

Contents

- Microservicios de .NET: Arquitectura para aplicaciones .NET en contenedor
 - Introducción a Containers y Docker
 - ¿Qué es Docker?
 - Terminología de Docker
 - Contenedores, imágenes y registros de Docker
 - Selección entre .NET Core y .NET Framework para contenedores de Docker
 - Orientación general
 - Cuándo elegir .NET Core para contenedores de Docker
 - Cuándo elegir .NET Framework para contenedores de Docker
 - Tabla de decisiones: versiones de .NET Framework para su uso con Docker
 - Selección del sistema operativo de destino con contenedores de .NET
 - Imágenes oficiales de Docker de .NET
 - Diseñar la arquitectura de aplicaciones basadas en contenedores y microservicios
 - Incluir en un contenedor aplicaciones monolíticas
 - Estado y datos en aplicaciones de Docker
 - Arquitectura orientada a servicios
 - Arquitectura de microservicios
 - Propiedad de los datos por microservicio
 - Arquitectura lógica frente a arquitectura física
 - Desafíos y soluciones de la administración de datos distribuidos
 - Identificación de los límites del modelo de dominio para cada microservicio
 - Comunicación directa de cliente a microservicio frente al patrón de puerta de enlace de API
 - Comunicación en una arquitectura de microservicio
 - Comunicación asincrónica basada en mensajes
 - Creación, desarrollo y control de versiones de los contratos y las API de microservicio
 - Direccionabilidad de microservicios y el Registro del servicio
 - Crear una interfaz de usuario compuesta en función de los microservicios, incluidos

la forma y el diseño visual de la interfaz de usuario generados por varios microservicios

Resistencia y alta disponibilidad en microservicios

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

Proceso de desarrollo de aplicaciones basadas en Docker

Flujo de trabajo de desarrollo para aplicaciones de Docker

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

Diseño de una aplicación orientada a microservicios

Creación de un microservicio CRUD sencillo controlado por datos

Definir una aplicación de varios contenedores con docker-compose.yml

Uso de un servidor de bases de datos que se ejecuta como un contenedor

Implementación de comunicación basada en eventos entre microservicios (eventos de integración)

Implementación de un bus de eventos con RabbitMQ para el entorno de desarrollo o de prueba

Suscripción a eventos

Probar aplicaciones web y servicios ASP.NET Core

Implementar tareas en segundo plano en microservicios con IHostedService

Implementación de puertas de enlace de API con Ocelot

Abordar la complejidad empresarial en un microservicio con patrones DDD y CQRS

Aplicar los patrones CQRS y DDD simplificados en un microservicio

Aplicar enfoques CQRS y CQS en un microservicio DDD en eShopOnContainers

Implementación de lecturas/consultas en un microservicio CQRS

Diseño de un microservicio orientado a un DDD

Diseño de un modelo de dominio de microservicio

Implementar un modelo de dominio de microservicio con .NET Core

Seedwork (interfaces y clases base reutilizables para su modelo de dominio)

Implementar objetos de valor

Uso de las clases de enumeración en lugar de los tipos de enumeración

Diseñar las validaciones en el nivel de modelo de dominio

Validación del lado cliente (validación de los niveles de presentación)

Eventos de dominio: diseño e implementación

Diseño de la capa de persistencia de infraestructura

Implementación del nivel de persistencia de la infraestructura con Entity Framework Core

Uso de bases de datos NoSQL como una infraestructura de persistencia

Diseñar el nivel de aplicación de microservicios y la API web

Implementación del nivel de aplicación de microservicios mediante la API web

Implementar aplicaciones resistentes

Controlar errores parciales

Estrategias para tratar errores parciales

Implementar reintentos con retroceso exponencial

Implementar conexiones SQL resistentes de Entity Framework Core

Exploración de reintentos de llamada HTTP personalizados con retroceso exponencial

Uso de HttpClientFactory para implementar solicitudes HTTP resistentes

Implementación de reintentos de llamada HTTP con retroceso exponencial con Polly

Implementación del patrón de interruptor

Seguimiento de estado

Proteger microservicios y aplicaciones web de .NET

Acerca de la autorización en microservicios y aplicaciones web de .NET

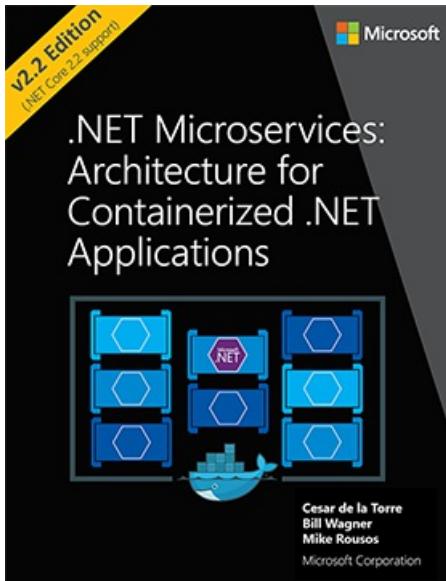
Almacenar secretos de aplicación de forma segura durante el desarrollo

Usar Azure Key Vault para proteger secretos en tiempo de producción

Puntos clave

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

31/10/2019 • 14 minutes to read • [Edit Online](#)



EDICIÓN v2.2: actualizada a ASP.NET Core 2.2

Esta guía es una introducción al desarrollo de aplicaciones basadas en microservicios y a su administración mediante contenedores. En ella se trata el diseño de la arquitectura y los métodos de implementación con .NET Core y contenedores de Docker.

Para que sea más fácil empezar a trabajar, la guía se centra en una aplicación de referencia en contenedor y basada en microservicios que puede explorar. La aplicación de referencia está disponible en el repositorio de GitHub [eShopOnContainers](#).

Vínculos de acción

- Descargue este libro electrónico en el formato que prefiera (solo disponible en inglés): | [PDF](#) | [MOBI](#) | [EPUB](#) |
- Clone o bifurque la aplicación de referencia [eShopOnContainers](#) en GitHub
- Vea el [vídeo de introducción en Channel 9](#)
- Conozca la [arquitectura de microservicios](#) inmediatamente

Introducción

Las empresas cada vez ahorran más costos, resuelven más problemas de implementación y mejoran más las operaciones de DevOps y producción mediante el uso de contenedores. Microsoft ha lanzado recientemente innovaciones en los contenedores de Windows y Linux con la creación de productos como Azure Kubernetes Service y Azure Service Fabric, contando además con la colaboración de líderes del sector como Docker, Mesosphere y Kubernetes. Estos productos ofrecen soluciones de contenedores que ayudan a las empresas a compilar e implementar aplicaciones a velocidad y escala de nube, sea cual sea la plataforma o las herramientas que hayan elegido.

Docker se está convirtiendo en el estándar de facto del sector de los contenedores, ya que es compatible con los proveedores más importantes del ecosistema de Windows y Linux. (Microsoft es uno de los principales proveedores de nube que admite Docker). En el futuro, Docker probablemente estará omnipresente en todos los centros de datos en la nube o locales.

Además, la arquitectura de [microservicios](#) se está convirtiendo en un enfoque fundamental 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 por separado.

Acerca de esta guía

Esta guía es una introducción al desarrollo de aplicaciones basadas en microservicios y a su administración mediante contenedores. En ella se trata el diseño de la arquitectura y los métodos de implementación con .NET Core y contenedores de Docker. Para que sea más fácil empezar a trabajar con contenedores y microservicios, la guía se centra en una aplicación de referencia en contenedor y basada en microservicios que puede explorar. Esta misma aplicación de ejemplo está disponible en el repositorio de GitHub [eShopOnContainers](#).

Esta guía incluye el desarrollo fundamental y una guía de arquitectura principalmente en el nivel del entorno de desarrollo con especial hincapié en dos tecnologías: Docker y .NET Core. Nuestra intención es que lea esta guía cuando reflexione sobre el diseño de las aplicaciones sin centrarse en la infraestructura (en la nube o local) de su entorno de producción. Tomará decisiones sobre la infraestructura más adelante, cuando cree aplicaciones listas para la producción. Por lo tanto, esta guía está diseñada para ser independiente de la infraestructura y centrarse en el desarrollo y el entorno.

Una vez que haya estudiado esta guía, el siguiente paso que debería dar es obtener información sobre los microservicios listos para la producción en Microsoft Azure.

Versión

Esta guía se ha revisado para tratar la versión **.NET Core 2.2**, además de muchas actualizaciones adicionales relacionadas con la misma "oleada" de tecnologías (es decir, tecnologías de Azure y tecnologías adicionales de terceros) que coincidan en el tiempo con .NET Core 2.2. Este es el motivo por el que la versión del libro se ha actualizado también a la versión **2.2**.

Aspectos no tratados en esta guía

Esta guía no se centra en el ciclo de vida de la aplicación, DevOps, las canalizaciones CI/CD ni el trabajo de equipo. La guía complementaria [Containerized Docker Application Lifecycle with Microsoft Platform and Tools](#) (Ciclo de vida de aplicaciones de Docker en contenedor con la plataforma y herramientas de Microsoft) se centra en esas cuestiones. La guía actual tampoco proporciona detalles de implementación de la infraestructura de Azure, como información sobre orquestadores específicos.

Recursos adicionales

- **Containerized Docker Application Lifecycle with Microsoft Platform and Tools** (Ciclo de vida de aplicaciones de Docker en contenedor con la plataforma y las herramientas de Microsoft; libro electrónico descargable)
<https://aka.ms/dockerlifecyclebook>

Destinatarios de esta guía

Esta guía se ha escrito para desarrolladores y arquitectos de soluciones que no están familiarizados con el desarrollo de aplicaciones basado en Docker y la arquitectura basada en microservicios. Esta guía será de su interés si quiere obtener información sobre cómo crear arquitecturas, diseñar e implementar aplicaciones de prueba de concepto con tecnologías de desarrollo de Microsoft (con un hincapié especial en .NET Core) y con

contenedores de Docker.

También le resultará útil si es el responsable de tomar decisiones técnicas (por ejemplo, un arquitecto empresarial) y necesita una descripción de la arquitectura y la tecnología antes de decidir qué enfoque tomar para el diseño de aplicaciones distribuidas tanto nuevas como modernas.

Cómo usar esta guía

La primera parte de esta guía presenta los contenedores de Docker, describe cómo elegir entre .NET Core y .NET Framework como marco de desarrollo y proporciona una visión general de los microservicios. Este contenido está destinado a arquitectos y responsables de la toma de decisiones técnicas que quieren obtener información general pero que no necesitan centrarse en los detalles de la implementación de código.

La segunda parte de la guía comienza con la sección [Proceso de desarrollo de aplicaciones basadas en Docker](#). Se centra en los patrones de desarrollo y microservicios usados en la implementación de las aplicaciones que utilizan .NET Core y Docker. Esta sección será de gran interés para los desarrolladores y los arquitectos que quieran centrarse en el código, en los patrones y los detalles de implementación.

Aplicación de referencia relacionada de microservicios y basada en contenedor: eShopOnContainers

La aplicación eShopOnContainers es una aplicación de referencia de código abierto para .NET Core y microservicios que está diseñada para implementarse mediante contenedores de Docker. La aplicación consta de varios subsistemas, incluidos varios front-end de interfaz de usuario de almacén electrónico (una aplicación web de MVC, una SPA web y una aplicación móvil nativa). También incluye microservicios y contenedores de back-end para todas las operaciones del lado servidor necesarias.

El propósito de la aplicación es presentar patrones arquitectónicos. **NO SE TRATA DE UNA PLANTILLA LISTA PARA PRODUCCIÓN** para iniciar aplicaciones del mundo real. De hecho, la aplicación se encuentra en un estado beta permanente, ya que también se usa para probar nuevas tecnologías potencialmente interesantes a medida que aparecen.

Envíenos sus comentarios.

Hemos creado esta guía para ayudarle a entender la arquitectura de aplicaciones y microservicios en contenedor en .NET. La guía y la aplicación de referencia relacionada irán evolucionando, por lo que le agradecemos sus comentarios. Si tiene comentarios sobre cómo se puede mejorar esta guía, envíelos a:

dotnet-architecture-ebooks-feedback@service.microsoft.com

Créditos

Coautores:

Cesar de la Torre, administrador de programas senior del equipo de producto de .NET, Microsoft Corp.

Bill Wagner, desarrollador de contenido senior de C+E, Microsoft Corp.

Mike Rousos, ingeniero de software principal del equipo de CAT de la división de desarrollo, Microsoft

Editores:

Mike Pope

Steve Hoag

Participantes y revisores:

Jeffrey Richter, ingeniero de software asociado del equipo de Azure, Microsoft

Jimmy Bogard, arquitecto jefe de Headspring

Udi Dahan, fundador y director general de Particular Software

Jimmy Nilsson, cofundador y director general de Factor10

Glenn Condron, director de programas sénior del equipo de ASP.NET

Mark Fussell, responsable principal de administración de programas del equipo de Azure Service Fabric, Microsoft

Diego Vega, responsable de administración de programas del equipo de Entity Framework, Microsoft

Barry Dorrans, director de programas de seguridad sénior

Rowan Miller, director de programas sénior, Microsoft

Ankit Asthana, director principal de administración de programas del equipo de .NET, Microsoft

Scott Hunter, director asociado de administración de programas del equipo de .NET, Microsoft

Nish Anil, director de administración de programas, equipo de .NET, Microsoft

Dylan Reisenberger, arquitecto y responsable de desarrollo de Polly

Steve "ardalis" Smith: instructor y arquitecto de software de [Ardalis.com](https://ardalis.com)

Cooper Ian, arquitecto de codificación de Brighter

Unai Zorrilla, arquitecto y responsable de desarrollo de Plain Concepts

Eduard Tomas, responsable de desarrollo de Plain Concepts

Ramon Tomas, desarrollador de Plain Concepts

David Sanz, desarrollador de Plain Concepts

Javier Valero, director de operaciones de Grupo Solutio

Pierre Millet, consultor sénior, Microsoft

Michael Friis, administrador de productos de Docker Inc.

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

Miguel Veloso, consultor sénior en Turing Challenge

Copyright

DESCARGA disponible en: <https://aka.ms/microservicesebook>

PUBLICADO POR

Equipos de producto de la División de desarrolladores de Microsoft, .NET y Visual Studio

División de Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2019 de Microsoft Corporation

Todos los derechos reservados. No se puede reproducir ni transmitir de ninguna forma ni por ningún medio ninguna parte del contenido de este libro sin la autorización por escrito del publicador.

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

Algunos ejemplos descritos aquí se proporcionan únicamente con fines ilustrativos y son ficticios. No debe deducirse ninguna asociación ni conexión reales.

Microsoft y las marcas comerciales indicadas en <https://www.microsoft.com> en la página web "Marcas comerciales" pertenecen al 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. Se usa con permiso.

El resto de marcas y logotipos pertenece a sus respectivos propietarios.

SIGUIENTE

Introducción a Containers y Docker

12/11/2019 • 3 minutes to read • [Edit Online](#)

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 puede probarse como una unidad e implementarse como una instancia de imagen de contenedor en el sistema operativo (SO) host.

Del mismo modo que los contenedores de mercancías permiten su transporte por barco, tren o camión independientemente de la carga de su interior, los contenedores de software actúan como una unidad estándar de implementación de software que puede contener diferentes dependencias y código. De esta manera, la inclusión del software en contenedor permite a los desarrolladores y los profesionales de TI implementarlo en entornos con pocas modificaciones o ninguna en absoluto.

Los contenedores también aíslan las aplicaciones entre sí en un sistema operativo compartido. Las aplicaciones en contenedor 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 o un servicio al completo, como se muestra en la figura 2-1. En este ejemplo, el host de Docker es un host de contenedor, y App 1, App 2, Svc 1 y Svc 2 son aplicaciones o servicios en contenedor.

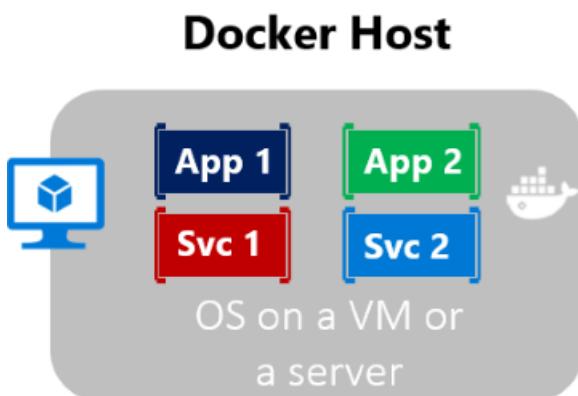


Figura 2-1. Varios contenedores ejecutándose en un host de contenedor.

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 el punto de vista de la aplicación, la creación de instancias de una imagen (la creación de un contenedor) es similar a la creación de instancias de un proceso como un servicio o una aplicación web. Pero con fines de confiabilidad, cuando ejecute varias instancias de la misma imagen en varios servidores host, seguramente le interesará que cada contenedor (instancia de imagen) se ejecute en un servidor host o máquina virtual diferente en dominios de error distintos.

En resumen, los contenedores ofrecen las ventajas del aislamiento, la portabilidad, la agilidad, la escalabilidad y el control a lo largo de todo el flujo de trabajo del ciclo de vida de la aplicación. La ventaja más importante es el aislamiento del entorno que se proporciona entre el desarrollo y las operaciones.

¿Qué es Docker?

25/11/2019 • 11 minutes to read • [Edit Online](#)

Docker es un [proyecto de código abierto](#) para automatizar la implementación de aplicaciones como contenedores portátiles y autosuficientes que se pueden ejecutar en la nube o localmente. Docker es también una [empresa](#) que promueve e impulsa esta tecnología, en colaboración con proveedores de la nube, Linux y Windows, incluido Microsoft.

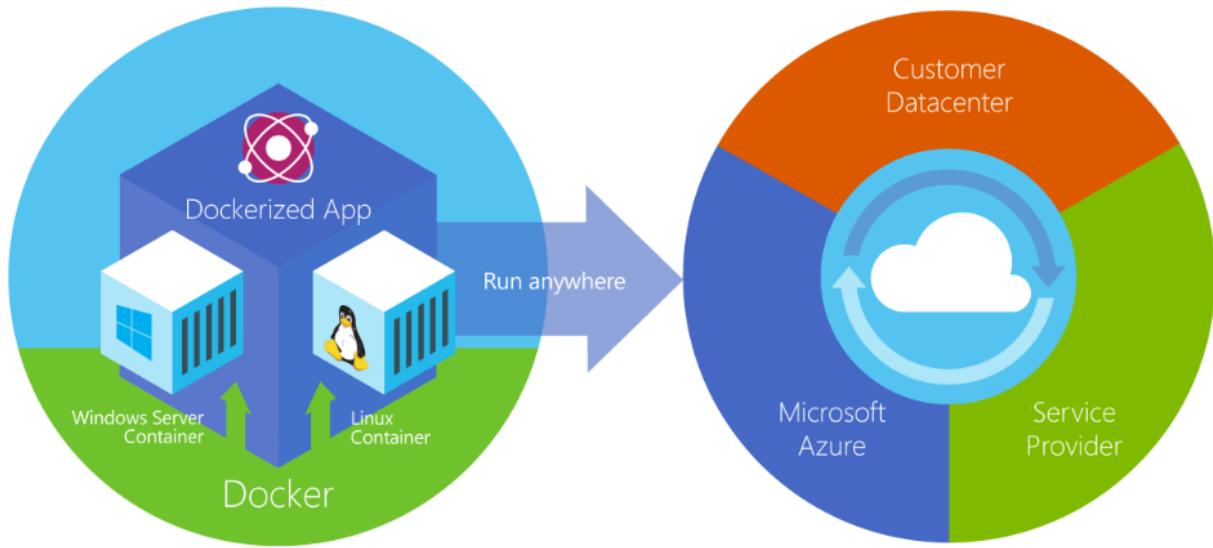


Figura 2-2. Docker implementa contenedores en todas las capas de la nube híbrida.

Los contenedores de Docker se pueden ejecutar en cualquier lugar, a nivel local en el centro de datos de cliente, en un proveedor de servicios externo o en la nube, en Azure. Los contenedores de imagen de Docker se pueden ejecutar de forma nativa en Linux y Windows. Sin embargo, las imágenes de Windows solo pueden ejecutarse en hosts de Windows y las imágenes de Linux pueden ejecutarse en hosts de Linux y hosts de Windows (con una máquina virtual Linux de Hyper-V, hasta el momento), donde host significa 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 en que se implementan imágenes de Docker, incluidas la aplicación y sus dependencias. Los desarrolladores que trabajan en Linux o en Mac usan un host de Docker basado en Linux y pueden crear imágenes solo para contenedores de Linux. (Los desarrolladores que trabajan en Mac pueden editar código o ejecutar la CLI de Docker en macOS, pero en el momento de redactar este artículo, los contenedores no se ejecutan directamente en macOS). Los desarrolladores que trabajan en Windows pueden crear imágenes para contenedores de Windows o Linux.

Para hospedar contenedores en entornos de desarrollo y proporcionar herramientas de desarrollador adicionales, Docker entrega [Docker Community Edition \(CE\)](#) para Windows o macOS. Estos productos instalan la máquina virtual necesaria (el host de Docker) para hospedar los contenedores. Docker también pone a disposición [Docker Enterprise Edition \(EE\)](#), que está diseñado para el desarrollo empresarial y se usa en los equipos de TI que crean, envían y ejecutan aplicaciones críticas para la empresa en producción.

Para ejecutar [contenedores de Windows](#), hay dos tipos de tiempos de ejecución:

- Los contenedores de Windows ofrecen aislamiento de aplicaciones a través de tecnología de aislamiento de proceso y de espacio de nombres. Un contenedor de Windows Server comparte el kernel con el host de contenedor y con todos los contenedores que se ejecutan en el host.

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

Las imágenes de estos contenedores se crean y funcionan de la misma manera. La diferencia radica en cómo se crea el contenedor desde la imagen ejecutando un contenedor de Hyper-V que requiere un parámetro adicional. Para más información, vea [Contenedores de Hyper-V](#).

Comparación de los contenedores de Docker con las máquinas virtuales

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

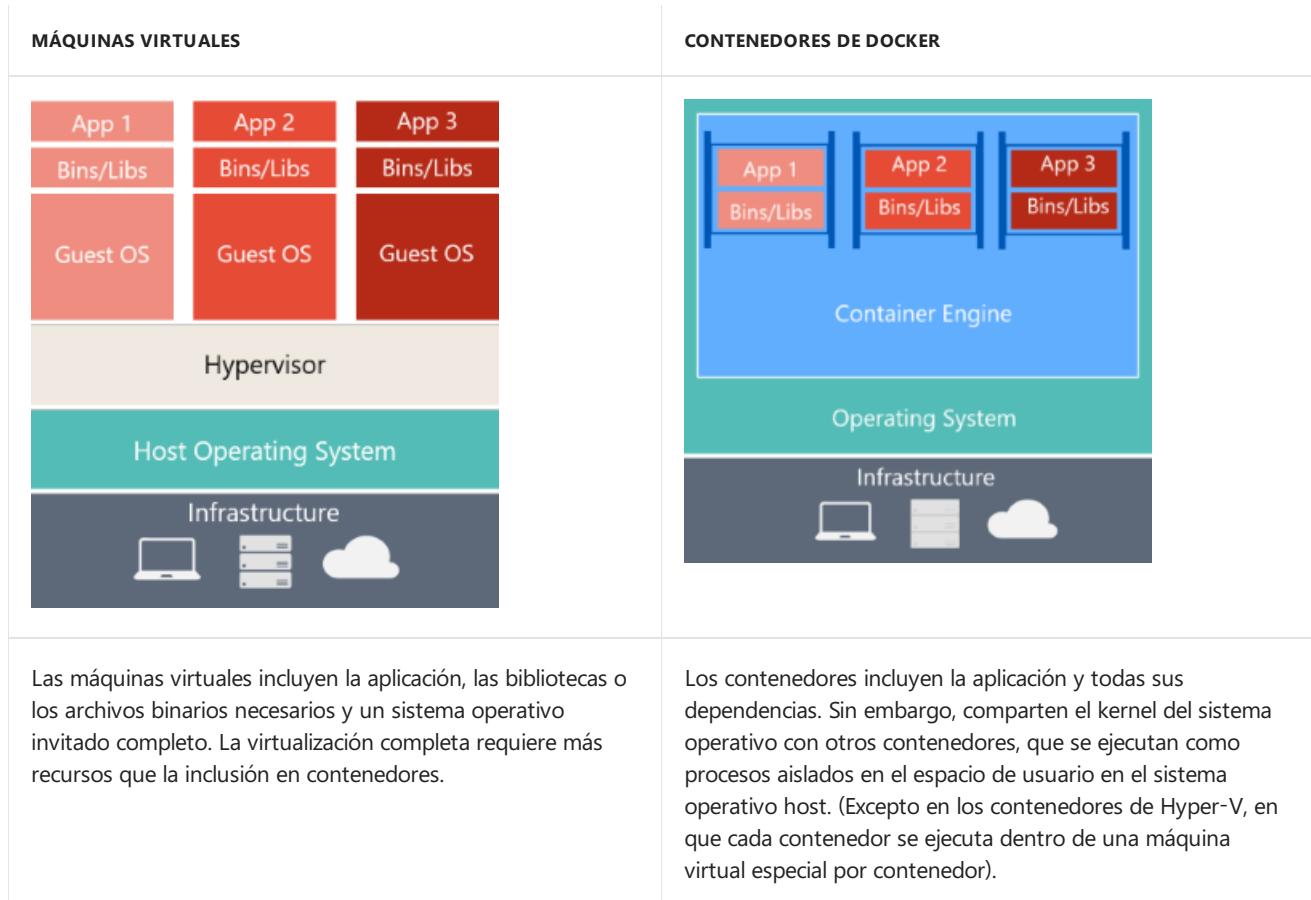


Figura 2-3. Comparación de las máquinas virtuales tradicionales con los contenedores de Docker

Para las máquinas virtuales, hay tres niveles de base en el servidor host, de manera ascendente: infraestructura, sistema operativo host y un hipervisor y, encima de todo eso, cada máquina virtual tiene su propio sistema operativo y todas las bibliotecas necesarias. En el caso de 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 con el uso compartido de los servicios del sistema operativo de base.

Dado que los contenedores requieren muchos menos recursos (por ejemplo, no necesitan un sistema operativo completo), se inician rápidamente y son fáciles de implementar. Esto permite tener una mayor densidad, lo que significa que se pueden ejecutar más servicios en la misma unidad de hardware, reduciendo así los costos.

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

El objetivo principal de una imagen es que hace que el entorno (dependencias) sea el mismo entre las 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 un servicio e implementarlo de forma confiable y reproducible. Podría decir que Docker no solo es una tecnología, sino también una filosofía y un proceso.

Al usar Docker, no escuchará a los desarrolladores decir "Si funciona en mi máquina, ¿por qué no en producción?". Pueden decir simplemente "Se ejecuta en Docker", porque la aplicación de Docker empaquetada puede ejecutarse en cualquier entorno de Docker compatible, y se ejecuta de la forma prevista en todos los destinos de implementación (como desarrollo, control de calidad, ensayo y producción).

Una analogía simple

Quizás una analogía simple puede ayudar a entender el concepto básico de Docker.

Vamos a remontarnos a la década de 1950 por un momento. No había ningún procesador de texto y las fotocopiadoras se utilizaban en todas partes y de todo tipo.

Imagine que es responsable de enviar lotes de cartas, según proceda, para enviarlas por correo a los clientes en papel y sobres reales, que se entregarán físicamente en la dirección postal de cada cliente (entonces no existía el correo electrónico).

En algún momento, se da cuenta de que las cartas no son más que una composición de un conjunto grande de párrafos, que se eligen y ordenan según proceda, según el propósito de la carga, por lo que diseña un sistema para emitir cartas rápidamente, esperando conseguir una increíble mejora.

El sistema es simple:

1. Comience con un conjunto de hojas transparentes que contienen un párrafo.
2. Para emitir un conjunto de cartas, elija las hojas con los párrafos necesarios, apílelas y alinéelas para que queden y se lean bien.
3. Por último, colóquelas en la fotocopiadora y presione inicio para producir tantas cartas como sean necesarias.

