducir los problemas de implementación causados por dependencias con errores en entornos de producción. Los contenedores mejoran significativamente las operaciones de DevOps y producción.

Los contenedores serán omnipresentes. Los contenedores basados en Docker se están convirtiendo en el estándar de facto del sector, compatibles con proveedores clave en los ecosistemas de Windows y Linux, como Microsoft, Amazon AWS, Google e IBM. Docker probablemente pronto estará omnipresente tanto en la nube como en los centros de datos locales.

Los contenedores como una unidad de implementación. Un contenedor de Docker se está convirtiendo en la unidad estándar de implementación para cualquier aplicación o servicio basado en servidor.

Microservicios. La arquitectura de microservicios se está convirtiendo en el enfoque preferido para aplicaciones críticas distribuidas y grandes o complejas basadas en muchos subsistemas

independientes en forma de servicios autónomos. En una arquitectura basada en microservicios, la aplicación se crea como una colección de servicios desarrollados, probados, versionados, implementados y escalados de forma independiente. Cada servicio puede incluir cualquier base de datos autónoma relacionada.

Diseño controlado por dominio y SOA. Los patrones de arquitectura de microservicios derivan de la arquitectura orientada a servicios (SOA) y el diseño controlado por dominio (DDD). Al diseñar y desarrollar microservicios para entornos con necesidades y reglas empresariales en constante evolución, es importante tener en cuenta los enfoques y patrones de DDD.

Desafíos de los microservicios. Los microservicios ofrecen muchas funcionalidades eficaces, como la implementación independiente, los límites sólidos del subsistema y la diversidad tecnológica. Sin embargo, también generan muchos nuevos desafíos relacionados con el desarrollo de aplicaciones distribuidas, como modelos de datos fragmentados e independientes, comunicación resistente entre microservicios, coherencia final y complejidad operativa que resulta de agregar información de registro y supervisión de varios microservicios. Estos aspectos presentan un nivel de complejidad mucho mayor que una aplicación monolítica tradicional. Como resultado, solo los escenarios específicos son adecuados para aplicaciones basadas en microservicios. Entre ellas se incluyen aplicaciones grandes y complejas con varios subsistemas en evolución. En estos casos, vale la pena invertir en una arquitectura de software más compleja, ya que proporcionará una mejor agilidad a largo plazo y mantenimiento de aplicaciones.

Contenedores para cualquier aplicación. Los contenedores son cómodos para los microservicios, pero también pueden ser útiles para las aplicaciones monolíticas basadas en .NET Framework tradicional, cuando se usan contenedores de Windows. Las ventajas de usar Docker, como resolver muchos problemas de implementación a producción y proporcionar entornos de desarrollo y pruebas de última generación, se aplican a muchos tipos diferentes de aplicaciones.

CLI frente a IDE. Con las herramientas de Microsoft, puede desarrollar aplicaciones .NET en contenedor mediante su enfoque preferido. Puede desarrollar con una CLI y un entorno basado en editor mediante la CLI de Docker y Visual Studio Code. También puedes utilizar un enfoque basado en el IDE con Visual Studio y sus características exclusivas para Docker, como la depuración de múltiples contenedores.

Aplicaciones en la nube resistentes. En los sistemas basados en la nube y los sistemas distribuidos en general, siempre existe el riesgo de errores parciales. Dado que los clientes y servicios son procesos independientes (contenedores), es posible que un servicio no pueda responder de forma oportuna a la solicitud de un cliente. Por ejemplo, un servicio podría estar inactivo debido a un error parcial o por mantenimiento; es posible que el servicio se sobrecargue y responda lentamente a las solicitudes; o puede que no sea accesible durante un

breve tiempo debido a problemas de red. Por lo tanto, una aplicación basada en la nube debe adoptar esos errores y tener una estrategia en vigor para responder a esos errores. Estas estrategias pueden incluir políticas de reintento (reenvío de mensajes o intentos de solicitud) e implementar patrones de corte de circuito para evitar la carga exponencial de solicitudes repetidas. Básicamente, las aplicaciones basadas en la nube deben tener mecanismos resistentes, ya sea en función de la infraestructura de la nube o personalizados, como los de alto nivel proporcionados por orquestadores o buses de servicio.

Seguridad. Nuestro mundo moderno de contenedores y microservicios puede exponer nuevas vulnerabilidades. Hay varias maneras de implementar la seguridad básica de las aplicaciones, en función de la autenticación y la autorización. Sin embargo, la seguridad de los contenedores debe tener en cuenta componentes clave adicionales que dan lugar a aplicaciones intrínsecamente más seguras. Un elemento fundamental de la creación de aplicaciones más seguras es tener una forma segura de comunicarse con otras aplicaciones y sistemas, algo que a menudo requiere credenciales, tokens, contraseñas y como, por lo general, conocido como secretos de aplicación. Cualquier solución segura debe seguir los procedimientos recomendados de seguridad, como cifrar secretos mientras están en tránsito y en reposo, e impedir que los secretos se filtren cuando la aplicación final lo consuma. Esos secretos deben almacenarse y mantenerse de forma segura, como cuando se usa Azure Key Vault.

Orquestadores. Los orquestadores basados en contenedores, como Azure Kubernetes Service y Azure Service Fabric, forman parte fundamental de cualquier aplicación significativa basada en microservicios y contenedores. Estas aplicaciones implican una complejidad elevada, demandas de escalabilidad y pasan por una evolución constante. En esta guía se han introducido orquestadores y su rol en soluciones basadas en microservicios y basadas en contenedores. Si las necesidades de su aplicación están haciendo que se incline hacia aplicaciones albergadas en contenedores complejas, le resultará útil buscar recursos adicionales para obtener más información sobre herramientas de orquestación de aplicaciones.

Anterior