Por tanto, para simplificar, esa es la idea principal de Docker.

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

Por lo tanto, al "Examinar" el sistema de archivos después de que se ha copiado la capa, verá todos los archivos, incluida la capa cuando se instaló el programa.

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

De forma similar, puede pensar en un contenedor como el "equipo" con el disco duro de imagen instalado. El contenedor, como un equipo, se puede apagar o desactivar.

[ANTERIOR](#)

[SIGUIENTE](#)

Terminología de Docker

04/11/2019 • 11 minutes to read • [Edit Online](#)

En esta sección se enumeran los términos y las definiciones que debe conocer antes de profundizar en el uso de Docker. Para consultar más definiciones, lea el amplio [glosario](#) que Docker proporciona.

Imagen de contenedor: un paquete con todas las dependencias y la información necesaria para crear un contenedor. Una imagen incluye todas las dependencias (por ejemplo, los marcos), así como la configuración de implementación y ejecución que usará el runtime de un contenedor. Normalmente, una imagen se deriva de varias imágenes base que son capas que se apilan unas encima de otras para formar el sistema de archivos del contenedor. Una vez que se crea una imagen, esta es inmutable.

Dockerfile: un archivo de texto que contiene instrucciones sobre cómo crear una imagen de Docker. Es como un script por lotes, la primera línea indica la imagen de base con la que comenzar y, a continuación, siga las instrucciones para instalar programas necesarios, copiar archivos y así sucesivamente, hasta que llegue al entorno de trabajo que necesita.

Compilación: la acción de crear una imagen de contenedor basada en la información y el contexto que proporciona su Dockerfile, así como archivos adicionales en la carpeta en que se crea la imagen. Puede crear imágenes con el comando de **docker build** de Docker.

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

Volúmenes: ofrece un sistema de archivos grabable que el contenedor puede usar. Puesto 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, encima de 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 por capas, que no es más que el sistema de archivos habitual. Los volúmenes residen en el sistema host y los administra Docker.

Etiqueta: una marca o una 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 de varias fases: es una característica, desde Docker 17.05 o versiones posteriores, que ayuda a reducir el tamaño de las imágenes finales. En pocas palabras, con la compilación de varias fases se puede usar, por ejemplo, una imagen base grande, que contiene el SDK, para compilar y publicar la aplicación y, a continuación, usando la carpeta de publicación con una imagen base pequeña solo en tiempo de ejecución, para generar una imagen final mucho más pequeña.

Repositorio: una colección de imágenes de Docker relacionadas, etiquetadas 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 pesada), una imagen que solo contiene runtimes (más ligera), etcétera. Estas 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: servicio que proporciona acceso a los repositorios. El registro predeterminado para la mayoría de las imágenes públicas es [Docker Hub](#) (propiedad de Docker como una organización). Normalmente, un registro contiene repositorios procedentes de varios equipos. Las empresas suelen tener registros privados para almacenar y administrar imágenes que han creado. Azure Container Registry es otro ejemplo.

Imagen multiarquitectura: para escenarios multiarquitectura, es una característica que simplifica la selección de la imagen apropiada, según la plataforma donde se está ejecutando Docker, por ejemplo, cuando un archivo Dockerfile solicita una imagen base **DESDE mcr.microsoft.com/dotnet/core/sdk:2.2** del registro que realmente obtiene **2.2-sdk-nanoserver-1709**, **2.2-sdk-nanoserver-1803**, **2.2-sdk-nanoserver-1809** o **2.2-sdk-stretch**, 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: recurso público para trabajar con imágenes de Docker y sus componentes en Azure. Esto proporciona un registro cercano a las implementaciones en Azure que le proporciona control sobre el acceso, lo que le permite usar los grupos y los permisos de Active Directory.

Docker Trusted Registry (DTR) : servicio del registro de Docker (ofrecido por Docker) que se puede instalar de forma local, por lo que se encuentra en el centro de datos y la red de la organización. Es ideal para imágenes privadas que deben administrarse dentro de la empresa. Docker Trusted Registry se incluye como parte del producto Docker Datacenter. Para más información, vea [Docker Trusted Registry \(DTR\)](#).

Docker Community Edition (CE) : herramientas de desarrollo para Windows y MacOS para compilar, ejecutar y probar contenedores localmente. Docker CE para Windows proporciona entornos de desarrollo para contenedores de Windows y Linux. El host de Docker de Linux en Windows se basa en una máquina virtual [Hyper-V](#). El host para los contenedores de Windows se basa directamente en Windows. Docker CE para Mac se basa en el marco del hipervisor de Apple y el [hipervisor xhyve](#), lo que proporciona una máquina virtual de host de Docker de Linux en Mac OS X. Docker CE para Windows y para Mac sustituye a Docker Toolbox, que se basaba en Oracle VirtualBox.

Docker Enterprise Edition (EE) : versión a escala empresarial de las herramientas de Docker para el desarrollo de Linux y Windows.

Compose: herramienta de línea de comandos y formato de archivo YAML con metadatos para definir y ejecutar aplicaciones de varios contenedores. Primero se define una sola aplicación basada en varias imágenes con uno o más archivos .yml que pueden invalidar los valores según el entorno. Después de crear las definiciones, puede implementar toda la aplicación de varios contenedores con un solo comando (composición de docker) que crea un contenedor por imagen en el host de Docker.

Clúster: colección de hosts de Docker que se expone como si fuera un solo host de Docker virtual, de manera que la aplicación se puede escalar a varias instancias de los servicios repartidos entre varios hosts del clúster. Los clústeres de Docker se pueden crear con Kubernetes, Azure Service Fabric, Docker Swarm y Mesosphere DC/OS.

Orquestador: herramienta que simplifica la administración de clústeres y hosts de Docker. Los orquestadores permiten administrar las imágenes, los contenedores y los hosts a través de una interfaz de línea de comandos (CLI) o una interfaz gráfica de usuario. Puede administrar las redes de contenedor, las configuraciones, el equilibrio de carga, la detección de servicios, la alta disponibilidad, la configuración del host de Docker y muchas cosas más. Un orquestador se encarga de ejecutar, distribuir, escalar y reparar las cargas de trabajo a través de una colección de nodos. Normalmente, los productos del orquestador son los mismos que proporcionan infraestructura de clúster, como Kubernetes y Azure Service Fabric, entre otras ofertas del mercado.

Contenedores, imágenes y registros de Docker

11/11/2019 • 3 minutes to read • [Edit Online](#)

Al usar Docker, un desarrollador crea una aplicación o un servicio y lo empaqueta, junto con 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. Inicialmente, los contenedores se prueban en un entorno de desarrollo o un PC.

Los desarrolladores deben almacenar las imágenes en un registro, que actúa como una biblioteca de imágenes y es necesario cuando se implementa en orquestadores de producción. Docker mantiene un registro público a través de [Docker Hub](#); otros proveedores ofrecen registros para distintas 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 de Docker con otros componentes. También se muestran las diversas ofertas de registro de los proveedores.

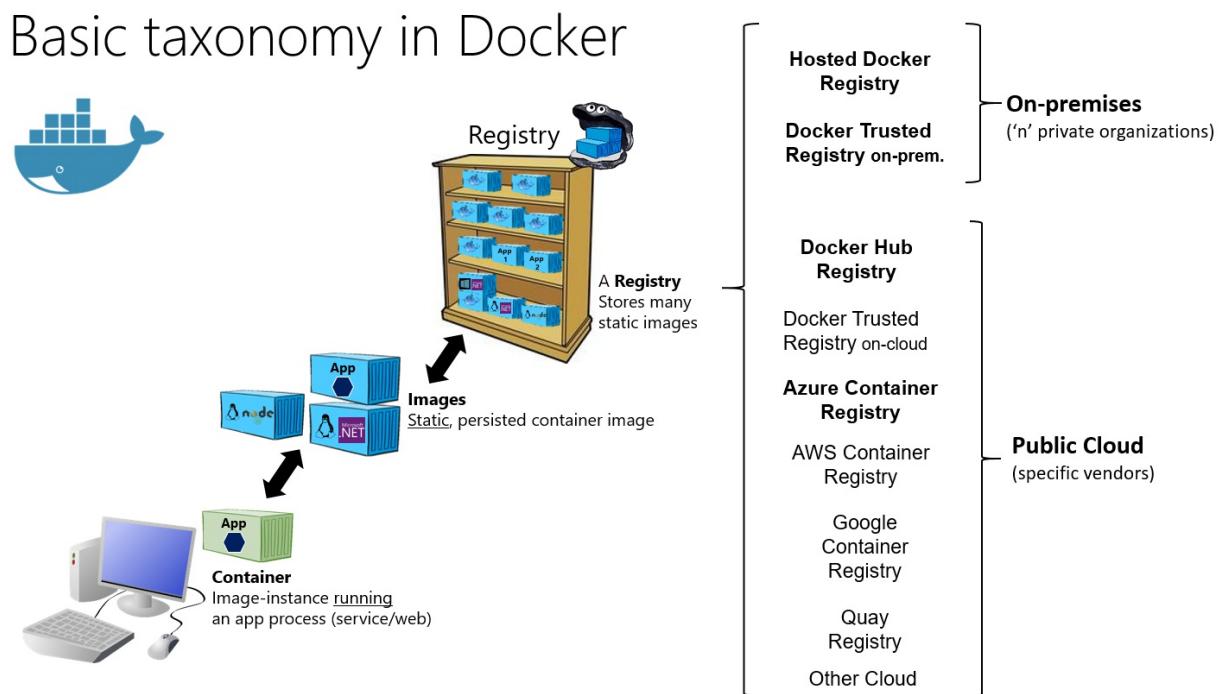


Figura 2-4. Taxonomía de términos de Docker y conceptos

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 de Docker privados a nivel local y en la nube pública. Docker Hub es que un registro público mantenido por Docker; junto con Docker Trusted Registry, una solución a nivel empresarial, Azure ofrece Azure Container Registry. AWS, Google y otros también tienen registros de contenedor.

Colocar imágenes en un registro le permite almacenar fragmentos de la aplicación que son estáticos e inmutables, incluidas todas sus dependencias a nivel de marco. Después, esas imágenes se pueden versionear e implementar en varios entornos y, por tanto, proporcionar una unidad de implementación coherente.

Los registros de imágenes privados, ya sean hospedados localmente o en la nube, se recomiendan cuando:

- Las imágenes no deben compartirse públicamente por motivos de 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, puede tener un registro de confianza de Docker local disponible dentro de la misma red local.

[ANTERIOR](#)

[SIGUIENTE](#)

Selección entre .NET Core y .NET Framework para contenedores de Docker

23/10/2019 • 2 minutes to read • [Edit Online](#)

Se admiten dos marcos para compilar aplicaciones de Docker en contenedor del lado servidor con .NET: [.NET Framework y .NET Core](#). Ambos comparten muchos de los componentes de la plataforma .NET y es posible compartir código entre ellos. Aun así, presentan diferencias fundamentales, y la elección del marco dependerá de lo que quiera realizar. En esta sección se proporciona orientación sobre cuándo se debe elegir cada marco.

[ANTERIOR](#)

[SIGUIENTE](#)

Orientación general

25/11/2019 • 3 minutes to read • [Edit Online](#)

En esta sección se proporciona un resumen de cuándo es mejor elegir .NET Core y cuándo es preferible .NET Framework. Se proporcionan más detalles acerca de estas opciones en las secciones siguientes.

Se recomienda utilizar .NET Core con contenedores de Linux o Windows en la aplicación de servidor Docker en contenedor cuando:

- Tenga necesidades multiplataforma. Por ejemplo, si quiere utilizar contenedores de Linux y Windows.
- La arquitectura de la aplicación esté basada 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 contenedores, debe optar por .NET Core como opción predeterminada, ya que esta opción presenta muchas ventajas y es la que mejor se adapta a la filosofía y al estilo de trabajo de los contenedores.

Otra ventaja adicional de usar .NET Core es que puede ejecutar versiones paralelas de .NET para aplicaciones en el mismo equipo. Esta ventaja es más importante para servidores o máquinas virtuales que no utilizan contenedores, porque los contenedores aíslan las versiones de .NET que necesita la aplicación. (Siempre que sean compatibles con el sistema operativo subyacente).

Se recomienda utilizar .NET Framework en la aplicación de servidor Docker en contenedor cuando:

- La aplicación ya utilice .NET Framework y dependa fuertemente de Windows.
- Tenga que usar API de Windows que no sean compatibles con .NET Core.
- Necesite usar bibliotecas .NET de terceros o paquetes NuGet que no estén disponibles para .NET Core.

Utilizar .NET Framework en Docker puede mejorar sus experiencias de implementación minimizando los problemas de implementación. Este [escenario de migración mediante lift-and-shift](#) es importante para aplicaciones en contenedor heredadas que se desarrollaron originalmente con .NET Framework, como formularios web de ASP.NET, aplicaciones web MVC o servicios de WCF (Windows Communication Foundation).

Recursos adicionales

- **Libro electrónico: Modernize existing .NET Framework applications with Azure and Windows Containers** (Libro electrónico: Modernización de las aplicaciones .NET Framework existentes con contenedores de Azure y de Windows)
<https://aka.ms/liftandshiftwithcontainersebook>
- **Sample apps: Modernization of legacy ASP.NET web apps by using Windows Containers** (Aplicaciones de ejemplo: Modernización de aplicaciones web de ASP.NET heredadas mediante contenedores de Windows)
<https://aka.ms/eshopmodernizing>

Cuándo elegir .NET Core para contenedores de Docker

23/10/2019 • 8 minutes to read • [Edit Online](#)

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

Además, .NET Core es multiplataforma, por lo que puede 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, le ofrecemos una explicación más detallada sobre por qué elegir .NET Core.

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 Core, puesto que .NET Framework solo es compatible con Windows.

.NET Core 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.

[Visual Studio para Mac](#) es un IDE, una evolución de Xamarin Studio, que se ejecuta en macOS y admite el desarrollo de aplicaciones basadas en Docker. Debe ser la opción preferida para los desarrolladores que trabajan en equipos Mac y que también deseen usar un IDE eficaz.

También puede usar [Visual Studio Code](#) (VS Code) en macOS, Linux y Windows. VS Code es compatible con .NET Core, incluidos IntelliSense y depuración. Como VS Code es un editor ligero, puede usarlo para desarrollar aplicaciones en contenedor en Mac, junto con la interfaz de la línea de comandos de Docker y la [interfaz de la línea de comandos \(CLI\) de .NET Core](#). También puede utilizar .NET Core 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 las herramientas de la [CLI de .NET Core](#) 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 puede usar .NET Framework en contenedores de Windows, la modularidad y ligereza de .NET Core 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 Core 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.

En cambio, .NET Core es la mejor opción si ha adoptado un sistema orientado a microservicios que se basa en contenedores, puesto que .NET Core es ligero. Además, sus imágenes de contenedor relacionadas, ya sea la imagen de Linux o la imagen de Windows Nano, 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 fáciles de exemplificar, como la imagen de contenedor de .NET Core.

Una arquitectura de microservicios también le permite mezclar tecnologías en un límite de servicio. Esto permite migrar gradualmente a .NET Core microservicios nuevos que funcionan 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 Core y ASP.NET Core son las mejores opciones. ASP.NET Core es hasta diez veces más rápido que ASP.NET en el tradicional .NET Framework y lidera otras tecnologías para microservicios populares del sector, como servlets de Java, Go y Node.js.

Esto 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 tiempo de ejecución de .NET Core) 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

25/11/2019 • 8 minutes to read • [Edit Online](#)

Mientras que .NET Core ofrece ventajas significativas para las aplicaciones nuevas y los patrones de aplicación, .NET Framework continuará siendo una buena elección para muchos escenarios existentes.

Migrar aplicaciones existentes directamente a un contenedor de Windows Server

Puede usar los contenedores de Docker simplemente para simplificar la implementación, incluso si no va a crear microservicios. Por ejemplo, es posible que quiera mejorar el flujo de trabajo de DevOps con Docker; los contenedores pueden proporcionarle entornos de prueba mejor aislados y también pueden eliminar los problemas de implementación causados por las dependencias que faltan al moverse a un entorno de producción. En estos casos, incluso si está implementando una aplicación monolítica, tiene sentido usar contenedores de Windows y Docker para las aplicaciones de .NET Framework actuales.

En la mayoría de los casos para este escenario, no deberá migrar las aplicaciones existentes a .NET Core; puede usar los contenedores de Docker que incluyen el .NET Framework tradicional. Sin embargo, un enfoque recomendado es usar .NET Core al extender una aplicación existente, como escribir un servicio nuevo en ASP.NET Core.

Uso de bibliotecas .NET de terceros o paquetes de NuGet que no están disponibles para .NET Core

Las bibliotecas de terceros están adoptando rápidamente [.NET Standard](#), que permite el uso compartido de código entre todos los tipos .NET, entre ellos .NET Core. Con la biblioteca .NET Standard 2.0 y versiones posteriores, la compatibilidad de la superficie de la API a través de diferentes marcos ha aumentado significativamente y, en aplicaciones .NET Core 2.x, también puede hacer referencia directamente a las bibliotecas de .NET Framework existentes (vea [.NET Standard 2.0](#)).

Además, el [paquete de compatibilidad de Windows](#) se publicó en noviembre de 2017 para ampliar la superficie de API disponible para .NET Standard 2.0 en Windows. Este paquete permite volver a compilar la mayoría del código existente para .NET Standard 2.x con poca o ninguna modificación para ejecutarse en Windows.

Sin embargo, incluso con ese avance excepcional desde .NET Standard 2.0 y .NET Core 2.1, puede haber casos en que ciertos paquetes de NuGet necesiten Windows para ejecutar y puede que no admitan .NET Core. Si los paquetes son críticos para la aplicación, entonces debe usar .NET Framework en los contenedores de Windows.

Uso de tecnologías de .NET que no están disponibles para .NET Core

Algunas tecnologías de .NET Framework no están disponibles en la versión actual de .NET Core (versión 2.2 cuando se redactó este documento). Algunas estarán disponibles en versiones posteriores de .NET Core (.NET Core 2.x), pero otras no se aplican a los nuevos patrones de aplicaciones a los que se dirige .NET Core y puede que nunca estén disponibles.

En la lista siguiente se muestra la mayoría de las tecnologías que no están disponibles en .NET Core 2.x:

- ASP.NET Web Forms. Esta tecnología solo está disponible en .NET Framework. Actualmente no está previsto migrar ASP.NET Web Forms a .NET Core.

- Servicios WCF. Aunque hay una [biblioteca cliente de WCF](#) disponible para consumir servicios WCF desde .NET Core, a partir de mediados de 2017, la implementación del servidor WCF solo está disponible en .NET Framework. Este escenario podría considerarse para futuras versiones de .NET Core; incluso se contempla la inclusión de algunas API en el [paquete de compatibilidad de Windows](#).
- Servicios relacionados con el flujo de trabajo. Windows Workflow Foundation (WF), Workflow Services (WCF + WF en un único servicio) y WCF Data Services (antes conocido como ADO.NET Data Services) solo están disponibles en .NET Framework. Actualmente no existen planes de ponerlos en .NET Core.

Además de las tecnologías indicadas en la [guía básica de .NET Core](#) oficial, se pueden pasar otras características a .NET Core. Para obtener una lista completa, examine los elementos etiquetados como [port-to-core](#) en el sitio de CoreFX GitHub. Tenga en cuenta que esta lista no representa ningún compromiso de Microsoft para llevar dichos componentes a .NET Core; los elementos simplemente capturan las solicitudes de la comunidad. Si le interesa cualquiera de los componentes mencionados anteriormente, puede participar en las discusiones en GitHub para que su voz pueda ser escuchada. Y si piensa que falta algo, [registre un nuevo problema en el repositorio de CoreFX](#).

Aunque .NET Core 3 (en proceso en el momento en que se redacta este artículo) incluirá compatibilidad para una gran cantidad de API existentes de .NET Framework, también se orientan a entornos de escritorio, aunque actualmente no se usan en el contexto de los contenedores.

Uso de una plataforma o API no compatible con .NET Core

Algunas plataformas de Microsoft o de terceros no son compatibles con .NET Core. Por ejemplo, algunos servicios de Azure proporcionan un SDK que aún no está disponible para su consumo en .NET Core. Esto es temporal, porque todos los servicios acabarán usando .NET Core. Por ejemplo, el [Azure DocumentDB SDK para .NET Core](#) se publicó como una versión preliminar el 16 de noviembre de 2016, pero ahora está disponible con carácter general (GA) como una versión estable.

Mientras tanto, si cualquier plataforma o servicio en Azure todavía no es compatible con .NET Core con la API de cliente, puede usar la API de REST equivalente del servicio de Azure o el SDK de cliente en .NET Framework.

Recursos adicionales

- **Guía de .NET Core**
<https://docs.microsoft.com/dotnet/core/index>
- **Portabilidad a .NET Core desde .NET Framework**
<https://docs.microsoft.com/dotnet/core/porting/index>
- **.NET Core en la guía de Docker**
<https://docs.microsoft.com/dotnet/core/docker/introduction>
- **Introducción a los componentes de .NET**
<https://docs.microsoft.com/dotnet/standard/components>

[ANTERIOR](#)

[SIGUIENTE](#)

Tabla de decisiones: versiones de .NET Framework para su uso con Docker

23/10/2019 • 2 minutes to read • [Edit Online](#)

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

IMPORTANT

Los equipos de desarrollo ejecutarán un host de Docker, ya sea Linux o Windows. Todos los microservicios relacionados que quiera ejecutar y probar juntos en una solución deberán ejecutarse en la misma plataforma de contenedor.

ARQUITECTURA/TIPO DE APLICACIÓN	CONTENEDORES DE LINUX	CONTENEDORES DE WINDOWS
Microservicios en contenedores	Núcleo de .NET	Núcleo de .NET
Aplicación monolítica	Núcleo de .NET	.NET Framework Núcleo de .NET
Rendimiento y escalabilidad líderes	Núcleo de .NET	Núcleo de .NET
Migración de aplicación heredada de Windows Server ("brown-field") a contenedores	--	.NET Framework
Nuevo desarrollo basado en contenedor ("green-field")	Núcleo de .NET	Núcleo de .NET
ASP.NET Core	Núcleo de .NET	.NET Core (recomendado) .NET Framework
ASP.NET 4 (MVC 5, API web 2 y formularios Web Forms)	--	.NET Framework
Servicios SignalR	.NET Core 2.1 o versiones posteriores	.NET Framework .NET Core 2.1 o versiones posteriores
WCF, WF y otros marcos heredados	WCF en .NET Core (solo la biblioteca cliente)	.NET Framework WCF en .NET Core (solo la biblioteca cliente)
Consumo de servicios de Azure	Núcleo de .NET (finalmente todos los servicios de Azure proporcionarán el SDK de cliente para .NET Core)	.NET Framework Núcleo de .NET (finalmente todos los servicios de Azure proporcionarán el SDK de cliente para .NET Core)

[ANTERIOR](#)

[SIGUIENTE](#)

Selección del sistema operativo de destino con contenedores de .NET

25/11/2019 • 4 minutes to read • [Edit Online](#)

Teniendo en cuenta la diversidad de sistemas operativos compatibles con Docker y las diferencias entre .NET Framework y .NET Core, debe escoger un sistema operativo de destino específico y versiones específicas según el marco que utilice.

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

Para Linux, hay varias distribuciones disponibles y compatibles en imágenes oficiales del Docker de .NET (por ejemplo, Debian).

En la Figura 3-1 se puede ver la posible versión de sistema operativo en función de la versión de .NET Framework que se utilice.

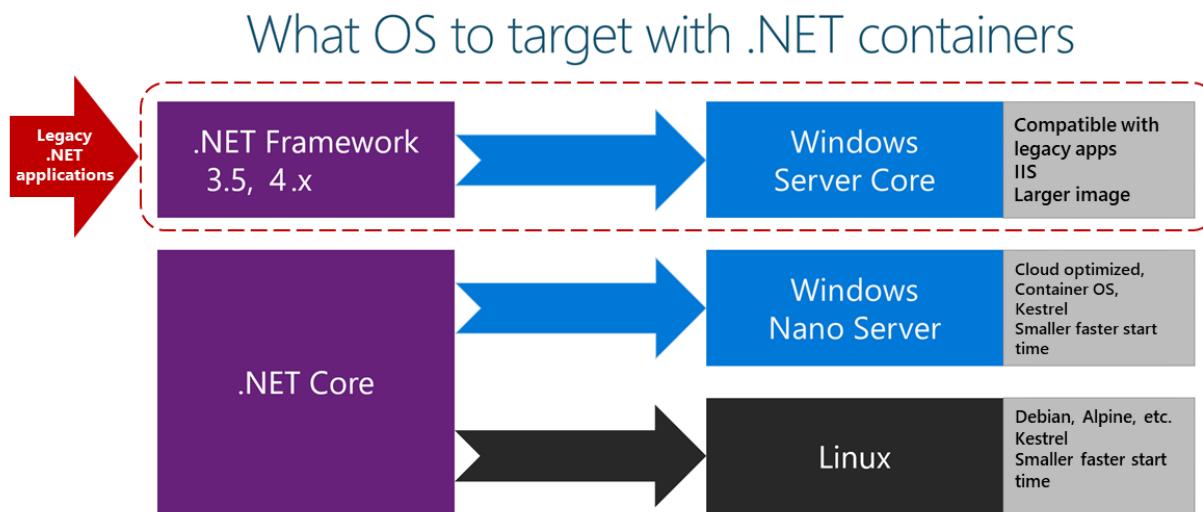


Figura 3-1. Sistemas operativos de destino en función de las versiones de .NET Framework

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 Core, puede tener como destino Windows Nano Server, que está optimizado para la nube, usa Kestrel y es más pequeño y se inicia más rápido. También puede tener como destino Linux, con compatibilidad con Debian, Alpine y otros. También usa Kestrel, que es más pequeño y se inicia más rápido.

También puede crear su propia imagen de Docker en los casos en que quiera utilizar una distribución de Linux diferente o que quiera una imagen con las versiones no proporcionadas por Microsoft. Por ejemplo, puede crear una imagen con ASP.NET Core ejecutándose en los tradicionales .NET Framework y Windows Server Core, que no sería un escenario tan habitual para Docker.

IMPORTANT

Al usar imágenes de Windows Server Core, es posible que falten algunos archivos DLL cuando se comparan con Imágenes de Windows completas. Es posible que pueda resolver este problema mediante la creación de una imagen de Server Core personalizada, agregando los archivos que faltan en el momento de la compilación de la imagen, tal como se ja mencionado en este [comentario de GitHub](#).

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

IMAGEN	COMENTARIOS
mcr.microsoft.com/dotnet/core/runtime:2.2	Arquitectura múltiple de .NET Core 2.2: es compatible con Linux y Windows Nano Server en función del host de Docker.
mcr.microsoft.com/dotnet/core/aspnet:2.2	Arquitectura múltiple de .NET Core 2.2: es compatible con Linux y Windows Nano Server en función del host de Docker. La imagen de aspnetcore tiene algunas optimizaciones para ASP.NET Core.
mcr.microsoft.com/dotnet/core/aspnet:2.2-alpine	.NET Core 2.2 solo en tiempo de ejecución en una distribución de Alpine Linux
mcr.microsoft.com/dotnet/core/aspnet:2.2-nanoserver-1803	.NET Core 2.2 solo en tiempo de ejecución en Windows Nano Server (Windows Server 1803)

Recursos adicionales

- **Se produce un error en BitmapDecoder debido a que falta WindowsCodecsExt.dll (incidencia de GitHub) .**

<https://github.com/microsoft/dotnet-framework-docker/issues/299>

[ANTERIOR](#)

[SIGUIENTE](#)

Imágenes oficiales de Docker de .NET

23/10/2019 • 6 minutes to read • [Edit Online](#)

Las imágenes oficiales de Docker de .NET son imágenes de Docker que Microsoft ha creado y optimizado. Están disponibles públicamente en los repositorios de Microsoft en [Docker Hub](#). 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, etcétera).

Desde .NET Core 2.1, todas las imágenes de .NET Core, incluidas las de ASP.NET Core, están disponibles en Docker Hub, en el repositorio de imágenes de .NET Core: https://hub.docker.com/_/microsoft-dotnet-core/.

La mayoría de los repositorios de imágenes ofrece 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 Core y Docker para desarrollo y 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 Core.
- Imágenes que se usan para *ejecutar* aplicaciones de .NET Core.

¿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 Core de desarrollo (mcr.microsoft.com/dotnet/core/sdk:2.2) 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.

¿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 que se pueden implementar e iniciar los contenedores según una imagen de .NET Core de producción. Por tanto, la imagen solo en tiempo de ejecución basada en mcr.microsoft.com/dotnet/core/aspnet:2.2 es pequeña, por lo que puede desplazarse 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 Core 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 Docker Hub, 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:

IMAGEN	COMENTARIOS
<code>mcr.microsoft.com/dotnet/core/aspnet:2.2</code>	ASP.NET Core, solo con tiempo de ejecución y las optimizaciones de ASP.NET Core, en Linux y Windows (multiarquitectura)
<code>mcr.microsoft.com/dotnet/core/sdk:2.2</code>	.NET Core, con los SDK incluidos, en Linux y Windows (multiarquitectura)

[ANTERIOR](#)

[SIGUIENTE](#)

Diseño de la arquitectura de aplicaciones basadas en contenedores y microservicios

23/10/2019 • 4 minutes to read • [Edit Online](#)

Los microservicios ofrecen grandes ventajas, pero también generan nuevos desafíos enormes. Los patrones de arquitectura de microservicios son los pilares fundamentales a la hora de crear una aplicación basada en microservicios.

Previamente en esta guía, ha aprendido los conceptos básicos sobre los contenedores y Docker. Esta era la información mínima necesaria para empezar a trabajar con contenedores. Aunque los contenedores son habilitadores y se consideran una gran elección para microservicios, no son obligatorios para una arquitectura de microservicios, y muchos conceptos arquitectónicos de esta sección sobre la arquitectura también se podrían aplicar sin contenedores. A pesar de ello, esta guía se centra en la intersección de ambos debido a la importancia actual 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. En esos casos, debe comprender otros enfoques de arquitectura, como son los microservicios y determinados patrones de diseño guiado por el dominio (DDD), además de conceptos de orquestación de contenedores. Tenga en cuenta que en este capítulo no solo se describen los microservicios en contenedor, sino 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 un límite de proceso, puede crear primitivas que se usen para escalar el proceso o para procesarlo por lotes.

Al diseñar una imagen de contenedor, verá una definición **ENTRYPOINT** en el archivo Dockerfile. Esto 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 se implementarían como [WebJobs](#) de Azure.

Si se produce un error en el proceso, el contenedor finaliza y lo sustituye el orquestador. Si el orquestador está configurado para mantener cinco instancias en ejecución y se produce un error en una de ellas, el orquestador creará otra instancia del contenedor para reemplazar al proceso con error. En un trabajo por lotes, el proceso se inicia con parámetros. Cuando el proceso finalice, el trabajo se habrá completado. Más adelante en esta guía se exploran en profundidad los orquestadores.

Es posible que en algún momento le interese que varios procesos se ejecuten en un solo contenedor. En ese caso, dado que solo puede haber un punto de entrada por contenedor, puede ejecutar dentro del contenedor un script que inicie todos los programas que sean necesarios. Por ejemplo, puede usar [Supervisor](#) o una herramienta similar para que se encargue de iniciar varios procesos dentro de un único contenedor. Este enfoque no es muy habitual, aunque existan arquitecturas que contengan varios procesos por contenedor.

[ANTERIOR](#)

[SIGUIENTE](#)

Incluir en un contenedor aplicaciones monolíticas

25/11/2019 • 11 minutes to read • [Edit Online](#)

Puede compilar una aplicación o un servicio web único, implementado de forma monolítica, e implementarlo como un contenedor. La propia aplicación podría no ser internamente monolítica, sino estructurada en forma de varias bibliotecas, componentes o incluso capas (nivel de aplicación, capa de dominio, capa de acceso a datos, etc.). Pero, externamente, es un contenedor único, un proceso único, una aplicación web única o un servicio único.

Para administrar este modelo, debe implementar un único contenedor para representar la aplicación. Para aumentar la capacidad, deberá escalar horizontalmente, es decir, solo tiene que agregar más copias con un equilibrador de carga delante. La simplicidad proviene de administrar una única implementación en un solo contenedor o máquina virtual.

Monolithic Containerized application

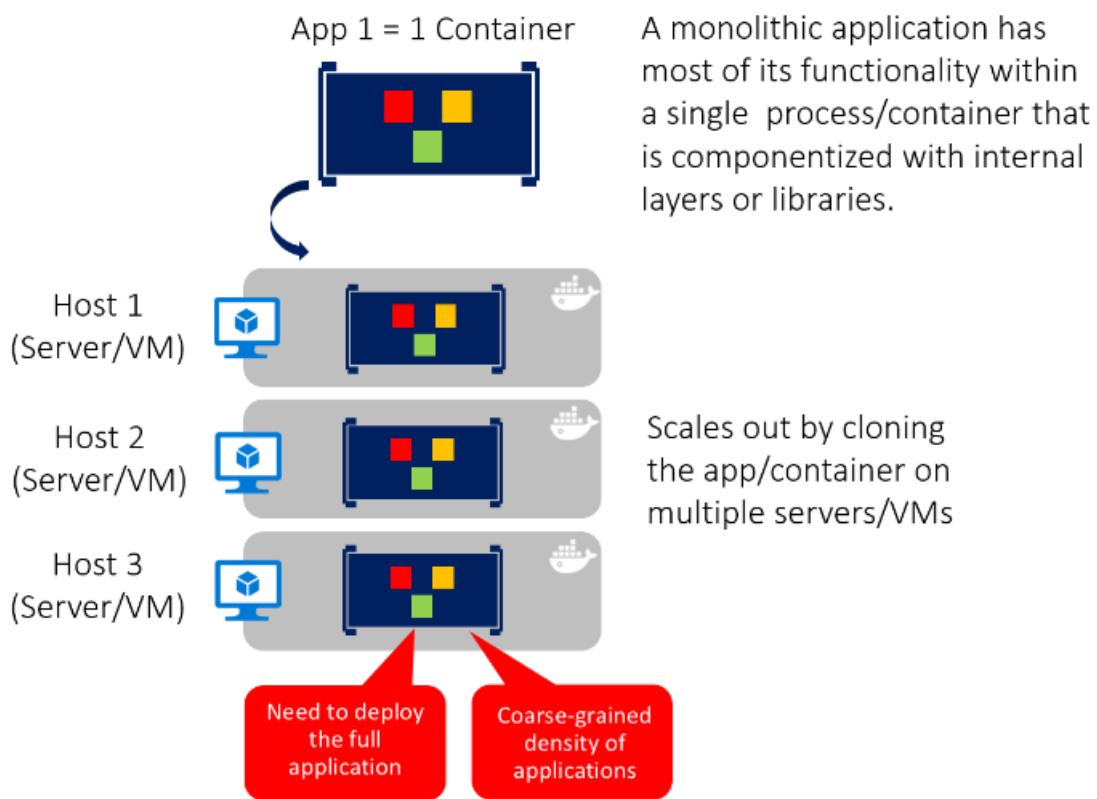


Figura 4-1. Ejemplo de la arquitectura de una aplicación monolítica en contenedores

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

El inconveniente de este enfoque se vuelve evidente si la aplicación aumenta y debe escalarse. Si puede escalar toda la aplicación, no es realmente un problema. Sin embargo, en la mayoría de los casos, solo unos pocos elementos de la aplicación son los puntos de obstrucción que deben escalarse, mientras que otros componentes se usan menos.

Por ejemplo, en una aplicación típica de comercio electrónico, es probable que deba escalar el subsistema de información del producto, dado que muchos más clientes examinan los productos en lugar de comprarlos. Más clientes usan la cesta en lugar de usar la canalización de pago. Menos clientes publican comentarios o consultan su historial de compras. Y es posible que solo un grupo reducido de empleados deba administrar el contenido y las campañas de marketing. Si escala el diseño monolítico, todo el código para estas distintas tareas se implementa varias veces y se escala al mismo nivel.

Hay varias formas de escalar una aplicación: duplicación horizontal, división de diferentes áreas de la aplicación y partición de conceptos o datos empresariales similares. Pero, además del problema de escalar todos los componentes, para introducir cambios en un único componente se debe volver a probar por completo toda la aplicación e implementar por completo todas las instancias.

Sin embargo, el enfoque monolítico es común, porque el desarrollo de la aplicación es inicialmente más fácil que en el caso de los enfoques de microservicios. Por tanto, muchas organizaciones desarrollan con este enfoque arquitectónico. Mientras que algunas organizaciones han tenido resultados suficientemente buenos, otras solo llegan a los límites. Muchas organizaciones diseñaron sus aplicaciones mediante este modelo ya que, años atrás, crear arquitecturas orientadas a servicios (SOA) con infraestructura y herramientas resultaba demasiado difícil y no vieron la necesidad hasta que la aplicación creció.

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

Host running multiple apps/containers

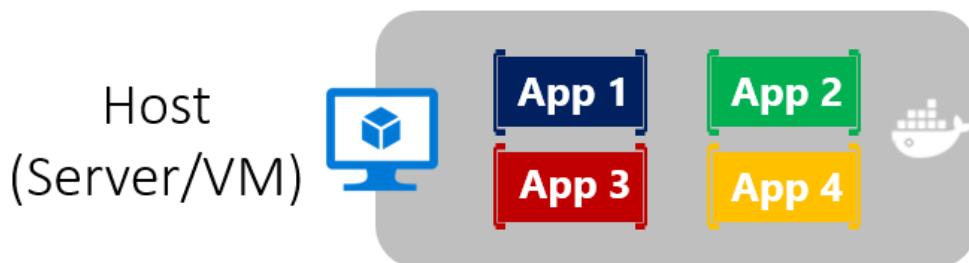


Figura 4-2. Aplicación monolítica: el host ejecuta varias aplicaciones, y cada una se ejecuta como un contenedor.

Las aplicaciones monolíticas en Microsoft Azure se pueden implementar con máquinas virtuales dedicadas para cada instancia. Además, con los [conjuntos de escalado de máquinas virtuales de Azure](#), las máquinas virtuales se pueden escalar fácilmente. [Azure App Service](#) también puede ejecutar aplicaciones monolíticas y escalar fácilmente instancias sin necesidad de administrar las máquinas virtuales. Además, desde 2016, Azure App Services puede ejecutar instancias únicas de contenedores de Docker, lo que simplifica la implementación.

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

Architecture in Docker infrastructure for monolithic applications

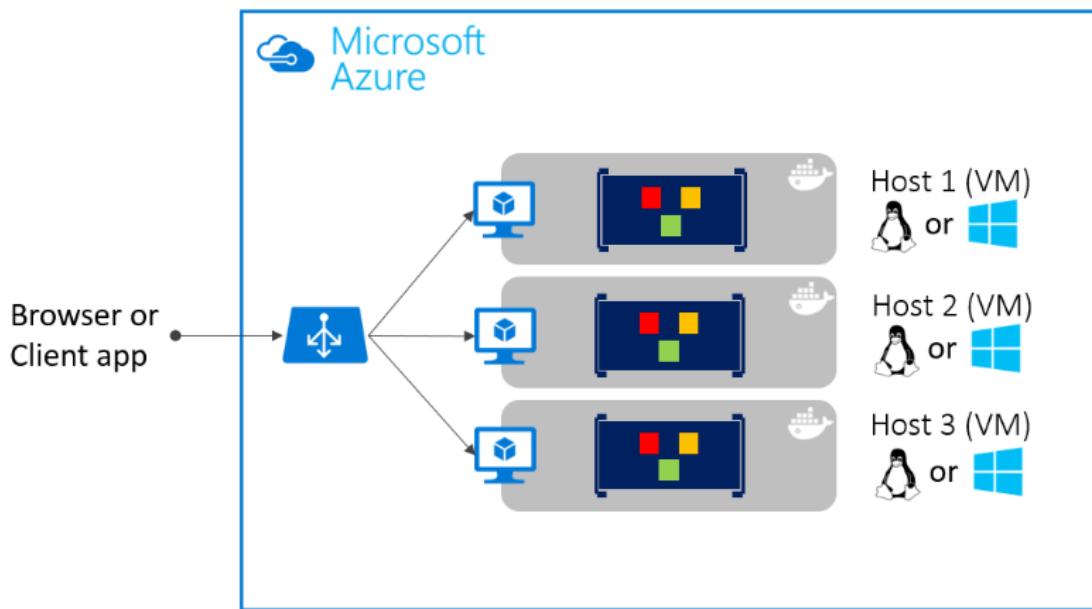


Figura 4-3. Ejemplo de varios hosts que escalan verticalmente una sola aplicación de contenedor

La implementación en los distintos hosts puede administrarse con técnicas de implementación tradicionales. Los hosts de Docker pueden administrarse con comandos como `docker run` o `docker-compose` ejecutados manualmente o a través de automatización como las canalizaciones de entrega continua (CD).

Implementar una aplicación monolítica como un contenedor

Usar contenedores para administrar las implementaciones de aplicaciones monolíticas tiene una serie de ventajas. Escalar las instancias de contenedor es mucho más rápido y fácil que implementar 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 aplicaciones tradicionales en lugar de contenedores, la configuración de la aplicación se administra como parte de la máquina virtual, lo que no es ideal.

Implementar las actualizaciones como imágenes de Docker es mucho más rápido y eficaz en la red. Normalmente, las imágenes de Docker se inician en segundos, lo que acelera los lanzamientos. Anular una instancia de la 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 máquinas virtuales dañadas. En cambio, los scripts de actualización para una máquina virtual podrían olvidar tener en cuenta algún archivo o configuración concreto que se haya quedado en el disco.

Si bien las aplicaciones monolíticas pueden beneficiarse de Docker, estamos examinando estos beneficios muy por encima. Las ventajas adicionales de administrar contenedores proceden de implementar con orquestadores de contenedor, que administran las distintas instancias y el ciclo de vida de cada instancia del contenedor. Separar la aplicación monolítica en subsistemas que se pueden escalar, desarrollar e implementar de forma individual es el punto de entrada al reino de los microservicios.

Publicar una aplicación basada en un solo contenedor en Azure App

Service

Tanto si quiere obtener la validación de un contenedor implementado en Azure o cuando una aplicación es simplemente una aplicación de un solo contenedor, Azure App Service proporciona una excelente manera de proporcionar servicios escalables basados en un solo contenedor. Usar Azure App Service es sencillo. Proporciona una integración excelente con Git para que resulte sencillo tomar el código, crearlo en Visual Studio e implementarlo directamente en Azure.

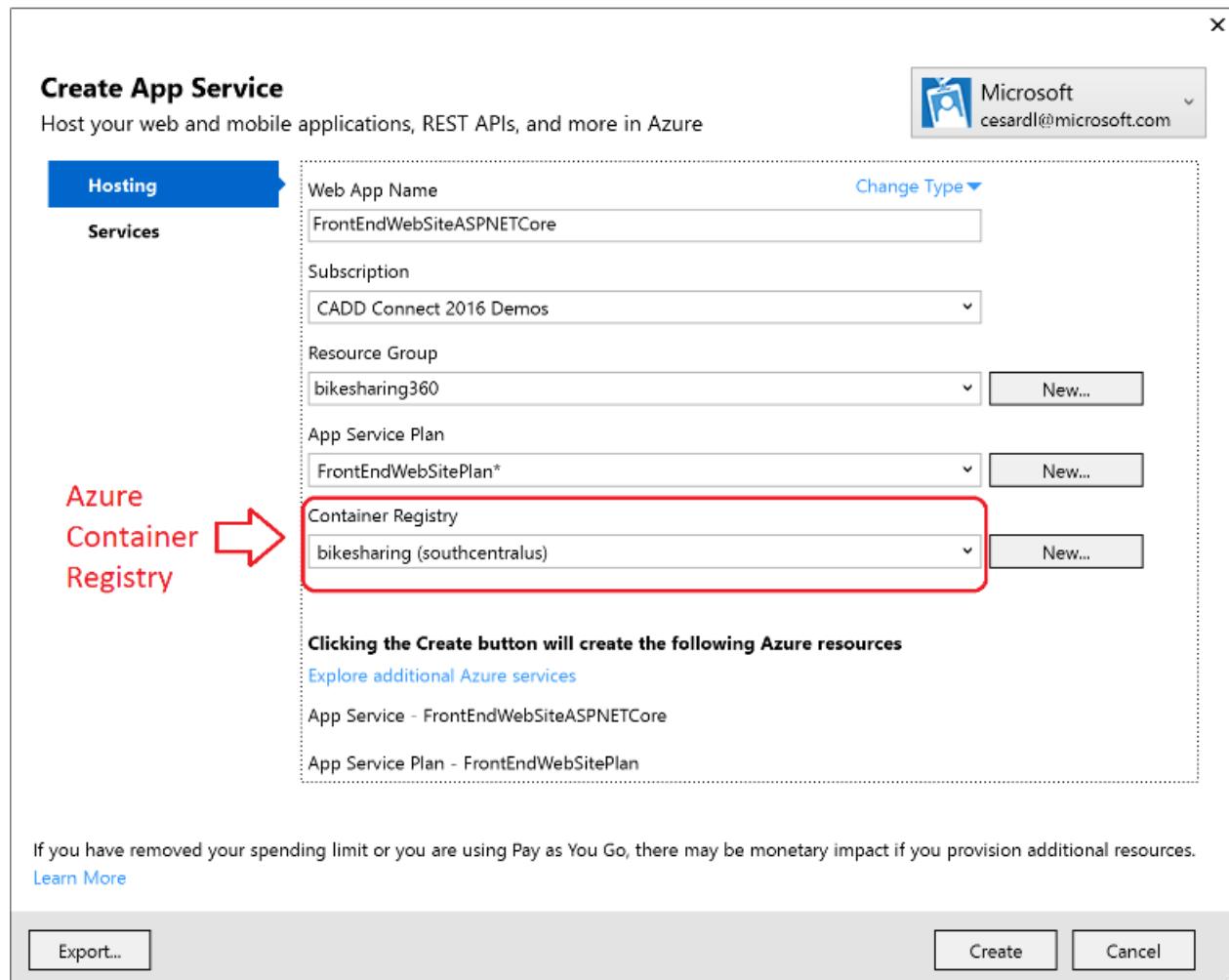


Figura 4-4. Publicar una aplicación de contenedor único en Azure App Service desde Visual Studio

Sin Docker, si necesitaba otras capacidades, marcos o dependencias que no fueran compatibles con Azure App Service, tenía que esperar a que el equipo de Azure actualizara esas dependencias en App Service. O tenía que cambiar a otros servicios como Azure Cloud Services o máquinas virtuales, en que tenía más control y podía instalar un componente o un marco necesario para la aplicación.

La compatibilidad con contenedores de Visual Studio 2017 y posteriores le ofrece la capacidad de incluir todo lo que quiera en el entorno de aplicación, tal como se muestra en la figura 4-4. Puesto que la está ejecutando en un contenedor, si agrega una dependencia a la aplicación, puede incluir la dependencia en Dockerfile o 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 protegido por las cuentas y los grupos de Azure Active Directory) o cualquier otro registro de Docker, como Docker Hub o un registro local.

Estado y datos en aplicaciones de Docker

31/10/2019 • 10 minutes to read • [Edit Online](#)

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. Esta 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.

Data Volume and Data Volume Container

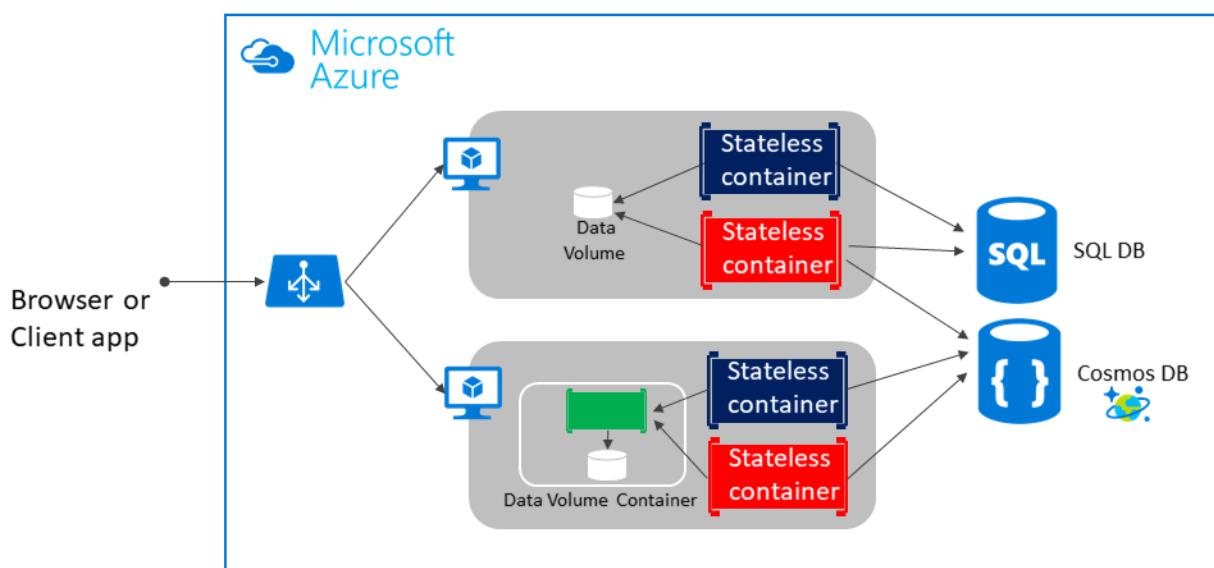


Figura 4-5. Volúmenes y orígenes de datos externos para aplicaciones basadas en contenedor

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:

- El almacenamiento de blobs 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 también se conoce 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 tabla 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. Existen muchas opciones de bases de datos externas, desde las bases de datos relacionales como SQL Server, PostgreSQL u Oracle, o las bases de datos NoSQL como Azure Cosmos DB, MongoDB, etc. En esta guía no se explicarán estas bases de datos puesto que eso se hace en un tema completamente diferente.

[ANTERIOR](#)

[SIGUIENTE](#)

Arquitectura orientada a servicios

23/10/2019 • 2 minutes to read • [Edit Online](#)

La arquitectura orientada a servicios (SOA) era un término sobreutilizado que significaba cosas diferentes para cada persona. Pero, como denominador común, SOA significa que se estructura una aplicación descomponiéndola en varios servicios (normalmente como servicios HTTP) que se pueden clasificar en tipos diferentes, como subsistemas o niveles.

Estos servicios ahora se pueden implementar como contenedores de Docker, con lo que se resuelven los problemas de implementación, puesto que todas las dependencias se incluyen en la imagen de contenedor. Pero cuando se necesita escalar verticalmente aplicaciones SOA, es posible que tenga problemas de escalabilidad y disponibilidad si va a efectuar la implementación en función de hosts de Docker únicos. Aquí es donde puede ayudarle el software de agrupación en clústeres de Docker, o un orquestador, como se explica en secciones posteriores, donde se describen los enfoques de implementación para microservicios.

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

Los microservicios se derivan de SOA, pero SOA no es lo mismo que la arquitectura de microservicios. Características como los grandes agentes centrales, los orquestadores centrales en el nivel de organización y el [Bus de servicio empresarial \(ESB\)](#) son habituales en SOA. Pero en la mayoría de los casos son antipatrones en la comunidad de microservicios. De hecho, hay quien argumenta que "la arquitectura de microservicios es SOA bien hecho".

Esta guía se centra en los microservicios, puesto que los enfoques SOA son menos prescriptivos que los requisitos y técnicas empleados en una arquitectura de microservicios. Si sabe cómo crear una aplicación basada en microservicios, también sabrá cómo crear una aplicación orientada a servicios más sencilla.

[ANTERIOR](#)

[SIGUIENTE](#)

Arquitectura de microservicios

23/10/2019 • 6 minutes to read • [Edit Online](#)

Como su nombre indica, una arquitectura de microservicios es un enfoque para la generación de una aplicación de servidor como un conjunto de servicios pequeños. Esto significa que una arquitectura de microservicios está orientada principalmente hacia el back-end, aunque el enfoque también se utiliza 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 un dominio de un extremo a otro específico o una capacidad empresarial dentro de un determinado límite de contexto, y cada uno se debe desarrollar de forma autónoma e implementar de forma independiente. Por último, cada microservicio debe poseer su modelo de datos de dominio relacionado y su lógica del dominio (soberanía y administración de datos descentralizada), y podría basarse en otras tecnologías de almacenamiento de datos (SQL, NoSQL) y lenguajes de programación.

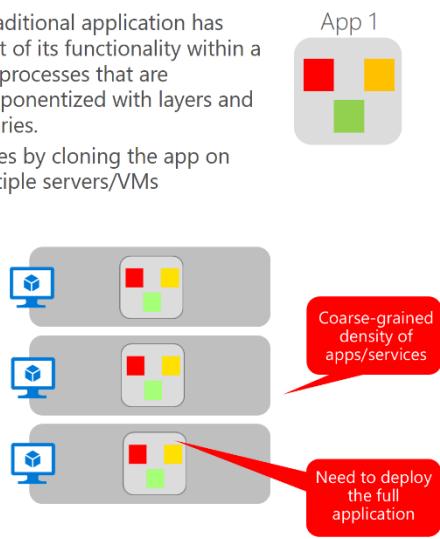
¿Qué tamaño debe tener un microservicio? Al desarrollar un microservicio, el tamaño no debe ser lo más importante. En su lugar, el punto importante debe ser crear libremente servicios acoplados para que tenga autonomía de desarrollo, implementación y escala, para cada servicio. Por supuesto, al identificar y diseñar microservicios, debe intentar que sean lo más pequeños posible, siempre y cuando 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 respecto a otros servicios.

¿Por qué se debe tener una arquitectura de microservicios? En resumen, proporciona agilidad a largo plazo. Con los microservicios puede crear aplicaciones basadas en muchos servicios que se pueden implementar de forma independiente y que tienen ciclos de vida granulares y autónomos, lo que permite un mejor mantenimiento en sistemas complejos, grandes y altamente escalables.

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 esa forma, puede escalar solo el área funcional que necesita más potencia de procesamiento o ancho de banda para admitir la demanda, en lugar de escalar horizontalmente otras partes de la aplicación que no hace falta escalar. Así, puede ahorrar en costes porque necesita menos hardware.

Monolithic deployment approach

- A traditional application has most of its functionality within a few processes that are componentized with layers and libraries.
- Scales by cloning the app on multiple servers/VMs



Microservices application approach

- A microservice application segregates functionality into separate smaller services.
- Scales out by **deploying each service independently** with multiple instances across servers/VMs

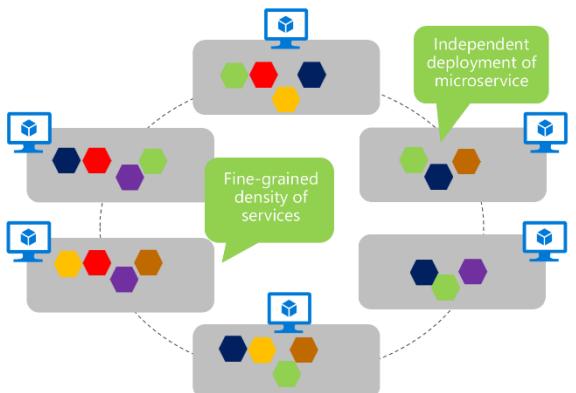


Figura 4-6. Implementación monolítica frente al enfoque de los microservicios

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 aísla en servicios más pequeños, por lo que cada servicio puede escalarse de forma independiente. El enfoque de los microservicios permite modificaciones ágiles e iteraciones rápidas 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 los microservicios de manera aislada y hacerlos evolucionar de forma autónoma a la vez que mantiene contratos claros entre ellos. 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.

Después se indican aspectos importantes para habilitar el éxito de pasar a producción con un sistema basado en microservicios:

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

En esta guía solo se cubren o introducen los tres primeros aspectos. Los dos últimos puntos, que están relacionados con el ciclo de vida de la aplicación, se tratan en el libro electrónico adicional [Ciclo de vida de aplicaciones de Docker en contenedor con la plataforma y las herramientas de Microsoft](#).

Recursos adicionales

- **Mark Russinovich. Microservices: Microservicios: una revolución de las aplicaciones con la 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. Informática en la nube de fragmentos**
<https://www.infoq.com/articles/CCC-Jimmy-Nilsson>
- **Cesar de la Torre. Ciclo de vida de aplicaciones de Docker en contenedor con la plataforma y las herramientas de Microsoft;** (libro electrónico descargable)
<https://aka.ms/dockerlifecyclebook>

Propiedad de los datos por microservicio

25/11/2019 • 10 minutes to read • [Edit Online](#)

Una regla importante de la arquitectura de microservicios es que cada microservicio debe ser propietario de sus datos de dominio y su lógica. Al igual que una aplicación completa posee su lógica y sus datos, cada microservicio debe poseer 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. Piense en las aplicaciones empresariales, donde las aplicaciones de administración de las relaciones con el cliente (CRM), los subsistemas de compras transaccionales y los subsistemas de asistencia al cliente llaman cada uno de ellos a datos y atributos de entidades de cliente únicos y usan un contexto enlazado diferente.

Este principio es similar en el [diseño guiado por el dominio \(DDD\)](#), donde cada [contexto enlazado](#) o subsistema o servicio autónomo debe ser propietario de su modelo de dominio (datos más lógica y comportamiento). Cada contexto enlazado de DDD se correlaciona con un microservicio de negocios (uno o varios servicios). En la sección siguiente se ofrece más información sobre el patrón de contexto enlazado.

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

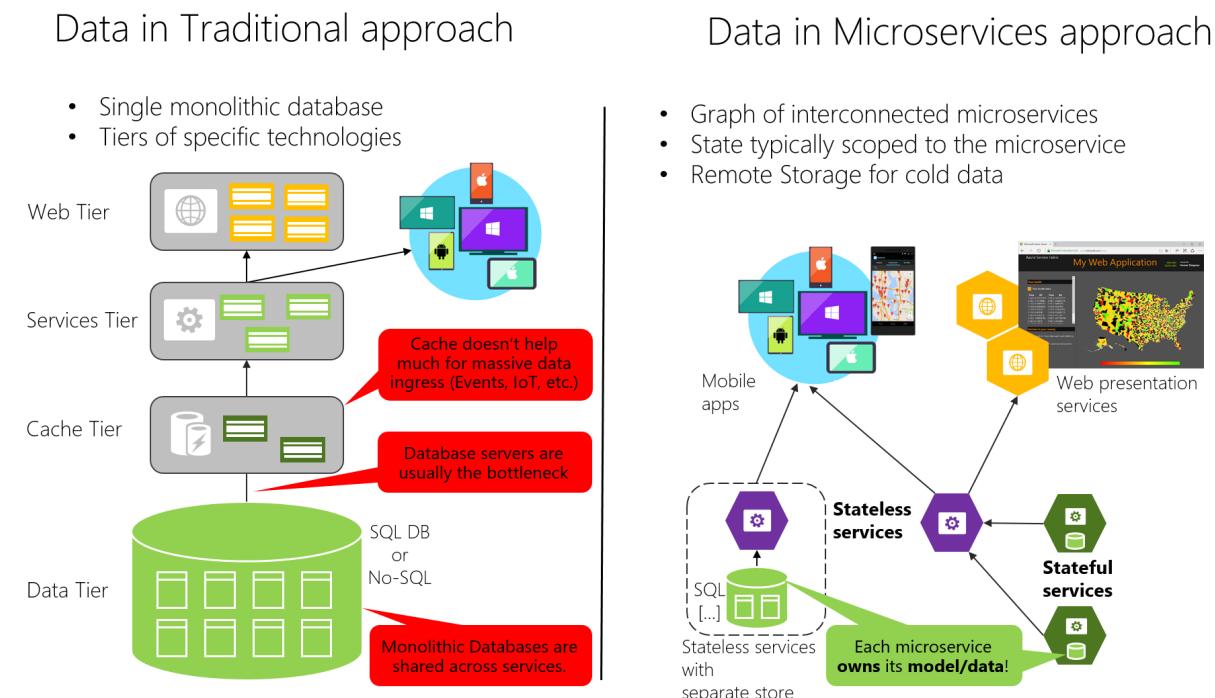


Figura 4-7. Comparación de propiedad de datos: base de datos monolítica frente a microservicios

En el enfoque tradicional, hay una base de datos compartida en todos los servicios, normalmente en una arquitectura en capas. En el enfoque de microservicios, cada microservicio posee sus datos o modelos. El enfoque de la base de datos centralizada en principio parece más sencillo y parece permitir la reutilización de entidades de diferentes subsistemas para que todo sea coherente. Pero la realidad es que se acaban teniendo tablas enormes que sirven a muchos subsistemas distintos e incluyen atributos y columnas que no se necesitan en la mayoría de los casos. Es como intentar usar el mismo mapa físico para ir de excursión un par de horas, para hacer un viaje en coche que dure todo un día y para aprender geografía.

Una aplicación monolítica que normalmente tiene una sola base de datos relacional presenta dos ventajas importantes: [Transacciones ACID](#) y el lenguaje SQL, ambos funcionando en todas las tablas y los datos relacionados con la aplicación. Este enfoque proporciona una manera sencilla de escribir una consulta que combina datos de varias tablas.

Pero el acceso a los datos es mucho más complejo cuando se migra a una arquitectura de microservicios. Incluso cuando se usan transacciones ACID dentro de un microservicio o contexto delimitado, es fundamental tener en cuenta que los datos que pertenecen a cada microservicio son privados para ese microservicio y que solo se debe acceder a ellos de forma sincrónica a través de los puntos de conexión de su API (REST, gRPC, SOAP, etc.), o bien de forma asincrónica a través de mensajería (AMQP o similar).

La encapsulación de los datos garantiza que los microservicios estén acoplados de forma imprecisa y puedan evolucionar independientemente unos de otros. Si varios servicios estuvieran accediendo a los mismos datos, las actualizaciones de esquema exigirían actualizaciones coordinadas de 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 única transacción ACID en microservicios. A su vez, esto significa que debe usar la coherencia final cuando un proceso empresarial abarque varios microservicios. Esto es mucho más difícil de implementar que meras combinaciones SQL, porque no se pueden crear restricciones de integridad o usar las transacciones distribuidas entre bases de datos independientes, como se explicará más adelante. De forma similar, muchas otras características de base de datos relacional no están disponibles en varios microservicios.

Si vamos aún más allá, los distintos microservicios suelen usar *tipos* diferentes de bases de datos. Las aplicaciones modernas almacenan y procesan distintos tipos de datos, así que una base de datos relacional no siempre es la mejor opción. En algunos casos de uso, una base de datos no SQL, como Azure CosmosDB o MongoDB, podría tener un modelo de datos más adecuado 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 usar una combinación de bases de datos SQL y no SQL, lo que a veces se denomina enfoque de [persistencia políglota](#).

Una arquitectura con particiones de persistencia políglota para el almacenamiento de datos tiene muchas ventajas. Estas incluyen servicios acoplados de forma imprecisa y mejor rendimiento, escalabilidad, costos y manejabilidad. Pero puede presentar algunos desafíos de administración de datos distribuida, como se explica en "[Identificar los límites del modelo de dominio para cada microservicio](#)" más adelante en este capítulo.

Relación entre microservicios y el patrón de contexto enlazado

El concepto de microservicio deriva del [patrón de contexto enlazado](#) en el [diseño guiado por el dominio \(DDD\)](#). DDD trabaja con modelos grandes al dividirlos en varios contextos enlazados y ser explícito sobre sus límites. Cada contexto enlazado debe tener su propio modelo y base de datos; del mismo modo, cada microservicio es propietario de sus datos relacionados. Además, cada contexto enlazado normalmente tiene su propio [lenguaje ubicuo](#) para la comunicación entre desarrolladores de software y expertos de dominio.

Esos términos (principalmente entidades de dominio) en el lenguaje ubicuo pueden tener otros nombres en otros contextos enlazados, incluso cuando varias entidades de dominio comparten la misma identidad (es decir, el identificador único que se usa para leer la entidad desde el 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.

Un microservicio es, por tanto, como un contexto enlazado, pero además especifica que es un servicio distribuido. Se compila como un proceso independiente para cada contexto enlazado y debe usar los protocolos distribuidos indicados anteriormente, como HTTP/HTTPS, WebSockets o [AMQP](#). Pero el patrón de contexto enlazado no especifica si el contexto enlazado es un servicio distribuido o si es simplemente un límite lógico (por ejemplo, un subsistema genérico) de una aplicación de implementación monolítica.

Es importante resaltar que la definición de un servicio para cada contexto enlazado es un buen principio. Pero no

es necesario restringir el diseño a esto. A veces debe diseñar un contexto enlazado o microservicio de negocios formado por varios servicios físicos. Pero, en última instancia, ambos patrones, contexto enlazado y microservicio, están estrechamente relacionados.

DDD se beneficia de los microservicios al obtener límites reales en forma de microservicios distribuidos. Pero ideas como no compartir el modelo entre microservicios son las que también se quieren en un contexto enlazado.

Recursos adicionales

- **Chris Richardson. Patrón: Database per service** (Patrón: base de datos por servicio)
<https://microservices.io/patterns/data/database-per-service.html>
- **Martin Fowler. BoundedContext**
<https://martinfowler.com/bliki/BoundedContext.html>
- **Martin Fowler. PolyglotPersistence**
<https://martinfowler.com/bliki/PolyglotPersistence.html>
- **Alberto Brandolini. Diseño orientado a dominios estratégicos con asignación de contexto**
<https://www.infoq.com/articles/ddd-contextmapping>

[ANTERIOR](#)

[SIGUIENTE](#)

Arquitectura lógica frente a arquitectura física

23/10/2019 • 4 minutes to read • [Edit Online](#)

En este momento, es útil detenerse y analizar la diferencia 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 sin formato. Los microservicios son una arquitectura lógica.

Además, incluso cuando un microservicio podría implementarse físicamente como un único servicio, proceso o contenedor (para simplificar, es el enfoque adoptado en la versión inicial de [eShopOnContainers](#)), esta paridad entre microservicio empresarial y servicio o contenedor físico no es necesaria en todos los casos al compilar una aplicación grande y compleja formada por muchas docenas o incluso cientos de servicios.

Aquí es donde hay una diferencia entre la arquitectura lógica y la arquitectura física de una aplicación. La arquitectura lógica y los límites lógicos de un sistema no se asignan necesariamente uno a uno a la arquitectura física o de implementación. Esto puede suceder, pero a menudo no es así.

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

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

Como se muestra en la figura 4-8, el microservicio empresarial de catálogo podría estar compuesto de varios servicios o procesos. Estos podrían ser varios servicios de ASP.NET Web API o cualquier otro tipo de servicio que use HTTP o cualquier otro protocolo. Lo más importante es que los servicios puedan compartir los mismos datos, siempre y cuando estos servicios sean cohesivos con relación al mismo dominio empresarial.

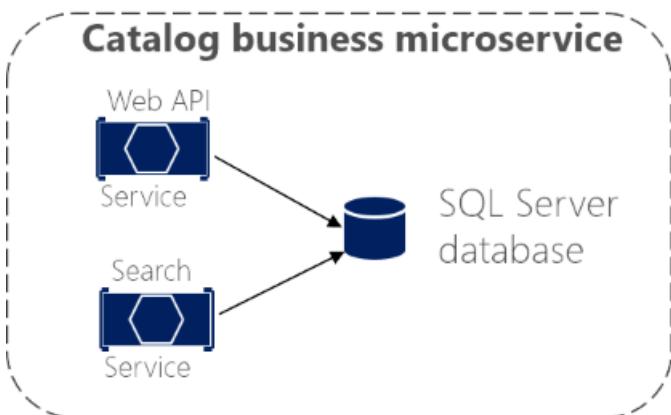


Figura 4-8. Microservicio empresarial con varios servicios físicos

Los servicios del ejemplo comparten el mismo modelo de datos porque el servicio Web API tiene como destino los mismos datos que el servicio Search. Por tanto, en la implementación física del microservicio empresarial, esa función se divide de manera que pueda escalar horizontal o verticalmente cada uno de esos servicios internos según sea necesario. Es posible que el servicio Web API normalmente necesite más instancias que el servicio

Search, 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 se mencione un microservicio, se entiende que es un microservicio empresarial o lógico que se puede asignar a uno o más servicios (físicos). En la mayoría de los casos, se trata de un servicio único, pero puede que sean más.

[ANTERIOR](#)

[SIGUIENTE](#)

Desafíos y soluciones de la administración de datos distribuidos

23/10/2019 • 22 minutes to read • [Edit Online](#)

Desafío n.º 1: Cómo definir los límites de cada microservicio

Definir los límites del microservicio es probablemente el primer desafío con el que nos encontramos. Cada microservicio debe formar parte de la aplicación y a la vez ser autónomo con todas las ventajas y los desafíos que eso conlleva. Pero, ¿cómo se identifican estos límites?

En primer lugar, hay que centrarse en los modelos de dominio de la lógica de la aplicación y en los datos relacionados. Procure identificar islas de datos desacopladas y otros contextos dentro de la misma aplicación. Cada contexto podría tener un lenguaje empresarial diferente (términos empresariales diferentes). Los contextos deben definirse y administrarse de forma independiente. Los términos y las entidades que se usan en esos contextos pueden parecer similares, pero es posible que un concepto empresarial se use para otro propósito según el contexto, e incluso podría tener otro nombre. Por ejemplo, un usuario puede denominarse usuario en el contexto de identidad o pertenencia, cliente en un contexto CRM, comprador en un contexto de pedidos y así sucesivamente.

La manera en que identifica los límites entre varios contextos de aplicación con un dominio diferente para cada contexto es exactamente cómo puede identificar los límites de cada microservicio de negocio con sus respectivos datos y modelo de dominio. Siempre se intenta minimizar el acoplamiento entre esos microservicios. Más adelante en esta guía se explica con más detalle este modelo de diseño de identificación y modelo de dominio en la sección [Identificación de los límites del modelo de dominio para cada microservicio](#).

Desafío n.º 2: Cómo crear consultas que recuperen datos de varios microservicios

Un segundo desafío es implementar consultas que recuperen datos de varios microservicios, evitando al mismo tiempo un exceso de comunicación entre los microservicios y las aplicaciones cliente remotas. Un ejemplo podría ser una pantalla de una aplicación móvil que necesita mostrar información de usuario perteneciente a los microservicios de cesta de la compra, catálogo e identidad de usuario. Otro ejemplo sería un informe complejo que implica muchas tablas ubicadas en varios microservicios. La solución adecuada depende de la complejidad de las consultas. En cualquier caso, se necesita una manera de agregar información para mejorar la eficacia de las comunicaciones del sistema. Las soluciones más comunes son las siguientes:

Puerta de enlace de API. Para una agregación de datos simple de varios microservicios que poseen diferentes bases de datos, el enfoque recomendado es utilizar un microservicio de agregación conocido como puerta de enlace de API. No obstante, se debe tener cuidado con la implementación de este patrón, ya que puede ser un punto de obstrucción en el sistema y puede infringir el principio de autonomía de microservicio. Para mitigar esta posibilidad, puede tener varias puertas de enlace de API y que cada una se centre en un segmento vertical o área de negocio del sistema. El patrón de puerta de enlace de API se detalla más adelante en la [sección Puerta de enlace de API](#).

CQRS con tablas de consulta o lectura. Otra solución para la agregación de datos de varios microservicios es el [patrón de vista materializada](#). En este enfoque, se genera de antemano (se preparan los 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 adaptado a las necesidades de la aplicación cliente.

Piense en algo parecido a la pantalla de una aplicación móvil. Si solo tiene una base de datos, puede reunir los

datos de esa pantalla mediante una consulta SQL que realiza una combinación compleja que implica varias tablas. Pero, si tiene varias bases de datos y cada una pertenece a un microservicio diferente, no se puede consultar las bases de datos y crear una instrucción join (combinación) de SQL. La consulta compleja se convierte en un desafío. Para abordar esta necesidad, se puede usar un enfoque CQRS: crear una tabla desnormalizada en otra base de datos que se use 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 que son necesarios para la pantalla de la aplicación y las columnas de la tabla de consulta. También pueden utilizarse con fines informativos.

Este enfoque no solo resuelve el problema original (cómo realizar consultas y combinaciones en varios microservicios); sino que también mejora el rendimiento considerablemente si se compara con una combinación compleja, puesto que los datos que necesita la aplicación ya están 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. Con todo, los requisitos de rendimiento y alta escalabilidad en [escenarios de colaboración](#) (o escenarios competitivos, según el punto de vista) son donde se debe aplicar CQRS con varias bases de datos.

"Datos fríos" en bases de datos centrales. Para informes complejos y consultas que no necesiten datos en tiempo real, un enfoque común consiste en exportar los "datos dinámicos" (datos transaccionales de los microservicios) como "datos fríos" en grandes bases de datos que se utilizan solo en informes. Dicho sistema de base de datos central puede ser un sistema basado en macrodatos, como Hadoop, un almacén de datos basado por ejemplo en Azure SQL Data Warehouse, o incluso una única base de datos SQL utilizada solamente para informes (si el tamaño no es un problema).

Debe tenerse en cuenta que esta base de datos centralizada tan solo se utilizará para consultas e informes que no requieran datos en tiempo real. Las actualizaciones y las transacciones originales, como origen confiable, deben estar en los datos de microservicios. La forma en que se sincronizarían los datos sería mediante comunicación orientada a eventos (descrita en las secciones siguientes) o mediante otras herramientas de importación y exportación de infraestructura de base de datos. Si se utiliza la comunicación orientada a eventos, el proceso de integración sería similar a la manera en que se propagan los datos como se describió anteriormente para las tablas de consulta CQRS.

Pero si el diseño de la aplicación implica agregar constantemente información procedente de varios microservicios para consultas complejas, podría ser un síntoma de un diseño incorrecto: un microservicio debería estar lo más aislado posible de los demás microservicios. (Esto excluye los informes o análisis que siempre deben usar bases de datos centrales de datos fríos). Si este problema se repite a menudo, podría ser un motivo para combinar los microservicios. Debe equilibrar la autonomía de la evolución y la implementación de cada microservicio con dependencias seguras, cohesión y agregación de datos.

Desafío n.º 3: Cómo lograr que varios microservicios sean coherentes

Como se mencionó anteriormente, los datos que pertenecen a cada microservicio son exclusivos de ese microservicio y solo se puede acceder a ellos mediante la API del microservicio. Por lo tanto, un desafío es cómo implementar procesos empresariales de extremo a extremo manteniendo la coherencia entre varios microservicios.

Para analizar este problema, veamos un ejemplo de la [aplicación de referencia eShopOnContainers](#). El microservicio de catálogo (Catalog) mantiene información sobre todos los productos, incluido el precio del producto. El microservicio de cesta administra datos temporales sobre elementos de producto que los usuarios agregan a su cesta de la compra, lo que incluye el precio de los elementos en el momento en que se han agregado a la cesta. Cuando se actualiza el precio de un producto en el catálogo, ese precio también debe actualizarse en las cestas activas que contienen ese mismo producto, además, el sistema probablemente debería advertir al usuario de que el precio de un elemento determinado ha cambiado desde que lo agregó a su cesta.

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

Cesta.

Pero en una aplicación basada en microservicios, las tablas Productos y Cesta pertenecen a sus respectivos microservicios. Ningún microservicio debería incluir en sus propias transacciones las tablas o el almacenamiento que pertenecen a otro microservicio, ni siquiera en consultas directas, como se muestra en la figura 4-9.

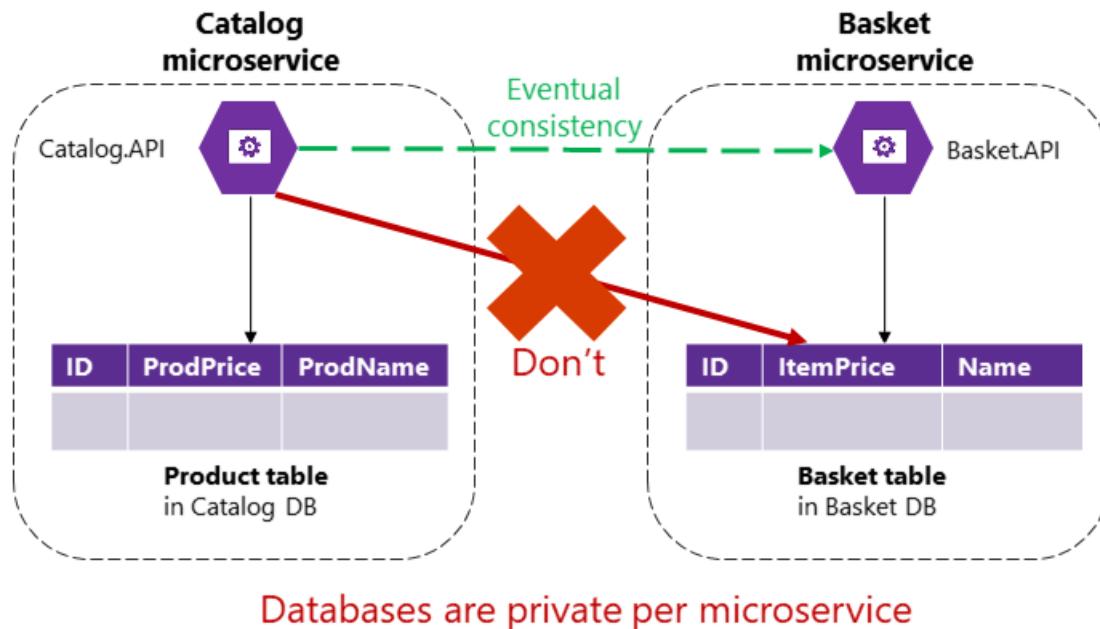


Figura 4-9. Un microservicio no puede acceder directamente a una tabla en otro microservicio

El microservicio de catálogo no debe actualizar directamente la tabla Cesta, dado que esta pertenece al microservicio de cesta. Para realizar una actualización en el microservicio de cesta, el microservicio de catálogo debe usar coherencia final probablemente basada en la comunicación asincrónica como eventos de integración (comunicación basada en mensajes y eventos). Así es como la aplicación de referencia [eShopOnContainers](#) lleva a cabo 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 escalabilidad elevada en lugar de coherencia fuerte. Las aplicaciones críticas deben permanecer activas y en ejecución, y los desarrolladores pueden solucionar el problema de coherencia mediante el uso de técnicas de trabajo con coherencia débil o eventual. Este es el enfoque adoptado por la mayoría de las arquitecturas basadas en microservicios.

Además, las transacciones de confirmación en dos fases de estilo ACID no solo van en contra de los principios de microservicios; la mayoría de las bases de datos NoSQL (Azure Cosmos DB, MongoDB, etc.) no son compatibles con las transacciones de confirmación en dos fases, típicas de los escenarios de bases de datos distribuidas. Pero es esencial mantener la coherencia de los datos entre los servicios y las bases de datos. Este desafío también está relacionado con la cuestión de cómo se propagan los cambios a los distintos microservicios cuando hay datos concretos que deben ser redundantes: por ejemplo, cuando necesite que el nombre o la descripción del producto estén en el microservicio de catálogo y en el microservicio de cesta.

Una buena solución para este problema consiste en usar coherencia eventual entre microservicios articulada mediante comunicación orientada a eventos y un sistema de publicación y suscripción. Estos temas se tratan más adelante en la sección [Comunicación asincrónica orientada a eventos](#) de esta guía.

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

Comunicarse a través de los límites de los microservicios supone un verdadero reto. En este contexto, la comunicación no hace referencia al protocolo que debe usar (HTTP y REST, AMQP, mensajería, etc.). En su lugar,

aborda el estilo de comunicación que se debe utilizar y, en especial, el grado de acoplamiento que deberían tener sus microservicios. Según el nivel de acoplamiento, cuando se produzca un error, el impacto de ese error en el sistema variará considerablemente.

En un sistema distribuido como es una aplicación basada en microservicios, con tantos artefactos desplazándose y con servicios distribuidos en varios servidores o hosts, se acabará produciendo algún error en los componentes. Puesto que se van a producir errores e interrupciones incluso mayores, es necesario diseñar los microservicios y la comunicación entre ellos teniendo en cuenta los riesgos comunes en este tipo de sistemas distribuidos.

Debido a su simplicidad, un enfoque popular consiste en implementar microservicios basados en HTTP (REST). Un enfoque basado en HTTP es absolutamente aceptable; aquí el problema está relacionado con el uso que se hace de él. No hay problema si usa solicitudes y respuestas HTTP para interactuar con sus microservicios desde las aplicaciones cliente o desde las puertas de enlace de API. Pero si crea cadenas largas de llamadas HTTP sincrónicas que afectan a varios microservicios, comunicándose a través de sus límites como si los microservicios fuesen objetos en una aplicación monolítica, la aplicación acabará teniendo problemas.

Por ejemplo, imagine que la aplicación cliente realiza una llamada API HTTP a un microservicio individual como el de pedidos. Si el microservicio de pedidos llama a su vez a otros microservicios mediante HTTP en el mismo ciclo de solicitud/respuesta, estará creando una cadena de llamadas HTTP. Aunque en un principio podría parecer razonable, hay aspectos importantes que se deben tener en cuenta:

- Bloqueo y bajo rendimiento. Debido a la naturaleza sincrónica de HTTP, la solicitud original no obtiene una respuesta hasta que finalicen todas las llamadas HTTP internas. Imagine que el número de estas llamadas aumenta considerablemente y, al mismo tiempo, se bloquea una de las llamadas HTTP intermedias a un microservicio. El resultado es que el rendimiento se verá perjudicado 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. Lo ideal es que "desconozcan" la existencia de otros microservicios. Si la aplicación se basa en el acoplamiento de microservicios como en el ejemplo, será casi imposible lograr la autonomía de cada microservicio.
- Error en un microservicio. Si ha implementado una cadena de microservicios vinculados mediante llamadas HTTP y se produce un error en cualquiera de los microservicios (lo que es seguro que ocurra), se producirá un error en toda la cadena de microservicios. Un sistema basado en microservicios se debe diseñar de modo que siga funcionando lo mejor posible cuando se producen errores parciales. Incluso si decide implementar la lógica de cliente que usa los reintentos con retroceso exponencial o mecanismos de disyuntor, cuanto más complejas sean las cadenas de llamadas HTTP, más difícil será implementar una estrategia contra errores basada en HTTP.

De hecho, si sus microservicios internos se comunican mediante la creación de cadenas de solicitudes HTTP tal como se ha descrito, podría argumentarse que tiene una aplicación monolítica, pero una basada en HTTP entre los procesos en lugar de mecanismos de comunicación intraprocesos.

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

El uso de comunicación asincrónica se explica con más detalle más adelante en esta guía, en las secciones [La integración asincrónica del microservicio obliga a su autonomía](#) y [Comunicación asincrónica basada en mensajes](#).

Recursos adicionales

- [Teorema CAP](#)

https://en.wikipedia.org/wiki/CAP_theorem

- **Coherencia de los eventos**

https://en.wikipedia.org/wiki/Eventual_consistency

- **Manual de coherencia de datos**

[https://docs.microsoft.com/previous-versions/msp-n-p/dn589800\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-n-p/dn589800(v=pandp.10))

- **Martin Fowler. [CQRS (Segregación de responsabilidades de consultas y comandos)]**

<https://martinfowler.com/bliki/CQRS.html>

- **Patrón Materialized View**

<https://docs.microsoft.com/azure/architecture/patterns/materialized-view>

- **Charles Row. ACID vs. BASE: el cambio del pH del procesamiento de transacciones de bases de datos**

<https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/>

- **Transacción de compensación**

<https://docs.microsoft.com/azure/architecture/patterns/compensating-transaction>

- **Udi Dahan. Service Oriented Composition (Composición orientada a servicios)**

<http://udidahan.com/2014/07/30/service-oriented-composition-with-video/>

[ANTERIOR](#)

[SIGUIENTE](#)

Identificar los límites del modelo de dominio para cada microservicio

23/10/2019 • 13 minutes to read • [Edit Online](#)

El objetivo al identificar los límites del modelo y el tamaño de cada microservicio no es llegar a la separación más específica posible (aunque debería intentar usar microservicios pequeños siempre que sea 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 más bien en las capacidades empresariales. Además, si se necesita una clara cohesión para un área concreta de la aplicación sobre la base de un gran número de dependencias, eso también indica la necesidad de un solo microservicio. La cohesión es una manera de identificar cómo separar o agrupar microservicios. En última instancia, al tiempo que conoce mejor el dominio, debe adaptar el tamaño de su microservicio de forma iterativa. Buscar el tamaño adecuado no es un proceso monoestable.

Sam Newman, un reconocido promotor de microservicios y autor del libro [Crear microservicios](#), resalta que los microservicios se deben diseñar de acuerdo con el patrón de contexto limitado (BC) (parte del diseño guiado por el dominio), como se ha mencionado anteriormente. 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 en un BC o un microservicio concreto. Un BC delimita la aplicabilidad de un modelo de dominio y ofrece a los miembros del equipo de desarrolladores una descripción clara y compartida de qué partes deben ser cohesivas y qué partes se pueden desarrollar de manera independiente. Estos son los mismos objetivos para los microservicios.

Otra herramienta que informa sobre su elección de diseño es la [ley de Conway](#), que indica que una aplicación reflejará los límites sociales de la organización que la produjo. Pero a veces sucede lo contrario: el software forma la organización de una empresa. Tal vez deba invertir la ley de Conway y establecer los límites de la forma que quiere que la empresa se organice, decantándose por la consultoría de procesos empresariales.

Para identificar los contextos limitados, puede usar un patrón DDD denominado [patrón de asignación de contexto](#). Con la asignación de contexto, puede identificar los distintos contextos de la aplicación y sus límites. Es habitual tener un contexto y un límite diferentes para cada subsistema pequeño, por ejemplo. La asignación de contexto es una manera de definir y establecer explícitamente esos límites entre dominios. Un BC es autónomo, incluye los detalles de un único dominio, como las entidades de dominio, y define los contratos de integración con otros BC. Esto es similar a la definición de un microservicio: es autónomo, implementa cierta capacidad de dominio y debe proporcionar interfaces. Esta es la razón por la que la asignación de contexto y el patrón de contexto limitado son enfoques excelentes para identificar los límites del modelo de dominio de sus microservicios.

Al diseñar una aplicación grande, verá cómo se puede fragmentar su modelo de dominio; por ejemplo, un experto en dominios del dominio de catálogo denominará las entidades de una manera diferente en los dominios de catálogo e inventario que un experto en dominios de envío. O puede que la entidad de dominio de usuario sea diferente en tamaño y número de atributos cuando se trata de un experto de CRM que quiere almacenar todos los detalles sobre el cliente, en comparación con un experto en dominios de pedido que solo necesita datos parciales sobre el cliente. Es muy difícil eliminar la ambigüedad de todos los términos del 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 que cada dominio proporciona. Si intenta tener una base de datos unificada para toda la aplicación, los intentos de establecer un vocabulario unificado serán difíciles y los resultados no sonarán bien a ninguno de los múltiples expertos en dominios. Por tanto, con los BC (implementados como microservicios) será más fácil aclarar dónde puede usar determinados términos del dominio y dónde debe dividir el sistema y crear BC adicionales con dominios diferentes.

Sabrá que obtuvo los límites y los tamaños correctos de cada BC y modelo de dominio 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 aplicaciones típicas.

Quizá la mejor respuesta a la pregunta de qué tamaño debe tener un modelo de dominio para cada microservicio es la siguiente: debe tener un BC autónomo, tan aislado como sea posible, que le permita trabajar sin tener que cambiar constantemente a otros contextos (otros modelos de microservicio). En la figura 4-10 puede ver cómo varios microservicios (varios BC) tienen su propio modelo y cómo se pueden definir sus entidades, según los requisitos específicos para cada uno de los dominios identificados en la aplicación.

Identifying a Domain Model per Microservice or Bounded Context

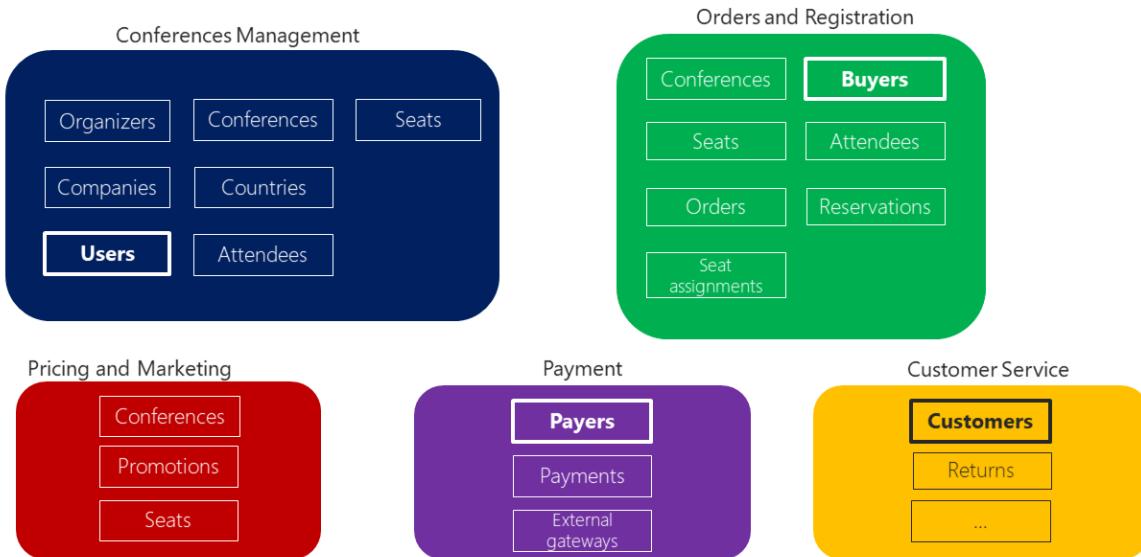


Figura 4-10. Identificación de las entidades y de los límites del modelo de microservicio

En la figura 4-10 se ilustra un escenario de ejemplo relacionado con un sistema de administración de conferencias en línea. La misma entidad aparece como "Users", "Buyers", "Payers" y "Customers" 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 solo en un modelo de microservicio único, como los pagos en el microservicio de pago. Serán fáciles de implementar.

Sin embargo, también puede tener entidades que tienen una forma diferente, pero comparten la misma identidad a través de los múltiples modelos de dominio de los diversos microservicios. Por ejemplo, la entidad User se identifica en el microservicio de administración de conferencias. Ese mismo usuario, con la misma identidad, es el que se llama compradores en el microservicio de pedidos, o el que se llama pagador en el microservicio de pago e incluso el que se llama cliente en el microservicio de servicio al cliente. Esto es porque, según el [lenguaje ubicuo](#) que cada experto en dominios use, un usuario podría tener una perspectiva distinta incluso con atributos diferentes. La entidad de usuario en el modelo de microservicio denominado Administración de conferencias podría tener la mayoría de sus atributos de datos personales. Sin embargo, puede ser que ese mismo usuario en la forma de pagador en el microservicio de pago o en la forma de cliente en el microservicio de servicio al cliente no necesite la misma lista de atributos.

Un enfoque similar se muestra en la figura 4-11.

Decomposing a traditional data model into multiple domain models (One domain model per microservice or Bounded-Context)

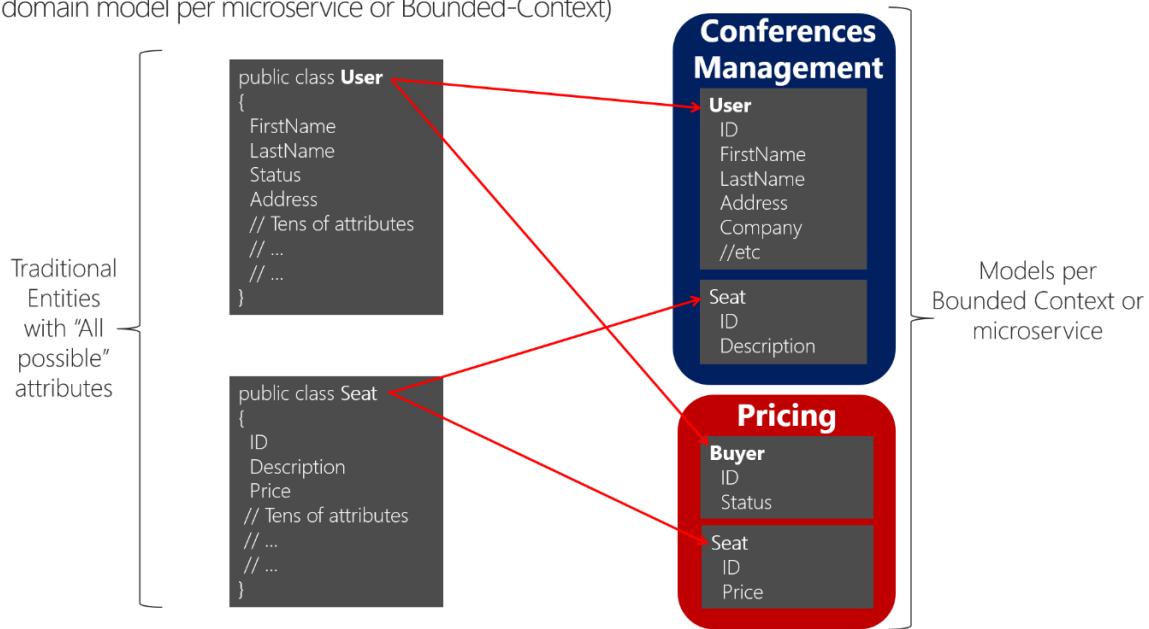


Figura 4-11. Descomponer los modelos de datos tradicionales en varios modelos de dominio

Al descomponer un modelo de datos tradicionales entre contextos limitados, puede tener distintas entidades que comparten la misma identidad (un comprador también es un usuario) con otros atributos en cada contexto delimitado. 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. Puede ser que no todos los microservicios o BC necesiten todos los datos relacionados con una entidad User, solo parte de ellos, según el problema que se deba resolver o el contexto. Por ejemplo, en el modelo del microservicio de precios, no necesita la dirección ni el identificador del usuario, solo el identificador (como identidad) y el estado, que influirán en los descuentos al poner precio al número de puestos por comprador.

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

Básicamente, hay 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, podría haber detalles adicionales o diferentes sobre la entidad de usuario. Por tanto, debe haber una manera de asignar una entidad de usuario de un dominio (microservicio) a otro.

No compartir la misma entidad de usuario con el mismo número de atributos entre dominios tiene varias ventajas. Una ventaja es reducir la duplicación, por lo que los modelos de microservicio no tienen ningún dato que no necesiten. Otra ventaja es tener un microservicio maestro que posee un determinado tipo de datos por entidad para que solo ese microservicio dirija las actualizaciones y las consultas para ese tipo de datos.

Diferencias entre el patrón de puerta de enlace de API y la comunicación directa de cliente a microservicio

04/11/2019 • 32 minutes to read • [Edit Online](#)

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

Comunicación directa de cliente a microservicio

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

Direct Client-To-Microservice communication Architecture

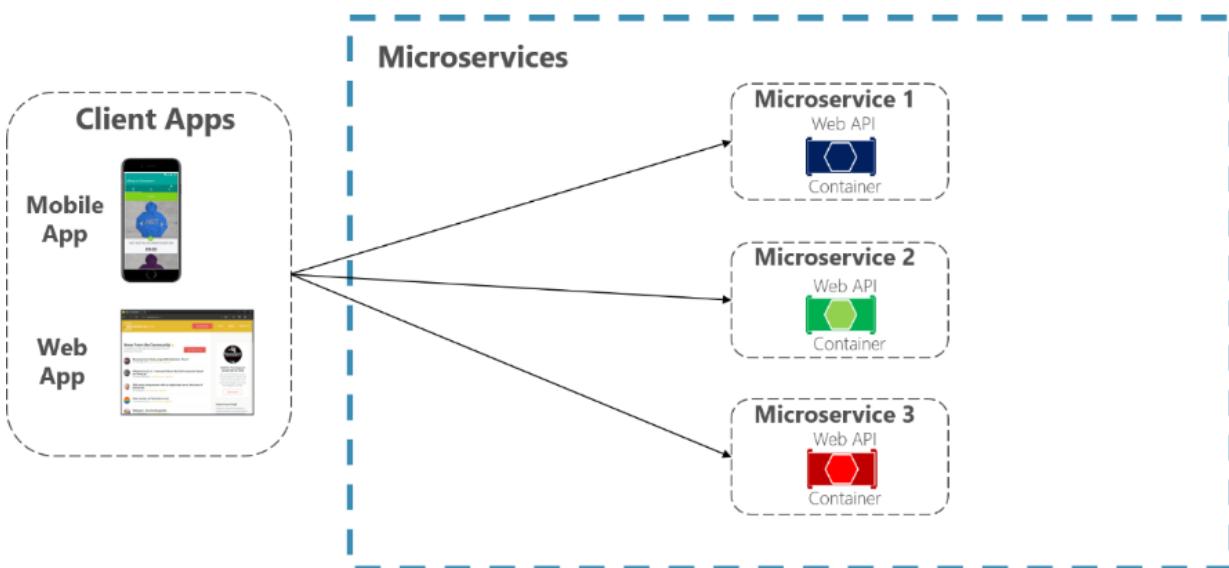


Figura 4-12. Uso de una arquitectura de comunicación directa de cliente a microservicio

En este enfoque, cada microservicio tiene un punto de conexión público, a veces con un puerto TCP distinto para cada microservicio. La siguiente dirección URL de Azure sería un ejemplo de URL de un servicio determinado:

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

En un entorno de producción basado en un clúster, la dirección URL anterior estaría asignada al equilibrador de carga que se utiliza en el clúster, que a su vez distribuye las solicitudes entre los microservicios. En entornos de producción, se puede tener un controlador de entrega de aplicaciones (ADC) como [Azure Application Gateway](#) entre los microservicios e Internet. Este controlador actúa como una capa transparente que no solo realiza el equilibrio de carga, sino que también protege sus servicios, ya que ofrece terminación SSL. Esto mejora la carga de los hosts, puesto que Azure Application Gateway se descarga de la terminación SSL y de otras tareas de enrutamiento que consumen mucha CPU. En cualquier caso, un equilibrador de carga y el ADC son transparentes

desde un punto de vista de la arquitectura de aplicación lógica.

Una arquitectura de comunicación directa de cliente a microservicio podría bastar para una pequeña aplicación basada en microservicio, especialmente si se trata de una aplicación web del lado cliente como, por ejemplo, una aplicación de ASP.NET MVC. Pero al compilar aplicaciones grandes y complejas basadas en microservicios (por ejemplo, al administrar docenas de tipos de microservicio) y, sobre todo, cuando las aplicaciones cliente son aplicaciones móviles remotas o aplicaciones web SPA, este enfoque se enfrenta a algunos problemas.

Al desarrollar una aplicación de gran tamaño basada en microservicios, considere las siguientes preguntas:

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

Interactuar con varios microservicios para crear una única pantalla de interfaz de usuario aumenta el número de recorridos de ida y vuelta a través de Internet. Esto aumenta la latencia y la complejidad en el lado de la interfaz de usuario. Idealmente, las respuestas se deberían agregar eficazmente en el lado del servidor. Esto reduce la latencia, ya que varios fragmentos de datos regresan en paralelo y alguna interfaz de usuario puede mostrar los datos tan pronto como estén listos.

- *¿Cómo se pueden controlar cuestiones transversales como la autorización, las transformaciones de datos y la distribución de solicitudes dinámicas?*

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

- *¿Cómo pueden las aplicaciones cliente comunicarse con servicios que usan protocolos no compatible con Internet?*

Normalmente, los protocolos usados en el lado del servidor (por ejemplo, AMQP o protocolos binarios) no se admiten en aplicaciones cliente. Por lo tanto, las solicitudes deben realizarse a través de protocolos como HTTP/HTTPS y convertirse posteriormente a los demás protocolos. Un enfoque *man-in-the-middle* puede ser útil en esta situación.

- *¿Cómo se puede dar forma a una fachada creada especialmente para las aplicaciones móviles?*

El diseño de la API de varios microservicios podría no adaptarse a las necesidades de diferentes aplicaciones cliente. Por ejemplo, las necesidades de una aplicación móvil pueden ser diferentes a las necesidades de una aplicación web. Para las aplicaciones móviles, la optimización debe ser incluso mayor para que las respuestas de datos sean más eficaces. Para lograr esto, puede agregar los datos de varios microservicios y devolver un único conjunto de datos y, a veces, eliminar los datos de la respuesta que no son necesarios para la aplicación móvil. Y, por supuesto, puede comprimir los datos. Una vez más, una fachada o API entre la aplicación móvil y los microservicios puede ser conveniente 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 generalmente necesitan consumir funcionalidades de más de un microservicio. Si ese consumo se realiza directamente, el cliente debe controlar varias llamadas a los puntos de conexión de microservicio. ¿Qué ocurre cuando la aplicación evoluciona y se introducen nuevos microservicios o se actualizan microservicios existentes? Si la aplicación tiene muchos microservicios, controlar tantos puntos de conexión desde las aplicaciones cliente puede ser una pesadilla. Puesto que la aplicación cliente debería acoplarse a esos puntos de conexión internos, la evolución de los microservicios en el futuro podría provocar un alto impacto para las aplicaciones cliente.

Por lo tanto, disponer de un nivel intermedio o un nivel de direccionamiento indirecto (puerta de enlace) puede ser muy práctico para las aplicaciones basadas en microservicios. Si no dispone de las puertas de enlace de API, las aplicaciones cliente deben enviar solicitudes directamente a los microservicios y eso genera 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 necesitan saber cómo se descomponen las diferentes áreas de la aplicación en microservicios. Al evolucionar y refactorizar los microservicios internos, esas acciones tienen un impacto bastante negativo en el mantenimiento porque provocan cambios bruscos en las aplicaciones cliente debido a la referencia directa a los microservicios internos desde las aplicaciones cliente. Las aplicaciones cliente deben actualizarse con frecuencia, lo que dificulta la evolución de la solución.
- **Demasiados ciclos de ida y vuelta:** Una única pantalla o página en la aplicación cliente puede requerir varias llamadas a varios servicios. Esto puede dar como resultado múltiples recorridos de ida y vuelta entre el cliente y el servidor, lo cual 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 se deben exponer al "mundo externo", haciendo que la superficie del ataque sea mayor que si se ocultan los microservicios internos que las aplicaciones cliente no usan de forma directa. Cuanto menor sea la superficie de ataque, más segura será la aplicación.
- **Intereses transversales:** Cada microservicio publicado públicamente debe ocuparse de cuestiones tales como autorización, SSL, etc. En muchos casos, estas cuestiones podrían controlarse en un solo nivel de manera que se simplifiquen los microservicios internos.

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

Al diseñar y crear aplicaciones basadas en microservicios grandes o complejas con varias aplicaciones cliente, un buen planteamiento podría ser una [puerta de enlace de API](#). Se trata de 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. Actúa como un proxy inverso, enruteando las solicitudes de los clientes a los servicios. También puede proporcionar características transversales adicionales, como autenticación, terminación SSL y caché.

La figura 4-13 muestra el encaje de una puerta de enlace de API personalizada en una arquitectura basada en microservicios simplificada con solo algunos microservicios.

Using a single custom API Gateway service

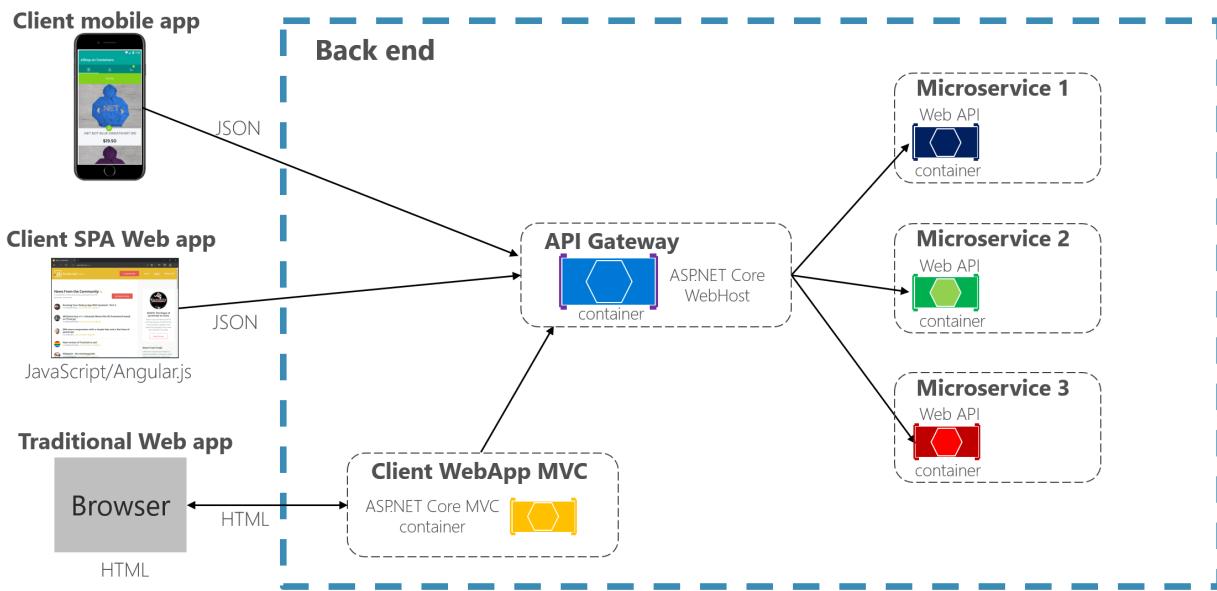


Figura 4-13. Uso de una puerta de enlace de API implementada como un servicio personalizado

Las aplicaciones se conectan a un único punto de conexión (la puerta de enlace de API) configurado para reenviar solicitudes a los microservicios individuales. En este ejemplo, la puerta de enlace de API se implementa como un servicio ASP.NET Core WebHost personalizado que se ejecuta como un contenedor.

Es importante resaltar que en ese diagrama se usa un único servicio de puerta de enlace de API personalizado con conexión a varias aplicaciones cliente distintas. Ese hecho puede suponer un riesgo importante porque el servicio de puerta de enlace de API irá creciendo y evolucionando en función de los muchos requisitos de las aplicaciones cliente. Finalmente, se verá sobredimensionado debido a las distintas necesidades y en la práctica podría ser bastante similar a una aplicación o un servicio monolíticos. Por eso es muy recomendable dividir la puerta de enlace de API en varios servicios o varias puertas de enlace de API más pequeñas, por ejemplo, una por cada tipo de forma de aplicación cliente.

Debe tener cuidado al implementar el patrón de puerta de enlace de API. No suele ser una buena idea de tener una única puerta de enlace de API en la que se agreguen todos los microservicios internos de la aplicación. Si es así, actúa como un orquestador o agregador monolítico e infringe la autonomía de los microservicios al acoplarlos todos.

Por lo tanto, las puertas de enlace de API se deberían segregar en función de los límites del negocio y las aplicaciones cliente no deberían actuar como un simple agregador para todos los microservicios internos.

Al dividir el nivel de puerta de enlace de API en múltiples puertas de enlace de API, si la aplicación tiene varias aplicaciones cliente, puede servir de pivote principal al identificar los múltiples tipos de puertas de enlace de API, de manera que puede tener otra fachada para 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:

Using multiple API Gateways / BFF

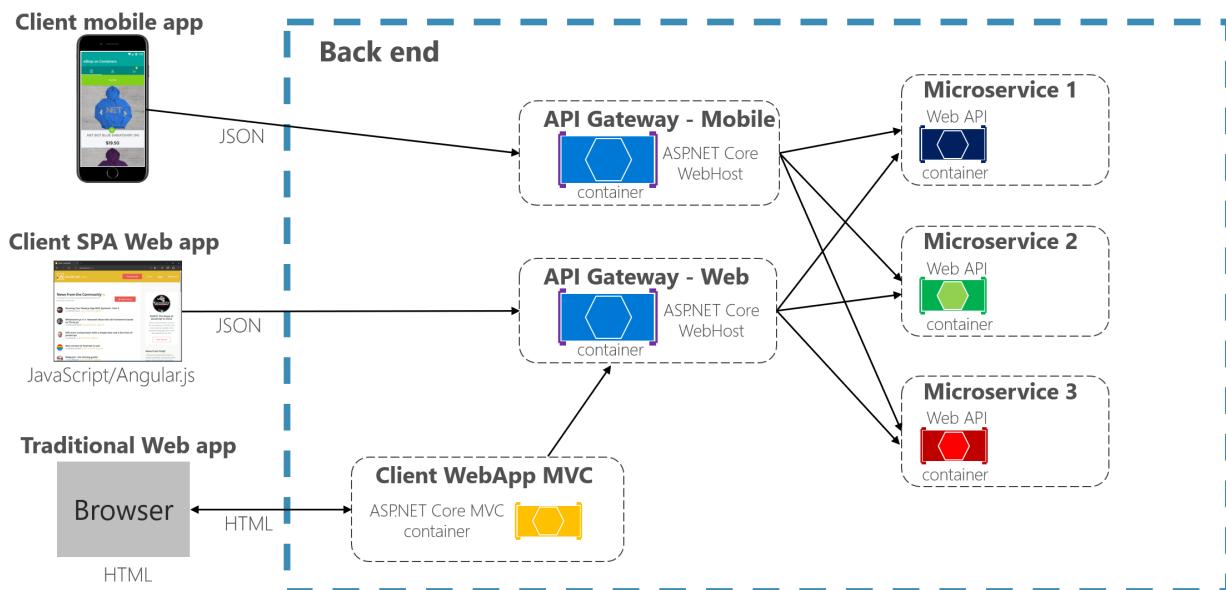


Figura 4-13.1. Uso de varias puertas de enlace de API personalizadas

Figura 4-13.1 que muestra puertas de enlace de API personalizadas, segregadas por tipo de cliente; una para los clientes móviles y otra para los 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 estrictamente en el patrón "back-end para front-end" (BFF); por tanto, se basan solo en la API necesaria para cada aplicación cliente. Pero 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 pivote de diseño.

Características principales en el patrón de puerta de enlace de API

Una puerta de enlace de API puede ofrecer varias características. Dependiendo del producto, podría ofrecer características más completas o más sencillas; sin embargo, las características más importantes y fundamentales para 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 las solicitudes (enrutamiento de capa 7, normalmente solicitudes HTTP) a los puntos de conexión de los microservicios internos. La puerta de enlace proporciona un único punto de conexión o 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 desacoplar las aplicaciones cliente de los microservicios, pero también es bastante 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 notarán 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 por ningún cambio de URI.

Para obtener más información, consulte [Patrón Gateway Routing](#).

Agregación de solicitudes. Como parte del patrón de puerta de enlace, es posible agregar varias solicitudes de cliente (normalmente las solicitudes HTTP) dirigidas a varios microservicios internos en una sola solicitud de cliente. Este patrón es especialmente útil cuando una página o pantalla de cliente necesita información de varios microservicios. Con este enfoque, la aplicación cliente envía una solicitud única a la puerta de enlace de API que

envía varias solicitudes a los microservicios internos y, a continuación, agrega los resultados y envía todo el contenido de nuevo a la aplicación cliente. La ventaja principal y el objetivo de este patrón de diseño radica en la reducción del intercambio de mensajes entre las aplicaciones cliente y la API de back-end, lo cual es especialmente importante para las aplicaciones remotas fuera del centro de datos donde residen los microservicios, como aplicaciones móviles o solicitudes de aplicaciones SPA que provienen de Javascript en exploradores de cliente remotos. En el caso de las aplicaciones web normales que realizan las solicitudes en el entorno del servidor (como una aplicación web ASP.NET Core MVC), este patrón no es tan importante, ya que la latencia es mucho menor que para las aplicaciones cliente remotas.

Dependiendo del producto de puerta de enlace de API que use, es posible que pueda realizar esta agregación. Pero en muchos casos resulta más flexible crear microservicios de agregación en el ámbito de la puerta de enlace de API, de manera que la agregación se define en el código (es decir, código de C#):

Para obtener más información, consulte [Patrón Gateway Aggregation](#).

Cuestiones transversales o descarga de puerta de enlace. Dependiendo 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 consolidando las cuestiones transversales en un nivel. Esto resulta especialmente útil para las características especializadas, que pueden ser bastante complicadas de implementar correctamente en cada microservicio interno, como la funcionalidad siguiente:

- Autenticación y autorización
- Integración del servicio de detección
- Almacenamiento en caché de respuestas
- Directivas de reintento, interruptor y QoS
- Limitación de velocidad
- Equilibrio de carga
- Registro, seguimiento, correlación
- Encabezados, cadenas de consulta y transformación de notificaciones
- Creación de listas blancas IP

Para obtener más información, consulte [Patrón Gateway Offloading](#).

Uso de productos con características de puerta de enlace de API

Puede haber muchas más cuestiones transversales ofrecidas por los productos de las puertas de enlace de API dependiendo de cada implementación. Aquí trataremos los siguientes puntos:

- [Azure API Management](#)
- [Ocelot](#)

Azure API Management

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

API Gateway with Azure API Management

Architecture

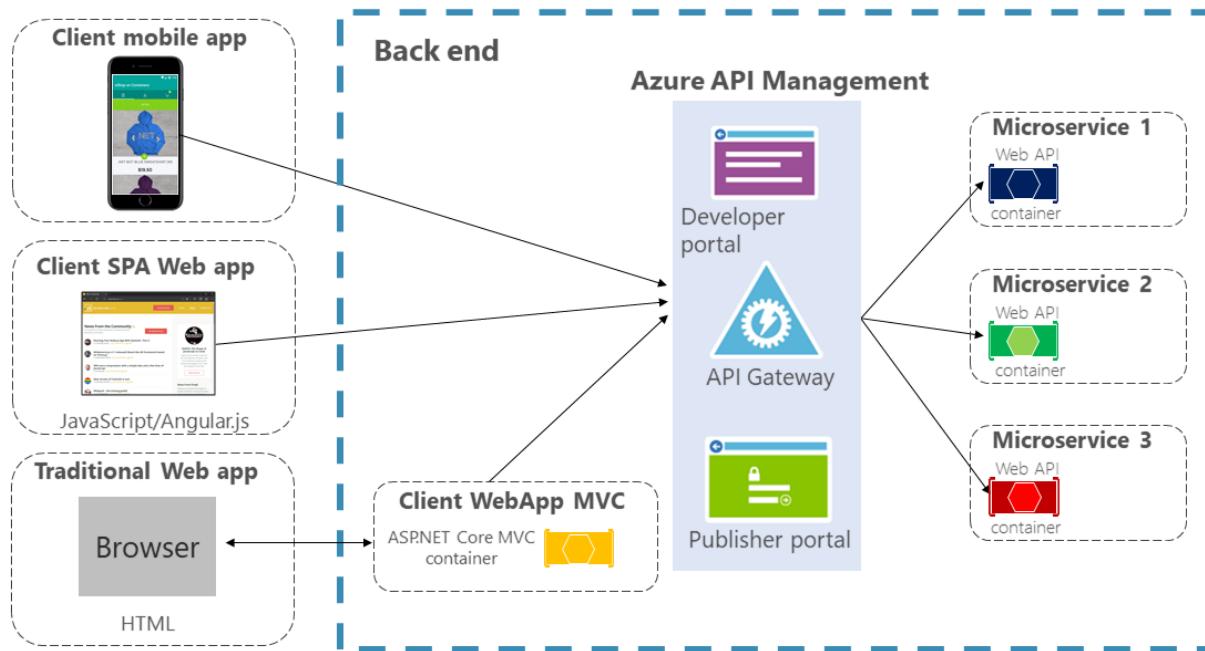


Figura 4-14. Uso de Azure API Management para la puerta de enlace de API

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

Los productos de puerta de enlace de API suelen actuar más como un proxy inverso para la comunicación de entrada, en el que se pueden filtrar las API de los microservicios internos y también aplicar la autorización a las API publicadas en este nivel único.

La información disponible desde un sistema API le ayuda a comprender cómo se están utilizando las API y cuál es su rendimiento. Para ello, le permiten ver informes de análisis prácticamente en tiempo real e identificar las tendencias que podrían afectar a su negocio. Además, puede obtener registros sobre la actividad de solicitudes y respuestas para su posterior análisis en línea y sin conexión.

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

En esta guía y en la aplicación de ejemplo de referencia (eShopOnContainers), nos limitamos a una arquitectura en contenedores más sencilla y personalizada para que pueda centrarse en los contenedores sin formato sin utilizar productos de PaaS como Azure API Management. Pero para las grandes aplicaciones basadas en microservicios que se implementan en Microsoft Azure, le recomendamos que valore Azure API Management como base para las puertas de enlace de API en producción.

Ocelot

Ocelot es una puerta de enlace de API ligera, recomendada para enfoques más simples. Ocelot es una puerta de enlace de API de código abierto basada en .NET Core especialmente diseñada para la arquitectura de microservicios que necesitan puntos de entrada unificados en su sistema. Es ligera, rápida, escalable y proporciona enruteamiento y autenticación, entre muchas otras características.

La razón principal para elegir Ocelot para la [aplicación de referencia eShopOnContainers](#) es porque es una puerta de enlace de API ligera de .NET Core que se puede implementar en el mismo entorno de implementación de aplicaciones en el que se implementan los microservicios y contenedores, como Docker Host, Kubernetes, etc. Y puesto que se basa en .NET Core, es multiplataforma, así que la puede implementar en Linux o Windows.

Los diagramas anteriores que muestran puertas de enlace de API personalizadas que se ejecutan en contenedores son precisamente la forma en que también puede ejecutar Ocelot en una aplicación basada en contenedor y microservicio.

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 iniciales de explicación de arquitectura y patrones, las siguientes secciones 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. Un acoplamiento así podría provocar problemas graves para la aplicación. Clemens Vaster, arquitecto del equipo de Azure Service Bus, se refiere a esta posible dificultad como "el nuevo ESB" en su sesión sobre "[mensajería y microservicios](#)" de GOTO 2016.
- Usar una puerta de enlace de API de microservicios crea un posible único punto de error adicional.
- Una puerta de enlace de API puede incrementar 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 exige un mayor desarrollo y mantenimiento futuro si incluye lógica personalizada y agregación de datos. Los desarrolladores deben actualizar la puerta de enlace de API con el fin de 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 la puerta de enlace de API. Pero si la puerta de enlace de API simplemente aplica seguridad, registro y control de versiones (como al utilizar Azure API Management), este costo de desarrollo adicional podría no ser aplicable.
- Si la puerta de enlace de API ha sido desarrollada por un único equipo, puede haber un cuello de botella de desarrollo. Este es otro de los motivos por el que es más adecuado tener varias puertas de enlace de API específicas que respondan a las distintas necesidades del cliente. También puede separar la puerta de enlace de API internamente en varias áreas o capas que pertenezcan a los diferentes equipos que trabajan en los microservicios internos.

Recursos adicionales

- **Chris Richardson. Patrón: Puerta de enlace de API o back-end para front-end**
<https://microservices.io/patterns/apigateway.html>
- **Puertas de enlace de API**
<https://docs.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. Service Oriented Composition (Composición orientada a servicios)**

<http://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 de API en resumen** (Serie de tutoriales de puerta de enlace de API de ASP.NET Core)

<https://www.pogsdotnet.com/2018/08/api-gateway-in-nutshell.html>

[ANTERIOR](#)

[SIGUIENTE](#)

Comunicación en una arquitectura de microservicio

25/11/2019 • 21 minutes to read • [Edit Online](#)

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. Pueden estar estrechamente acoplados si se crean objetos con código (por ejemplo, `new className()`) o pueden invocarse de forma desacoplada si se usa la inserción de dependencias al hacer referencia a abstracciones en lugar de a instancias de objeto concretas. En cualquier caso, los objetos se ejecutan en el mismo proceso. Lo más complicado a la hora de pasar de una aplicación monolítica a una aplicación basada en microservicios es 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 que conlleva diseñar un sistema distribuido correctamente son tan bien conocidos que incluso existe un canon llamado [Falacias del cómputo distribuido](#) que enumera las expectativas que suelen tener los desarrolladores al migrar de diseños monolíticos a distribuidos.

No existe una única solución, sino varias. Una de ellas implica aislar los microservicios de negocios lo máximo 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 se agrupan las llamadas y se devuelven los 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. Lo habitual es que cada instancia de servicio sea 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 "[puntos de conexión inteligentes y canalizaciones tontas](#)". Este eslogan fomenta un diseño lo más desacoplado posible entre microservicios y lo más cohesionado posible dentro de un único microservicio. Como se ha explicado anteriormente, cada microservicio posee sus propios datos y su propia lógica de dominio. Pero normalmente los microservicios que componen una aplicación de un extremo a otro se establecen sencillamente mediante comunicaciones de REST en lugar de protocolos complejos como WS-* y comunicaciones flexibles controladas por eventos en lugar de orquestadores de procesos de negocios centralizados.

Los dos protocolos que se usan habitualmente son respuesta-solicitud HTTP con API de recurso (sobre todo al consultar) y mensajería asincrónica ligera al comunicar actualizaciones en varios microservicios. Se explican más detalladamente en las secciones siguientes.

Tipos de comunicación

El cliente y los servicios pueden comunicarse a través de muchos tipos diferentes de comunicación, cada uno destinado a un escenario y unos objetivos distintos. 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). Lo 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 de nube) usan mensajes asincrónicos. Normalmente el código de cliente o el

remitente del mensaje no espera ninguna respuesta. Simplemente envía el mensaje al igual que cuando se envía un mensaje a una cola de RabbitMQ o a cualquier otro agente de mensajes.

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

- Receptor único. Cada solicitud debe ser procesada por un receptor o servicio exactamente. 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 asincrónica. Un ejemplo es el mecanismo de [publicación o suscripción](#) empleado en patrones como la [arquitectura controlada por eventos](#). Se basa en una interfaz de bus de eventos o un agente de mensajes para propagar las actualizaciones de datos entre varios microservicios mediante eventos; normalmente se implementa a través de un bus de servicio o algún artefacto similar como [Azure Service Bus](#) mediante [temas y suscripciones](#).

Una aplicación basada en microservicio suele usar una combinación de estos estilos de comunicación. El tipo más común es la comunicación de un único receptor con un protocolo sincrónico como HTTP/HTTPS al invocar a un servicio normal HTTP Web API. Además, los microservicios suelen usar protocolos de mensajería para la comunicación asincrónica entre microservicios.

Resulta útil conocer estos ejes para tener claros los posibles mecanismos de comunicación, aunque no son la preocupación más importante a la hora de compilar microservicios. Al integrar microservicios, no son importantes ni la naturaleza asincrónica de la ejecución de subprocessos de cliente ni la naturaleza asincrónica del protocolo seleccionado. Lo que sí es importante es poder integrar los microservicios de forma asincrónica a la vez que se mantiene su independencia, como se explica en la sección siguiente.

La integración asincrónica del microservicio obliga a su autonomía

Como se ha mencionado, lo importante al compilar una aplicación basada en microservicios es la forma de integrarlos. Lo ideal es intentar minimizar la comunicación entre los microservicios internos. Cuantas menos comunicaciones haya entre microservicios, mejor. Pero en muchos casos tendrá que integrar los microservicios de algún modo. Cuando necesite hacerlo, la regla fundamental es que la comunicación entre los microservicios debe ser asincrónica. Eso no significa que tenga que usar un protocolo determinado (por ejemplo, mensajería asincrónica frente a HTTP sincrónico). Simplemente significa que la comunicación entre los microservicios debe realizarse únicamente mediante la propagación asincrónica de datos, aunque se debe intentar no depender de otros microservicios internos como parte de la operación solicitud-respuesta HTTP del servicio inicial.

Si es posible, no dependa nunca de la comunicación sincrónica (solicitud-respuesta) entre varios microservicios, ni siquiera para las consultas. El objetivo de cada microservicio es ser autónomo y estar a disposición del cliente, aunque los demás servicios que forman parte de la aplicación de un extremo a otro estén inactivos o en mal estado. Si cree que necesita realizar una llamada desde un microservicio a otros (por ejemplo, una solicitud HTTP para una consulta de datos) para poder proporcionar una respuesta a una aplicación cliente, tiene una arquitectura que no resistirá si se producen errores en algunos microservicios.

Además, el tener dependencias HTTP entre microservicios, como al crear largos ciclos de solicitud-respuesta 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 afecta a su rendimiento en cuanto alguno de los servicios de esa cadena no funciona correctamente.

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

Synchronous vs. async communication across microservices

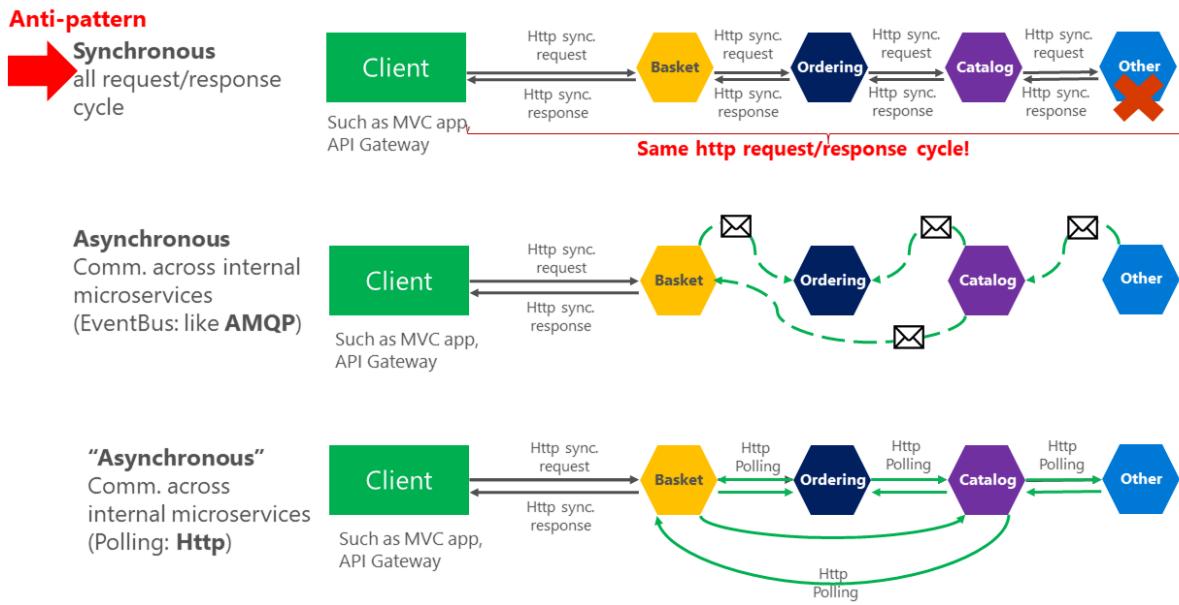


Figura 4-15. Anti-patrones y patrones de comunicación entre microservicios

Tal y como se muestra en el diagrama anterior, en la comunicación sincrónica se crea una "cadena" de solicitudes entre los microservicios mientras se atiende la solicitud del cliente. Esto es un antipatrón. En la comunicación asincrónica los microservicios usan mensajes asincrónicos o sondeo http para comunicarse con otros microservicios, pero la solicitud de cliente se sirve inmediatamente.

Si el microservicio tiene que producir una acción adicional en otro microservicio, siempre que sea posible, no realice esa acción de forma sincrónica como parte de la operación solicitud-respuesta original del microservicio. Por el contrario, hágalo de forma asincrónica (mediante mensajería asincrónica o eventos de integración, colas, etc.). Pero, siempre que sea posible, no invoque a la acción de forma sincrónica como parte de la operación solicitud-respuesta sincrónica original.

Y, por último (y aquí es donde surgen la mayoría de los problemas al compilar microservicios), si el microservicio inicial necesita datos cuyo propietario original es otro microservicio, no dependa de la realización de solicitudes sincrónicas para esos datos. En su lugar, replique o propague esos datos (solo los atributos que necesite) en la base de datos del servicio inicial mediante la coherencia final (normalmente mediante eventos de integración, como se explica en las próximas secciones).

Como se ha indicado anteriormente en la sección [Identificar los límites del modelo de dominio para cada microservicio](#), la duplicación de algunos datos en varios microservicios no es un diseño incorrecto, sino que permite convertir los datos al lenguaje o los términos determinados de ese dominio adicional o contexto enlazado. Por ejemplo, en la [aplicación de eShopOnContainers](#), hay un microservicio denominado identity.api que está a cargo de la mayoría de los datos del usuario con una entidad denominada User. Pero si necesita almacenar datos sobre el usuario en el microservicio Ordering, hágalo como una entidad diferente denominada Buyer. La entidad Buyer comparte la misma identidad con la entidad User original, pero podría tener solo los atributos necesarios para el dominio Ordering y no el perfil completo del usuario.

Podría usar cualquier protocolo para comunicar y propagar datos de forma asincrónica en microservicios para disponer de coherencia final. Como se ha mencionado, puede usar eventos de integración con un bus de eventos o un agente de mensajes o, si no, puede usar incluso HTTP mediante el sondeo de los demás servicios. No importa. Lo importante es no crear dependencias sincrónicas entre los microservicios.

En las siguientes secciones se explican los diversos estilos de comunicación que se pueden usar en una aplicación basada en microservicio.

Estilos de comunicación

Hay muchos protocolos y opciones que se pueden usar para la comunicación, según el tipo de comunicación que se quiera emplear. Si va a usar un mecanismo de comunicación sincrónico basado en solicitud-respuesta, los enfoques de protocolos como HTTP y REST son los más comunes, especialmente si va a publicar los servicios fuera del host de Docker o el clúster de microservicios. Si va a comunicarse entre servicios de forma interna (dentro del host de Docker o el clúster de microservicios), es posible que también quiera usar mecanismos de comunicación de formato binario (como WCF mediante TCP y formato binario). También puede usar mecanismos de comunicación asincrónicos basados en mensajes como AMQP.

Además hay varios formatos de mensaje como JSON o XML, o incluso formatos binarios, que pueden resultar más eficaces. Si el formato binario elegido no es estándar, probablemente no sea buena idea publicar los servicios con ese formato. Puede usar un formato no estándar para la comunicación interna entre los microservicios. Podría hacerlo así para la comunicación entre microservicios dentro del host de Docker o el clúster de microservicios (orquestadores de Docker, por ejemplo) o para las aplicaciones cliente de su propiedad que se comunican con los microservicios.

Comunicación solicitud-respuesta con HTTP y REST

Cuando un cliente usa la comunicación solicitud-respuesta, envía una solicitud a un servicio, este la procesa y luego envía una respuesta. La comunicación solicitud-respuesta resulta especialmente idónea para consultar datos de una interfaz de usuario en tiempo real (una interfaz de usuario activa) desde aplicaciones cliente. Por tanto, en una arquitectura de microservicio probablemente se use este mecanismo de comunicación para la mayoría de las consultas, como se muestra en la figura 4-16.

Request/response communication for live queries and updates HTTP-based Services

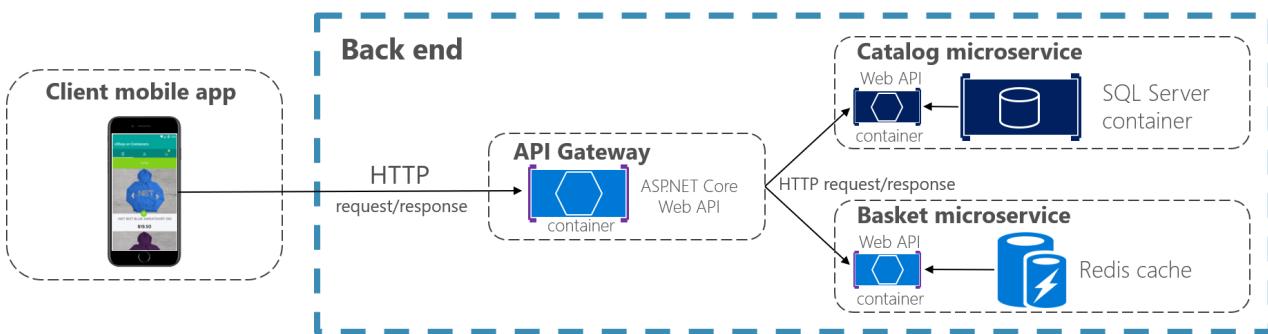


Figura 4-16. Uso de la comunicación solicitud-respuesta HTTP (sincrónica o asincrónica)

Cuando un cliente usa la comunicación solicitud-respuesta, da por hecho que la respuesta llegará en poco tiempo, normalmente en menos de un segundo, o unos pocos segundos como máximo. Si se retrasan las respuestas, debe implementar la comunicación asincrónica basada en [patrones de mensajería](#) y [tecnologías de mensajería](#), que es otro enfoque que se explica en la sección siguiente.

Un estilo arquitectónico popular para la comunicación solicitud-respuesta es [REST](#). Este enfoque se basa en el protocolo [HTTP](#) y está estrechamente relacionado con él, ya que adopta verbos HTTP como GET, POST y PUT. REST es el enfoque de arquitectura de comunicación más usado a la hora de crear servicios. Puede implementar servicios REST cuando desarrolle servicios Web API de ASP.NET Core.

El uso de servicios REST de HTTP como lenguaje de definición de interfaz ofrece algunas ventajas. 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. Richardson Maturity Model** A description of the REST model (Modelo de madurez

Richardson. Una descripción del modelo REST).

<https://martinfowler.com/articles/richardsonMaturityModel.html>

- **Swagger** Sitio oficial.

<https://swagger.io/>

Comunicación de inserción y en tiempo real basada en HTTP

Otra posibilidad (normalmente para fines distintos que REST) es una comunicación en tiempo real y de uno a varios con marcos de trabajo 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 hacer que el código de servidor inserte contenido en los clientes conectados a medida que los datos están disponibles, en lugar de hacer que el servidor espere a que un cliente pida nuevos datos.

Push and real-time communication based on HTTP

One-to-many communication

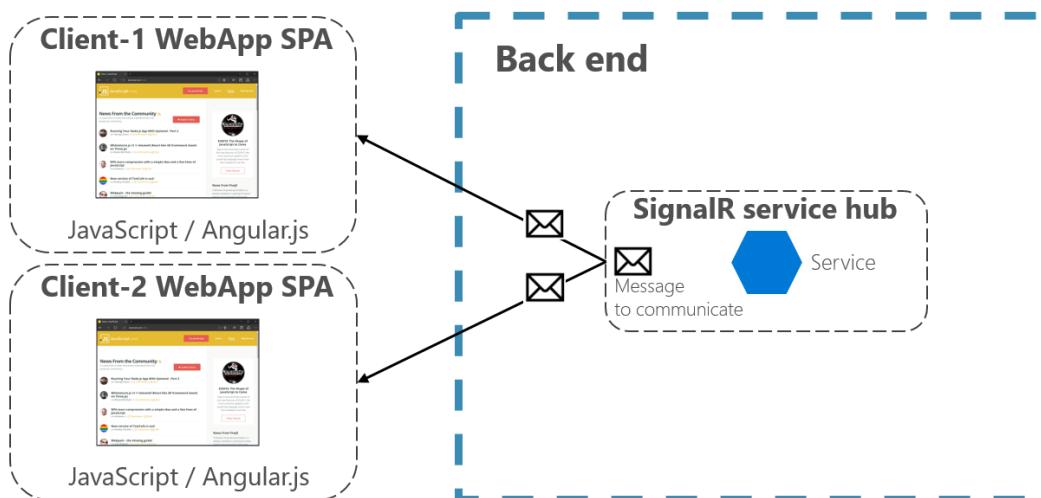


Figura 4-17. Comunicación de mensajes asincrónica en tiempo real uno a uno

SignalR es una buena forma de lograr una comunicación en tiempo real para insertar contenido a los clientes desde un servidor back-end. Puesto que la comunicación es en tiempo real, las aplicaciones cliente muestran los cambios prácticamente de forma inmediata. Normalmente, esto se controla mediante un protocolo como WebSockets, con muchas conexiones WebSockets (una por cliente). Un ejemplo típico es cuando un servicio comunica un cambio en el marcador de un partido a muchas aplicaciones web cliente a la vez.

[ANTERIOR](#)

[SIGUIENTE](#)

Comunicación asincrónica basada en mensajes

04/11/2019 • 16 minutes to read • [Edit Online](#)

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.

Single receiver message-based communication

(i.e. Message-based Commands)

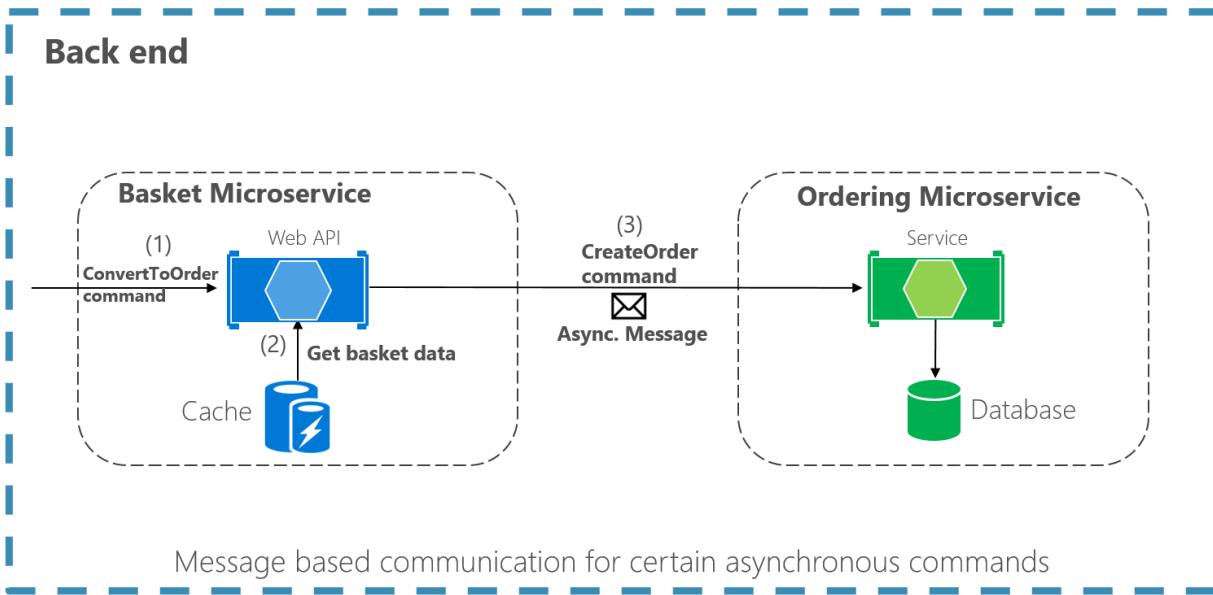


Figura 4-18. Un único microservicio en el que se recibe un mensaje asincrónico

Tenga en cuenta que cuando los comandos proceden de aplicaciones cliente, se pueden implementar como comandos sincrónicos de HTTP. Debe usar comandos basados en mensajes cuando necesite una mayor escalabilidad o cuando ya esté 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 recibirlas 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 normalmente 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 admite 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 totalmente 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. Esto 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.

Asynchronous event-driven communication

Multiple receivers

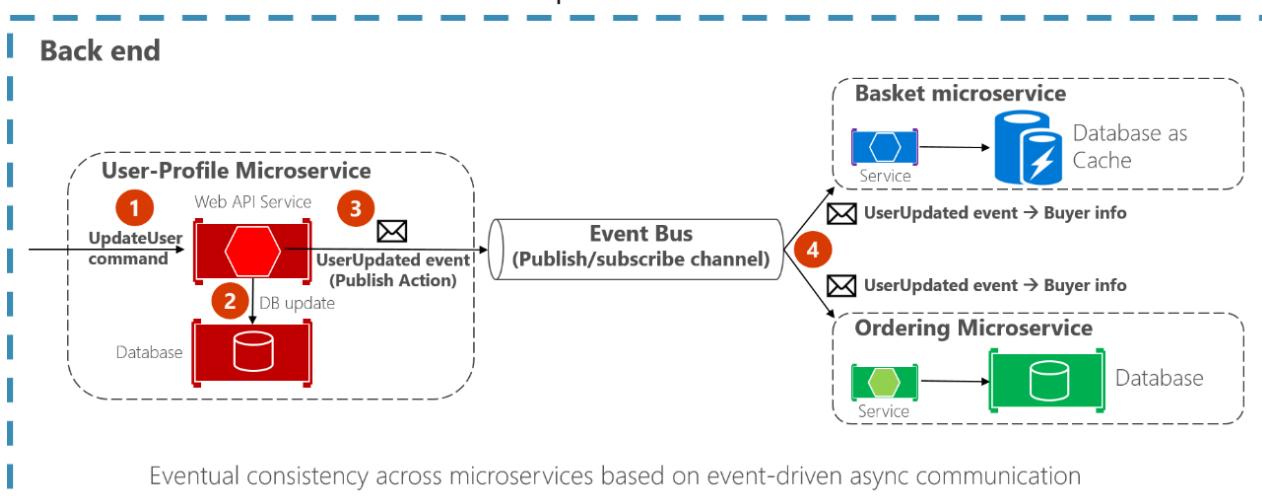


Figura 4-19. Comunicación asincrónica de mensajes controlada por eventos

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 lograrlo, 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.

Temas adicionales que se deben tener en cuenta al usar la comunicación asíncrona 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://soapatterns.org/design_patterns/event_driven_messaging
- **Canal de publicación y suscripción**
<https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- **Udi Dahan. CQRS aclarado**
<http://udidahan.com/2009/12/09/clarified-cqrs/>
- **Patrón Command and Query Responsibility Segregation (CQRS)**
<https://docs.microsoft.com/azure/architecture/patterns/cqrs>
- **Comunicación entre contextos delimitados**
[https://docs.microsoft.com/previous-versions/msp-n-p/jj591572\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-n-p/jj591572(v=pandp.10))
- **Coherencia de los eventos**
https://en.wikipedia.org/wiki/Eventual_consistency
- **Jimmy Bogard. Refactorización hacia la resiliencia: evaluación del acoplamiento**
<https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/>

[ANTERIOR](#)

[SIGUIENTE](#)

Creación, desarrollo y control de versiones de los contratos y las API de microservicio

23/10/2019 • 4 minutes to read • [Edit Online](#)

Una API de microservicio es un contrato entre el servicio y sus clientes. Solo podrá desarrollar un microservicio de forma independiente si no incumple el contrato de su API. Por este motivo el contrato es tan importante. Cualquier cambio en el contrato afectará a sus aplicaciones cliente o a la puerta de enlace de API.

La naturaleza de la definición de API depende del protocolo que esté utilizando. Por ejemplo, si usa mensajería (como [AMQP](#)), la API consiste en los tipos de mensaje. Si usa servicios HTTP y RESTful, la API consiste en las direcciones URL y los formatos JSON de solicitud y respuesta.

Pero, aunque piense en su contrato inicial, una API de servicio debe cambiar con el tiempo. Normalmente, cuando esto ocurre, y especialmente si la API es una API pública utilizada por varias aplicaciones cliente, no puede forzar a todos los clientes a actualizar la versión a su nuevo contrato de API. Lo más habitual es implementar progresivamente nuevas versiones de un servicio, de forma que se ejecuten simultáneamente las versiones anteriores y las nuevas de un contrato de servicio. Por tanto, es importante contar con una estrategia para el control de versiones del servicio.

Cuando los cambios en la API son pequeños, por ejemplo, si agrega atributos o parámetros a la API, los clientes que usen una API anterior deberán cambiar y trabajar con la nueva versión del servicio. Usted puede proporcionar los valores predeterminados para los atributos que falten y que sean necesarios, y los clientes pueden pasar por alto cualquier atributo de respuesta adicional.

Pero en ciertas ocasiones necesitará realizar cambios importantes e incompatibles en una API de servicio. Puesto que es posible que no pueda forzar a los servicios o aplicaciones cliente a que se actualicen inmediatamente a la nueva versión, un servicio debe admitir versiones anteriores de la API durante cierto período de tiempo. Si está utilizando un mecanismo basado en HTTP, como REST, una opción es insertar el número de versión de la API en la dirección URL o en un encabezado HTTP. A continuación, puede decidir si quiere implementar ambas versiones del servicio al mismo tiempo en la misma instancia de servicio o si prefiere implementar distintas instancias y que cada una controle una versión de la API. Una buena opción es utilizar el [patrón mediador](#) (por ejemplo, la [biblioteca MediatR](#)) para desacoplar las diferentes versiones de implementación en los controladores independientes.

Por último, si utiliza una arquitectura REST, [Hypermedia](#) es la mejor solución para controlar las versiones de los servicios y permitir las API avanzadas.

Recursos adicionales

- **Scott Hanselman. 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://docs.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api>
- **Roy Fielding. Control de versiones, hipermedios y REST**
<https://www.infoq.com/articles/roy-fielding-on-versioning>

Direccionabilidad de microservicios y el Registro del servicio

23/10/2019 • 3 minutes to read • [Edit Online](#)

Cada microservicio tiene un nombre único (URL) que se usa para resolver su ubicación. El microservicio debe ser direccionable en cualquier lugar donde se ejecute. Si tiene que pensar en qué equipo se ejecuta un microservicio determinado, todo puede ir mal rápidamente. De la misma manera que DNS resuelve una URL para un equipo en particular, su microservicio debe tener un nombre único para que su ubicación actual sea reconocible. Los microservicios deben tener nombres direccionables que les permitan ser independientes de la infraestructura en que se ejecutan. Esto implica que hay una interacción entre cómo se implementa el servicio y cómo se detecta, porque debe haber un [Registro del servicio](#). Del mismo modo, cuando se produce un error en un equipo, el servicio del Registro debe ser capaz de indicar que el servicio se está ejecutando.

El [patrón del Registro del servicio](#) es una parte fundamental de la detección de servicios. El Registro es una base de datos que contiene las ubicaciones de red de las instancias del servicio. Un Registro del servicio debe estar muy disponible y actualizado. Los clientes podrían almacenar en caché las ubicaciones de red obtenidas del Registro del servicio. Sin embargo, esa información finalmente se queda obsoleta y los clientes ya no pueden detectar las instancias del servicio. Por tanto, un Registro del servicio consta de un clúster de servidores que usan un protocolo de replicación para mantener su coherencia.

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

Recursos adicionales

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

[ANTERIOR](#)

[SIGUIENTE](#)

Creación de interfaces de usuario compuestas basadas en microservicios

23/10/2019 • 5 minutes to read • [Edit Online](#)

A menudo, la arquitectura de microservicios se inicia con el control de los datos y la lógica del lado servidor pero, en muchos casos, la interfaz de usuario se controla todavía como un monolito. Sin embargo, un enfoque más avanzado, llamado **micro front-end**, consiste en diseñar la interfaz de usuario de la aplicación también en función de los microservicios. Esto significa tener una interfaz de usuario compuesta generada 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 crea pueden completarse con representación lógica y visual.

En la figura 4-20 se muestra el enfoque más sencillo que consiste en simplemente consumir microservicios desde una aplicación cliente monolítica. Por supuesto, también podría tener un servicio de ASP.NET MVC que produzca HTML y JavaScript. La figura es una simplificación que resalta que tiene una sola (monolítica) interfaz de usuario cliente 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).

Monolithic UI consuming microservices

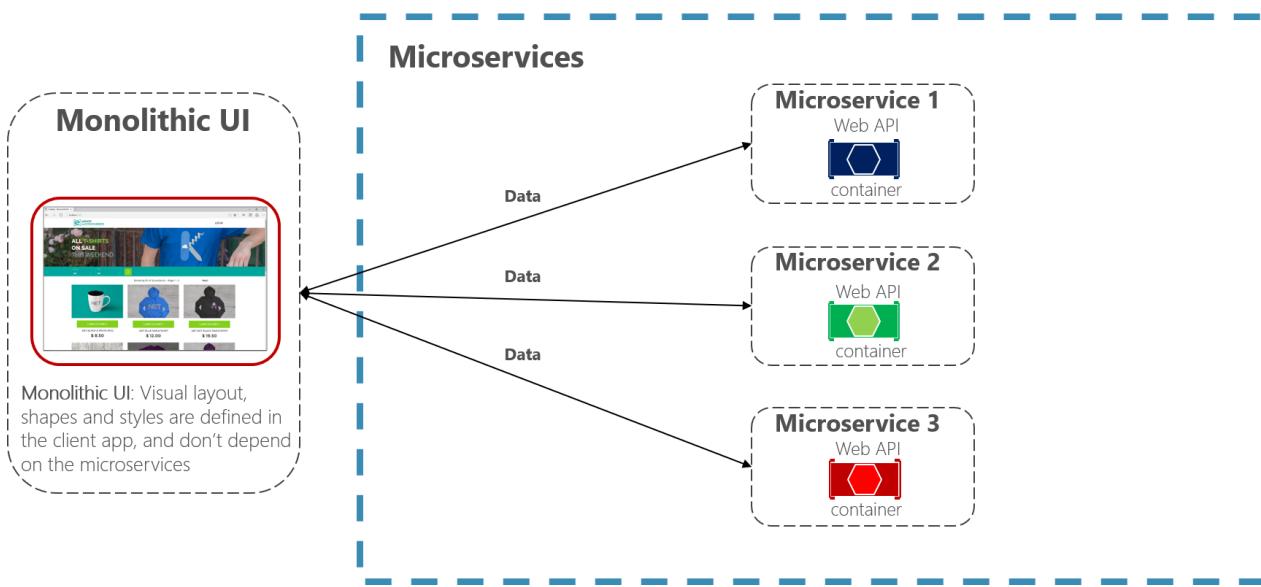


Figura 4-20. Una aplicación de interfaz de usuario monolítica que consume microservicios de back-end

En contraste, los propios microservicios generan y componen con precisión una interfaz de usuario compuesta. Algunos de los microservicios controlan la forma visual de áreas específicas de la interfaz de usuario. La principal diferencia es que tiene componentes de la interfaz de usuario cliente (por ejemplo, clases de TypeScript) basados en plantillas, y el ViewModel de la interfaz de usuario que perfila los datos para esas plantillas procede de cada microservicio.

En el momento de iniciarse la aplicación cliente, cada uno de los componentes de la interfaz de usuario cliente (por ejemplo, las clases de TypeScript) 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. Esto se simplifica

porque es posible que tenga otros microservicios que agreguen elementos pormenorizados basados en otras técnicas. Depende de si va a crear un enfoque web tradicional (ASP.NET MVC) o una SPA (aplicación de página única).

Composite UI generated by microservices

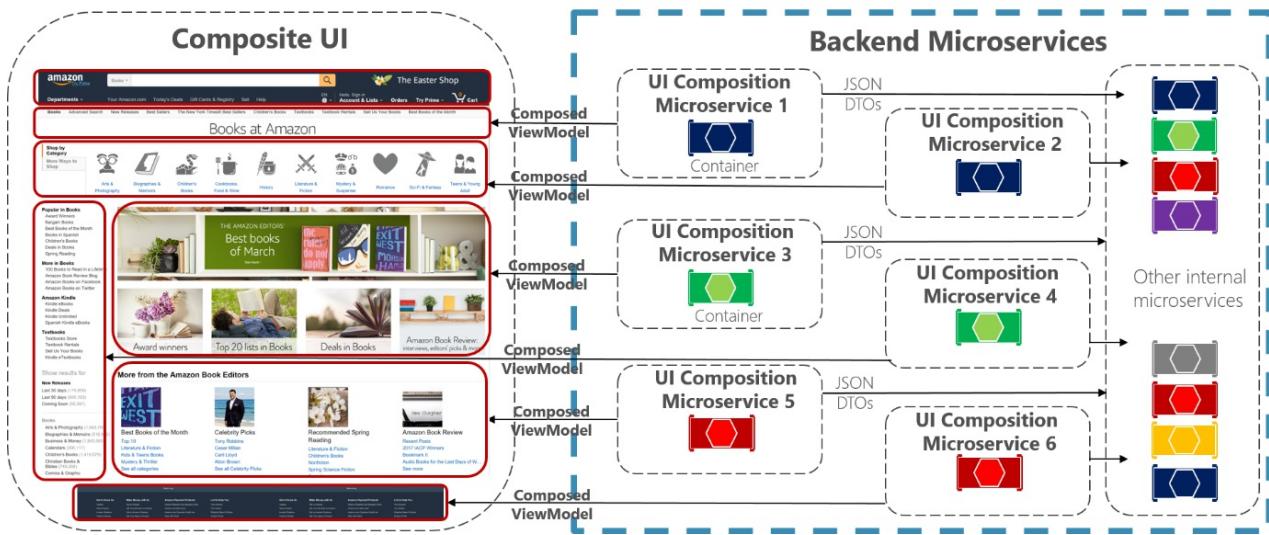


Figura 4-21. Ejemplo de una aplicación de interfaz de usuario compuesta formada por microservicios de backend

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

Un enfoque de interfaz de usuario compuesta controlada por microservicios puede ser más o menos complicado, según las tecnologías de interfaz de usuario que se usen. Por ejemplo, no usará las mismas técnicas para crear una aplicación web tradicional que para crear una SPA o una aplicación móvil nativa (como al desarrollar aplicaciones de Xamarin, que puede ser más complicado para este enfoque).

La aplicación de ejemplo [eShopOnContainers](#) usa el enfoque de interfaz de usuario monolítica por distintos motivos. En primer lugar, es una introducción a los microservicios y los contenedores. Una interfaz de usuario compuesta es más avanzada, pero también necesita mayor complejidad al diseñar y desarrollar la interfaz de usuario. En segundo lugar, eShopOnContainers también proporciona una aplicación móvil nativa basada en Xamarin, lo que resultaría más complejo en el lado del cliente C#.

Le recomendamos que use las siguientes referencias para saber más información sobre la interfaz de usuario compuesta basada en microservicios.

Recursos adicionales

- **Micro front-end (blog de Martin Fowler)**
<https://martinfowler.com/articles/micro-frontends.html>
- **Micro front-end (sitio de Michael Geers)**
<https://micro-frontends.org/>
- **Composición de la interfaz de usuario con ASP.NET (taller de Particular)**
<https://github.com/Particular/Workshop/tree/master/demos/asp-net-core>
- **Ruben Oostinga. El 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. Inclusión de componentes web de front-end en los 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>

[ANTERIOR](#)

[SIGUIENTE](#)

Resistencia y alta disponibilidad en microservicios

04/11/2019 • 13 minutes to read • [Edit Online](#)

Tratar errores inesperados es uno de los problemas más difíciles de resolver, especialmente en un sistema distribuido. Gran parte del código que los desarrolladores escriben implica controlar las excepciones, y aquí también es donde se dedica más tiempo a las pruebas. El problema es más complejo que escribir código para controlar los errores. ¿Qué ocurre cuando se produce un error en la máquina en que se ejecuta el microservicio? No solo es necesario detectar este error de microservicio (un gran problema de por sí), sino también contar con algo que reinicie su microservicio.

Un microservicio debe ser resistente a errores y poder reiniciarse a menudo en otra máquina a efectos de disponibilidad. Esta resistencia también se refiere al estado que se guardó en nombre del microservicio, en los casos en que el estado se puede recuperar a partir del microservicio, y al hecho de si el microservicio puede reiniciarse correctamente. En otras palabras, debe haber resistencia en la capacidad de proceso (el proceso puede reiniciarse en cualquier momento), así como en el estado o los datos (sin pérdida de datos y que se mantenga la consistencia de los datos).

Los problemas de resistencia se agravan durante otros escenarios, como cuando se producen errores durante la actualización de una aplicación. El microservicio, trabajando con el sistema de implementación, debe determinar si puede avanzar a la versión más reciente o, en su lugar, revertir a una versión anterior para mantener un estado consistente. Deben tenerse en cuenta cuestiones como si están disponibles suficientes máquinas para seguir avanzando y cómo recuperar versiones anteriores del microservicio. Esto requiere que el microservicio emita información de mantenimiento para que la aplicación en conjunto y el orquestador puedan tomar estas decisiones.

Además, la resistencia está relacionada con cómo deben comportarse los sistemas basados en la nube. Como se ha mencionado, un sistema basado en la nube debe estar preparado para los errores e intentar recuperarse automáticamente de ellos. Por ejemplo, en caso de errores de red o de contenedor, las aplicaciones de cliente o los servicios de cliente deben disponer de una estrategia para volver a intentar enviar mensajes o solicitudes, ya que en muchos casos, los errores en la nube son parciales. En la sección [Implementar aplicaciones resistentes](#) de esta guía se explica cómo controlar errores parciales. Se describen técnicas como los reintentos con retroceso exponencial o el patrón de interruptor en .NET Core en que se usan bibliotecas como [Polly](#), que ofrece una gran variedad de directivas para controlar este asunto.

Administración del estado y diagnóstico en microservicios

Puede parecer obvio, y a menudo se pasa por alto, pero un microservicio debe notificar su estado y diagnóstico. En caso contrario, hay poca información desde una perspectiva operativa. Correlacionar eventos de diagnóstico en un conjunto de servicios independientes y tratar los desajustes en el reloj de la máquina para dar sentido al orden de los eventos suponen un reto. De la misma manera que interactúa con un microservicio según protocolos y formatos de datos acordados, hay una necesidad de estandarizar cómo registrar los eventos de estado y diagnóstico que, en última instancia, terminan en un almacén de eventos para que se consulten y se vean. En un enfoque de microservicios, es fundamental que distintos equipos se pongan de acuerdo en un formato de registro único. Debe haber un enfoque coherente para ver los eventos de diagnóstico en la aplicación.

Comprobaciones de estado

El estado es diferente del diagnóstico. El estado trata de cuando el microservicio informa sobre su estado actual para que se tomen las medidas oportunas. Un buen ejemplo es trabajar con los mecanismos de actualización e implementación para mantener la disponibilidad. Aunque un servicio podría actualmente estar en mal estado debido a un bloqueo de proceso o un reinicio de la máquina, puede que el servicio siga siendo operativo. Lo

último que debe hacer es realizar una actualización que empeore esta situación. El mejor método consiste en realizar una investigación en primer lugar o dar tiempo a que el microservicio se recupere. Los eventos de estado de un microservicio nos ayudan a tomar decisiones informadas y, en efecto, ayudan a crear servicios de reparación automática.

En la sección [Implementación de comprobaciones de estado en servicios de ASP.NET Core](#) de esta guía se explica cómo usar una nueva biblioteca de ASP.NET HealthChecks en sus microservicios para que puedan informar sobre su estado a un servicio de supervisión para que se tomen las medidas oportunas.

También tiene la opción de usar una biblioteca de código abierto excelente llamada Beat Pulse, que está disponible en [GitHub](#) y como [paquete NuGet](#). Además, la biblioteca realiza comprobaciones de estado, concretamente de dos tipos:

- **Ejecución:** comprueba si el microservicio se está ejecutando, es decir, si puede aceptar solicitudes y responder a estas.
- **Preparación:** comprueba si las dependencias del microservicio, como la base de datos o los servicios de cola, están listas, de modo que el microservicio pueda funcionar como debería.

Utilización de secuencias de eventos de diagnóstico y registro

Los registros ofrecen información sobre cómo se ejecuta una aplicación o un servicio, incluidos las excepciones, las advertencias y los mensajes informativos simples. Normalmente, cada registro se presenta en un formato de texto con una línea por evento, aunque las excepciones también suelen mostrar el seguimiento de la pila en varias líneas.

En las aplicaciones monolíticas basadas en servidor, puede simplemente escribir registros en un archivo en disco (un archivo de registro) y, a continuación, analizarlo con cualquier herramienta. Puesto que la ejecución de la aplicación se limita a un servidor o una máquina virtual fijos, por lo general no es demasiado complejo analizar el flujo de eventos. Sin embargo, en una aplicación distribuida en la que se ejecutan varios servicios a través de muchos nodos en un clúster de orquestador, poder correlacionar los eventos distribuidos supone un reto.

Una aplicación basada en microservicio no debe intentar almacenar la secuencia de salida de eventos o archivos de registro por sí misma y ni siquiera intentar administrar el enruteamiento de los eventos a una ubicación central. Debe ser transparente, lo que significa que cada proceso solo debe escribir su secuencia de eventos en una salida estándar que la infraestructura de entorno de ejecución donde se está ejecutando recopilará por debajo. Un ejemplo de estos enruteadores de secuencia de eventos es [Microsoft.Diagnostic.EventFlow](#), que recopila secuencias de eventos de varios orígenes y las publica en sistemas de salida. Estos pueden incluir salidas estándar simples 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 otros fabricantes que pueden buscar, alertar, informar y supervisar registros, incluso en tiempo real, como [Splunk](#).

Cómo los orquestadores administran la información sobre el estado y el diagnóstico

Crear una aplicación basada en microservicio implica enfrentarse a cierto grado de complejidad. Por supuesto, un único microservicio es fácil 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 del microservicio; también necesita alta disponibilidad, capacidad de direccionamiento, resistencia, estado y diagnóstico si pretende disponer de un sistema estable y cohesivo.



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

Es muy difícil que pueda resolver por sí mismo los problemas complejos que se muestran en la figura 4-22. Los equipos de desarrollo deben centrarse en solucionar problemas empresariales y crear aplicaciones personalizadas con enfoques basados en microservicio. No deben centrarse en solucionar problemas de infraestructura complejos; si fuera así, el coste de cualquier aplicación basada en microservicio sería enorme. Por tanto, hay plataformas orientadas a microservicios, denominadas orquestadores o clústeres de microservicio, que tratan de solucionar los problemas complejos de crear y ejecutar un servicio y usar de forma eficaz los recursos de infraestructura. Esto reduce las complejidades de crear aplicaciones que usan un enfoque de microservicios.

Distintos orquestadores podrían parecer similares, pero las comprobaciones de diagnóstico y estado que ofrece cada uno de ellos difieren en las características y el estado de madurez, y a veces dependen de la plataforma del sistema operativo, como se explica en la sección siguiente.

Recursos adicionales

- La aplicación **Twelve-Factor**. XI. Logs: Treat logs as event streams (Registros: tratar los registros como secuencias de eventos)
<https://12factor.net/logs>
- Repositorio de GitHub **Microsoft Diagnostic EventFlow Library**.
<https://github.com/Azure/diagnostics-eventflow>
- ¿Qué es **Azure Diagnostics**?
<https://docs.microsoft.com/azure/azure-diagnostics>
- Conectar equipos Windows con el servicio **Azure Monitor**
<https://docs.microsoft.com/azure/azure-monitor/platform/agent-windows>
- Registrar lo importante: usar el bloque de aplicación de registro semántico
[https://docs.microsoft.com/previous-versions/msp-n-p/dn440729\(v=pandp.60\)](https://docs.microsoft.com/previous-versions/msp-n-p/dn440729(v=pandp.60))
- Sitio oficial de **Splunk**.
<https://www.splunk.com/>
- API **EventSource Class** para el seguimiento de eventos para Windows (ETW)
<https://docs.microsoft.com/dotnet/api/system.diagnostics.tracing.eventsource>

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

04/11/2019 • 17 minutes to read • [Edit Online](#)

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).

Composed Docker Applications in a Cluster

- For each service instance you use one container
- Docker images/containers are “units of deployment”
- A container is an instance of a Docker Image
- A host (VM/server) handles many containers

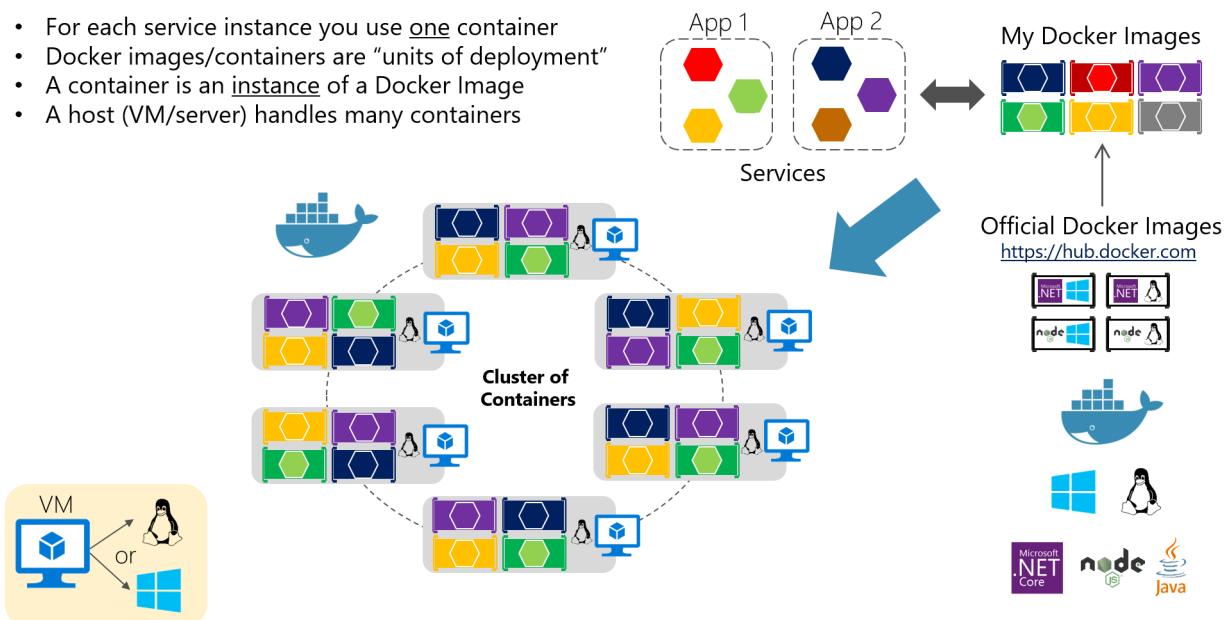


Figura 4-23. Un clúster de contenedores

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 automáticamente inicie contenedores, escala horizontalmente contenedores con varias instancias por imagen, los suspenda o los cierre cuando sea necesario y, a ser posible, también 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 muy 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* significa tener la capacidad de que un administrador inicie los 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

Kubernetes 	<p><i>Kubernetes</i> 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.</p> <p><i>Kubernetes</i> proporciona una infraestructura centrada en el contenedor que agrupa los contenedores de la aplicación en unidades lógicas para facilitar la administración y detección.</p> <p><i>Kubernetes</i> está más desarrollado en Linux que en Windows.</p>
Azure Kubernetes Service (AKS) 	<p><i>AKS</i> es un servicio administrado de orquestación de contenedores de Kubernetes en Azure que simplifica la administración, implementación y operaciones del clúster de Kubernetes.</p>

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.

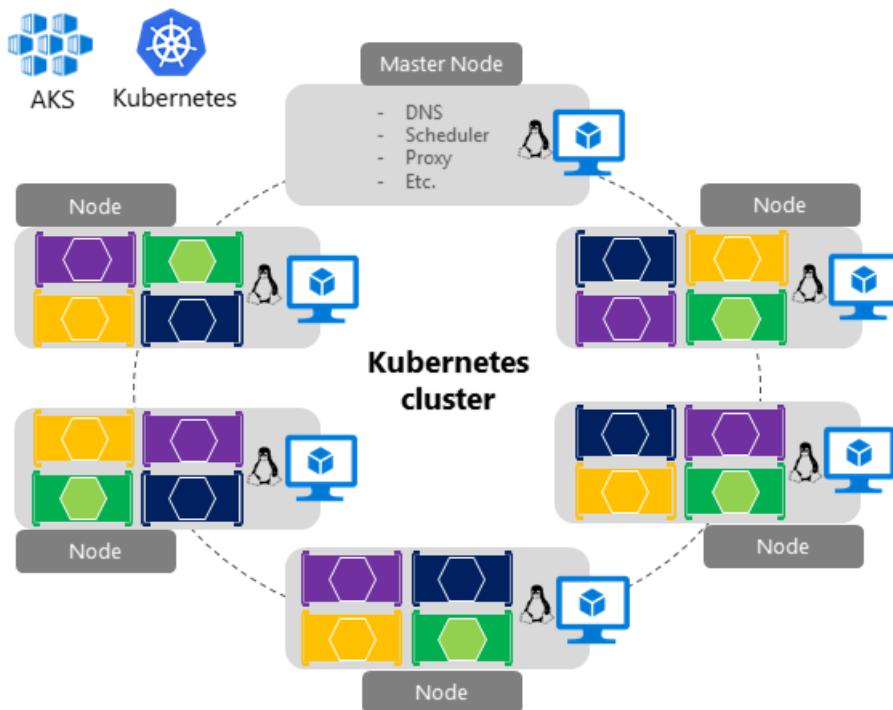


Figura 4-24. Topología y estructura simplificada del clúster de Kubernetes

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 y puede implementar contenedores en el resto de los nodos que se administran como un único grupo desde un punto de vista de la aplicación y permite escalar 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 puede ejecutarse en un único equipo de desarrollo (Windows 10 o macOS), basta con instalar [Docker Desktop](#). Puede implementar posteriormente en la nube (AKS) para obtener más pruebas de integración, como se muestra en la figura 4-25.

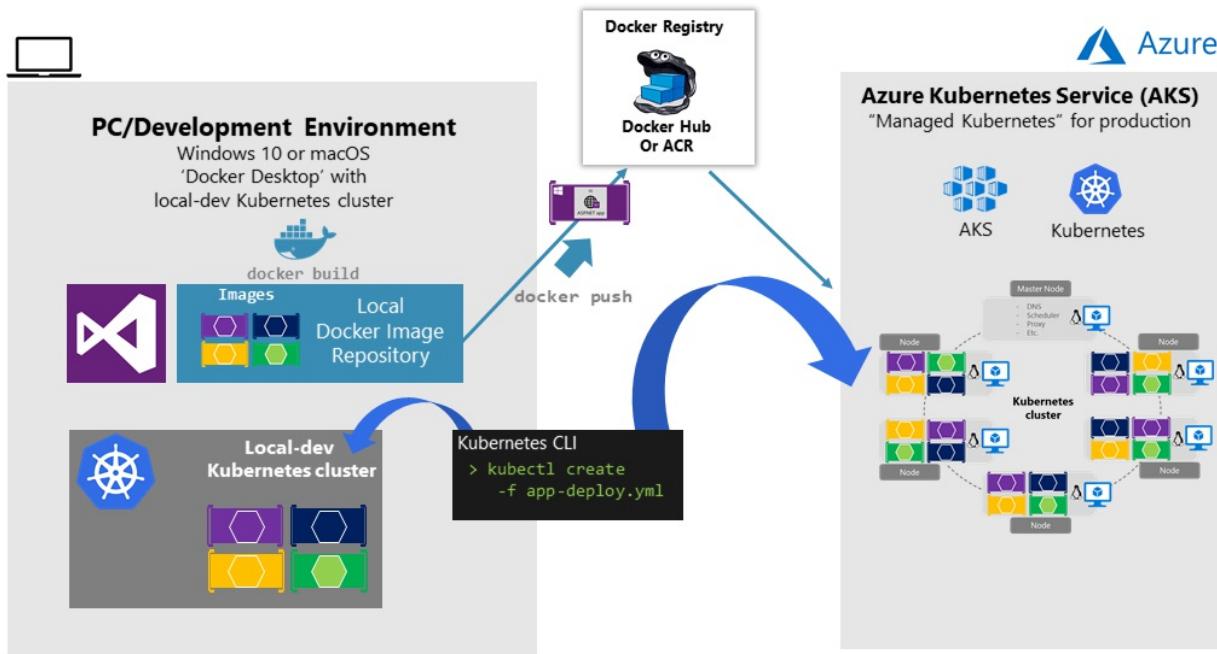


Figura 4-25. Ejecución de Kubernetes en el equipo de desarrollo y la nube

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 como, por ejemplo, la red y el almacenamiento. No hay ningún cargo incremental para AKS.

Para obtener más información de implementación sobre la implementación en Kubernetes en función de `kubectl` y archivos .yaml originales, vea la publicación [Setting eShopOnContainers up in AKS \(Azure Kubernetes Service\) \(Configuración de eShopOnContainers en AKS \[Azure Kubernetes Service\]\)](#).

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, instalación, 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 [Using Helm Charts to deploy eShopOnContainers to AKS](#) (Uso de gráficos de Helm para implementar eShopOnContainers en AKS).

Uso de Azure Dev Spaces para el ciclo de vida de la aplicación de Kubernetes

Azure Dev Spaces proporciona una experiencia de desarrollo de Kubernetes rápida e iterativa para los equipos. Con una configuración de máquina de desarrollo mínima, puede ejecutar y depurar contenedores de forma iterativa directamente en Azure Kubernetes Service (AKS). Desarrolle en Windows, Mac o Linux mediante herramientas familiares como Visual Studio, Visual Studio Code o la línea de comandos.

Como se mencionó, Kubernetes utiliza los gráficos de Helm al implementar las aplicaciones basadas en contenedores.

Azure Dev Spaces ayuda a los equipos de desarrollo a ser más productivos en Kubernetes porque permite iterar rápidamente y depurar código directamente en un clúster de Kubernetes global en Azure usando simplemente Visual Studio 2017 o Visual Studio Code. Ese clúster de Kubernetes en Azure es un clúster de Kubernetes administrado compartido, por lo que su equipo puede colaborar. Puede desarrollar código de forma aislada, después implementarlo en el clúster global y realizar pruebas de un extremo a otro con otros componentes sin tener que replicar ni simular dependencias.

Como se muestra en la figura 4-26, la característica distintiva de Azure Dev Spaces es la capacidad de crear "espacios" que se pueden ejecutar integrados al resto de la implementación global en el clúster.

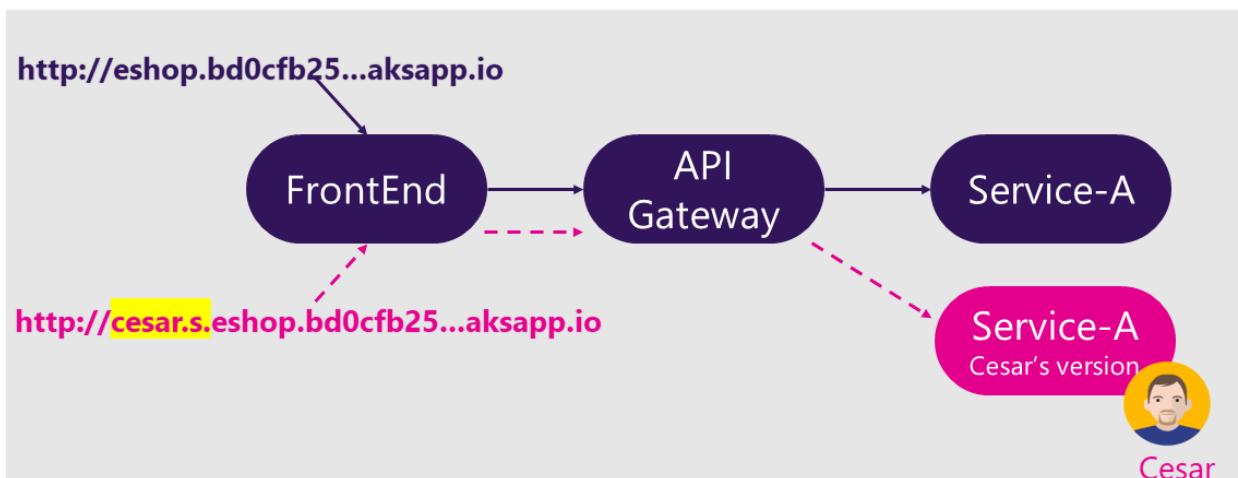


Figura 4-26. Uso de varios espacios en Azure Dev Spaces

Básicamente, puede configurar un espacio de desarrollo compartido en Azure. Así, cada programador se puede centrar exclusivamente en su parte de la aplicación y desarrollar de forma iterativa código previo a la confirmación en un espacio de desarrollo que ya contenga todos los demás servicios y recursos en la nube de los que sus escenarios dependen. Las dependencias siempre estarán actualizadas y los desarrolladores trabajarán de una manera que se asemeja bastante a un entorno de producción.

En Azure Dev Spaces existe el concepto de espacio, que permite trabajar de manera relativamente aislada y sin riesgo de romper el trabajo del equipo. Cada espacio de desarrollo forma parte de una estructura jerárquica que le permite invalidar un microservicio (o muchos), desde el espacio de desarrollo maestro "top", con su propio microservicio en curso.

Esta característica se basa en los prefijos de dirección URL, por lo que al utilizar cualquier prefijo de espacio de desarrollo en la dirección url, se sirve una solicitud desde el microservicio de destino si existe en el espacio de desarrollo, de lo contrario, se reenvía hasta la primera instancia del microservicio de destino que se encuentra en la jerarquía, que finalmente llega al espacio de desarrollo maestro en la parte superior.

Puede ver la [página wiki de eShopOnContainers en Azure Dev Spaces](#) para obtener una vista práctica en un ejemplo concreto.

Para obtener más información, vea el artículo sobre [Desarrollo en equipo con Azure Dev Spaces](#).

Recursos adicionales

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

[ANTERIOR](#)

[SIGUIENTE](#)

Proceso de desarrollo de aplicaciones basadas en Docker

08/01/2020 • 4 minutes to read • [Edit Online](#)

Desarrolle aplicaciones .NET en contenedor de la forma que prefiera, ya sea centradas en el IDE con Visual Studio y Visual Studio Tools para Docker o bien centradas en la CLI o el editor con la CLI de Docker y Visual Studio Code.

Entorno de desarrollo para aplicaciones de Docker

Opciones de herramientas de desarrollo: IDE o editor

Tanto si quiere un IDE eficaz y completo como si prefiere un editor ligero y ágil, Microsoft dispone de herramientas que puede usar para desarrollar aplicaciones de Docker.

Visual Studio (para Windows). Cuando desarrolle aplicaciones basadas en Docker con Visual Studio, se recomienda usar la versión 15.7 de Visual Studio 2017 o versiones posteriores, que incluyen herramientas de 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 (de un solo contenedor o de varios contenedores) directamente en un host de Docker, o bien presionar CTRL+F5 para editar y actualizar la aplicación sin tener que volver a compilar el contenedor. Se trata de la opción de desarrollo más eficaz para aplicaciones basadas en Docker.

Visual Studio para Mac. Es un IDE, evolución de Xamarin Studio, que se ejecuta en macOS y es compatible con Docker desde mediados de 2017. Debe ser la opción preferida para los desarrolladores que trabajan en equipos macOS y que también quieran usar un IDE eficaz.

Visual Studio Code y la CLI de Docker. Si prefiere un editor ligero multiplataforma que admita todos los lenguajes de programación, puede usar Microsoft Visual Studio Code (VS Code) y la CLI de Docker. Se trata de 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.

Mediante la instalación de [Docker Desktop Community Edition \(CE\)](#), 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/>
- **Visual Studio Code.** Sitio oficial.
<https://code.visualstudio.com/download>
- **Docker Desktop for Windows Community Edition (CE)**
<https://hub.docker.com/editions/community/docker-ce-desktop-windows>
- **Docker Desktop for Mac Community Edition (CE)**
<https://hub.docker.com/editions/community/docker-ce-desktop-mac>

Lenguajes y marcos de .NET para contenedores de Docker

Como se ha mencionado en secciones anteriores de esta guía, puede usar .NET Framework, .NET Core o el

proyecto Mono de código abierto para desarrollar aplicaciones .NET en contenedor de Docker. Puede desarrollar en C#, F# o Visual Basic cuando tenga como destino contenedores de Windows o Linux, en función de qué versión de .NET Framework esté en uso. Para obtener más información sobre lenguajes de .NET, vea la entrada de blog [The .NET Language Strategy](#) (Estrategia de lenguaje de .NET).

[ANTERIOR](#)

[SIGUIENTE](#)

Flujo de trabajo de desarrollo para aplicaciones de Docker

25/11/2019 • 52 minutes to read • [Edit Online](#)

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:

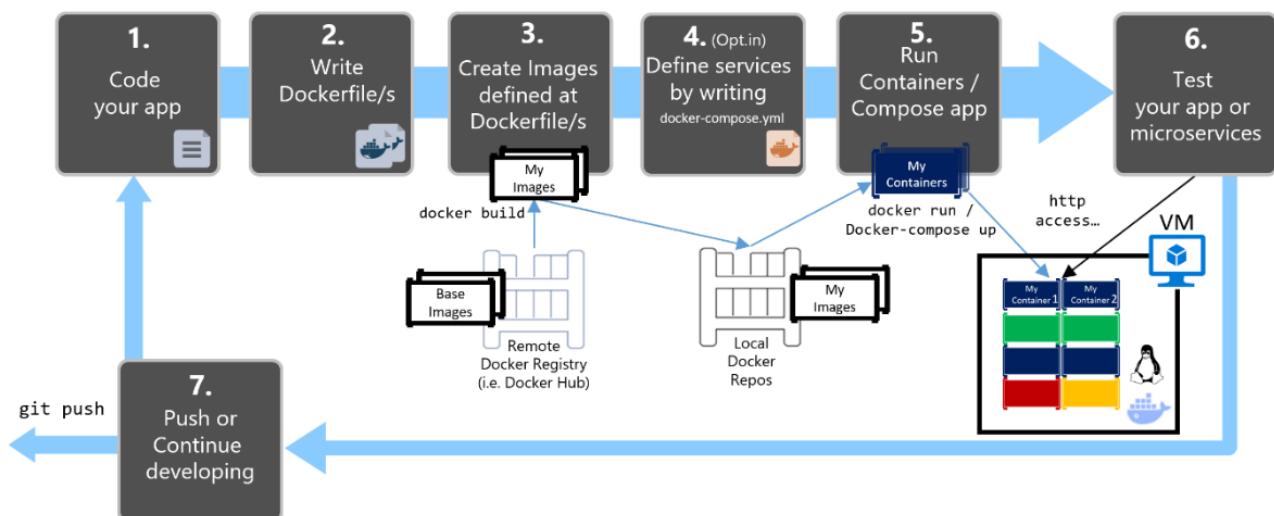
- Una selección de sistema operativo, por ejemplo, una distribución de Linux, Windows Nano Server o Windows Server Core.
- Archivos agregados durante el desarrollo, por ejemplo, archivos binarios de código fuente y aplicación.
- Información de configuración, como configuración de entorno y dependencias.

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.

Inner-Loop development workflow for Docker apps



Proceso de desarrollo de aplicaciones de Docker: 1. Programar la aplicación, 2. Escribir Dockerfiles, 3. Crear imágenes definidas en Dockerfiles, 4. (Opcional) Crear servicios en el archivo docker-compose.yml, 5. Ejecutar contenedor o aplicación docker-compose, 6. Probar la aplicación o los microservicios, 7. Insertar en el repositorio

y repetir.

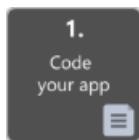
Figura 5-1. Flujo de trabajo paso a paso para el desarrollo de aplicaciones en contenedor de Docker

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).

Cuando se usa Visual Studio 2017, muchos de esos pasos se controlan de forma automática, lo que mejora considerablemente la productividad. Esto es así especialmente con Visual Studio 2017 y cuando el destino son aplicaciones de varios contenedores. Por ejemplo, con un solo clic, Visual Studio agrega el 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 Community Edition \(CE\)](#) para Windows, como se explica en estas instrucciones:

[Get started with Docker CE for Windows \(Introducción a Docker CE para Windows\)](#)

Además, se necesita Visual Studio 2017 versión 15.7 o posterior con la carga de trabajo **Desarrollo multiplataforma de .NET Core** instalada, como se muestra en la figura 5-2.

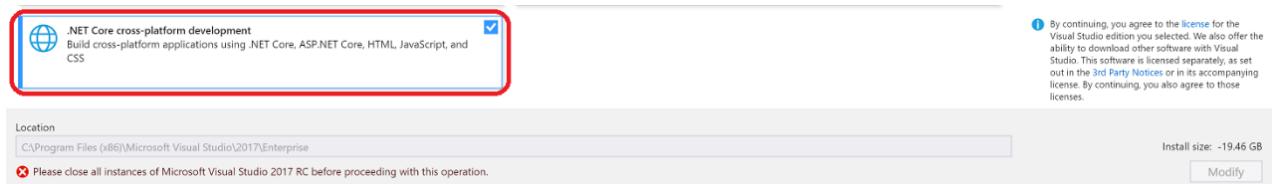


Figura 5-2. Selección de la carga de trabajo **Desarrollo multiplataforma de .NET Core** durante la instalación de Visual Studio 2017

Puede empezar a programar la aplicación en .NET sin formato (normalmente en .NET Core 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

- **Get started with Docker CE for Windows** (Introducción a Docker CE para Windows)

<https://docs.docker.com/docker-for-windows/>

- **Visual Studio 2017**

<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 nuevo proyecto en Visual Studio 2017, hay una opción denominada **Enable Container (Docker) Support** (Habilitar compatibilidad con contenedor (Docker)), como se muestra en la figura 5-3.

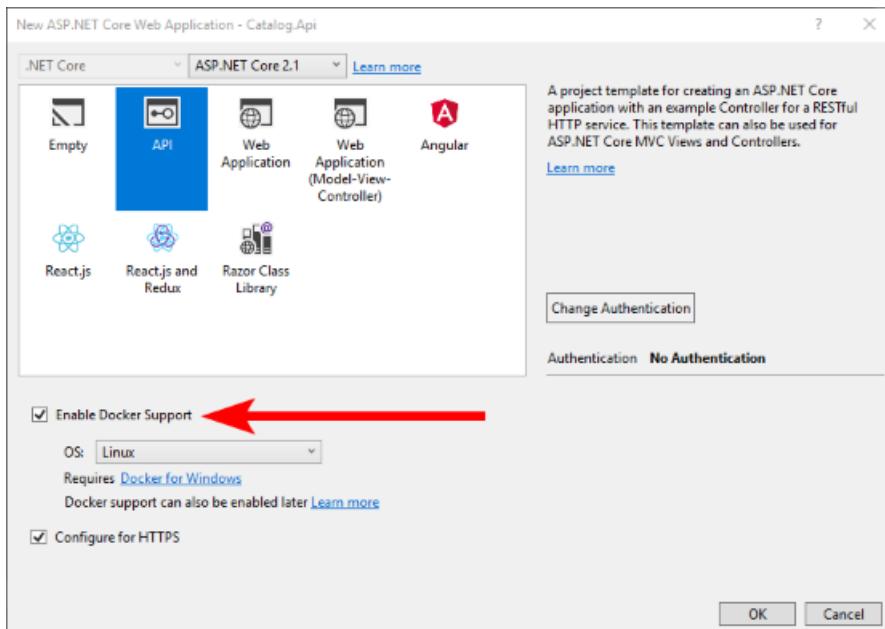


Figura 5-3. Activación de la compatibilidad con Docker al crear un nuevo proyecto de ASP.NET Core en Visual Studio 2017

También puede habilitar la compatibilidad con Docker en un proyecto de aplicación web de ASP.NET Core existente si hace clic con el botón derecho en el proyecto en el **Explorador de soluciones** y selecciona **Agregar**

> **Compatibilidad con Docker**, como se muestra en la figura 5-4.

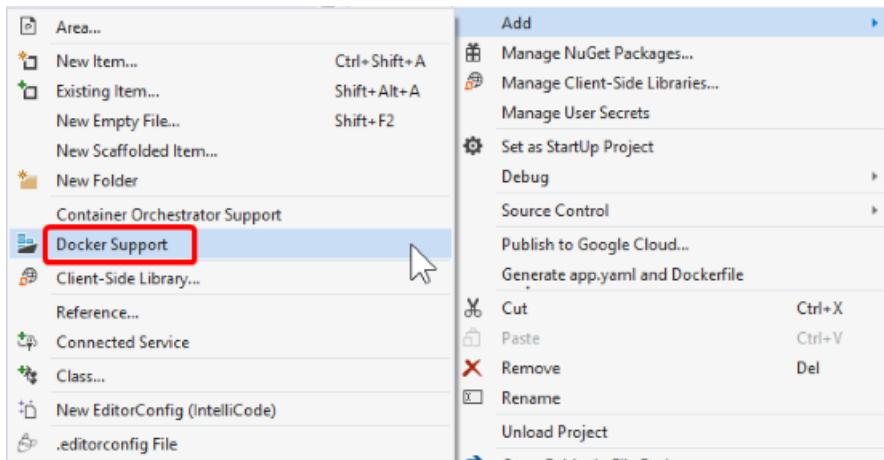


Figura 5-4. Habilitación de la compatibilidad con Docker en un proyecto existente de Visual Studio 2017

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 > Container Orchestrator Support** (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 `aspnetcore` 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/core/aspnet:2.2`. 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/core/aspnet:2.2` 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.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:2.2
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 2.2 de la imagen de Docker de ASP.NET Core oficial (multiarquitectura para Linux y Windows). Es el valor `FROM mcr.microsoft.com/dotnet/core/aspnet:2.2`. [Para obtener más información sobre esta imagen base, consulte la página [.NET Core Docker Image](#) (Imagen de Docker de .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

Por ejemplo, la línea ENTRYPPOINT con `["dotnet", "MySingleContainerWebApp.dll"]` indica a Docker que ejecute una aplicación .NET Core. Si usa el SDK y la CLI de .NET Core (dotnet CLI) para compilar y ejecutar la aplicación .NET, este valor sería diferente. La conclusión es que la línea ENTRYPPOINT 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 .NET Core**
<https://docs.microsoft.com/dotnet/core/docker/building-net-docker-images>
- **Compile su propia imagen.** En la documentación oficial de Docker.
<https://docs.docker.com/engine/tutorials/dockerimages/>
- **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 Docker: 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 `dotnet/core` disponible en el registro 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:

- `microsoft/dotnet:2.2-aspnetcore-runtime-stretch-slim`
Destinos: solo entorno de ejecución .NET Core 2.2 en Linux
- `microsoft/dotnet:2.2-aspnetcore-runtime-nanoserver-1809`
Destinos: solo entorno de ejecución .NET Core 2.2 en Windows Nano Server

Pero, si se especifica el mismo nombre de imagen, incluso con la misma etiqueta, las imágenes multiarquitectura (como la imagen `aspnetcore`) 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:

- `microsoft/dotnet:2.2-aspnetcore-runtime`
Arquitectura múltiple: solo el entorno de ejecución .NET Core 2.2 en Linux o Windows Nano Server en función del 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:

```
1 FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS base
2 WORKDIR /app
3 EXPOSE 80
4
5 FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS build
6 WORKDIR /src
7 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:

```
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:

```
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 `/ignorereprojectextensions:.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:

```
1 FROM mcr.microsoft.com/dotnet/core/aspnet:2.2 AS base
2 WORKDIR /app
3 EXPOSE 80
4
5 FROM mcr.microsoft.com/dotnet/core/sdk:2.2 AS publish
6 WORKDIR /src
7 COPY . .
8 RUN dotnet restore /ignorereprojectextensions:.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
13 WORKDIR /app
14 COPY --from=publish /app
15 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
121d7eeff6c20: 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 determinada. 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   49aaaf5daa850  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 en tiempo 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:

```

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=sql.data;Port=1433;Database=CatalogDB;...
    ports:
      - "81:80"
    depends_on:
      - sql.data

  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=sql.data;Database=OrderingDb;...
    ports:
      - "82:80"
    extra_hosts:
      - "CESARDLBOOKVHD:10.0.75.1"
    depends_on:
      - sql.data

  sql.data:
    image: mssql-server-linux:latest
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"

```

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, `sql.data`, 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 2017

Además de agregar un Dockerfile a un proyecto, como se ha mencionado antes, Visual Studio 2017 (a partir de 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.

Tiene que repetir 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 Docker Compose y Service Fabric.

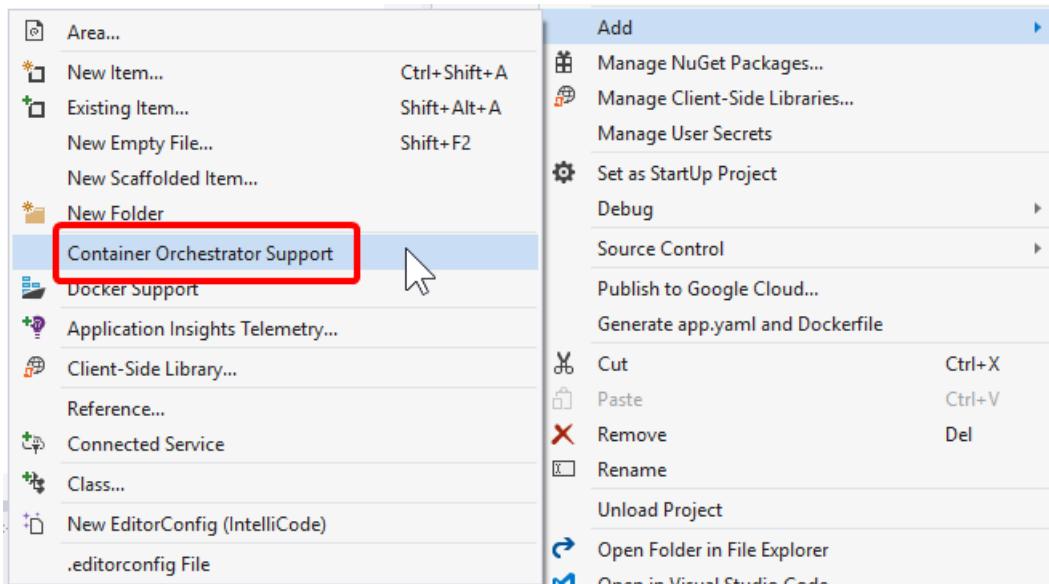


Figura 5-7. Adición de compatibilidad con Docker en Visual Studio 2017 al hacer clic con el botón derecho en un proyecto de ASP.NET Core

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.

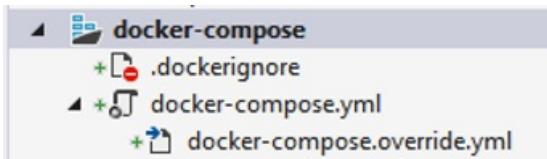


Figura 5-8. Nodo de árbol docker-compose agregado en el Explorador de soluciones de Visual Studio 2017

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). Pero si la aplicación contiene varios servicios, se puede implementar como una 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:

```
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.

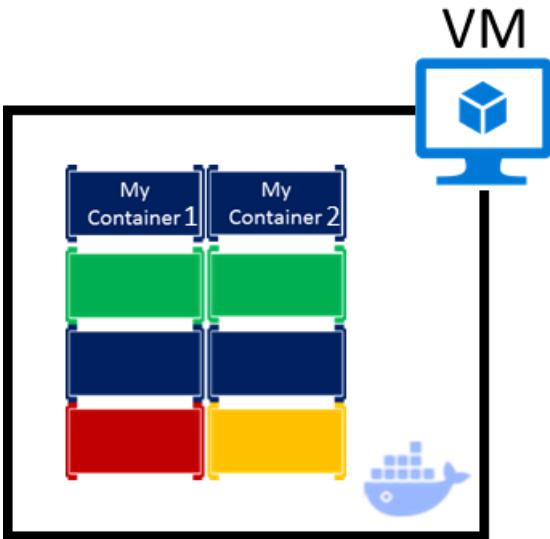


Figura 5-10. Máquina virtual con contenedores de Docker implementados

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 2017 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", simplemente así.

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 2017 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.

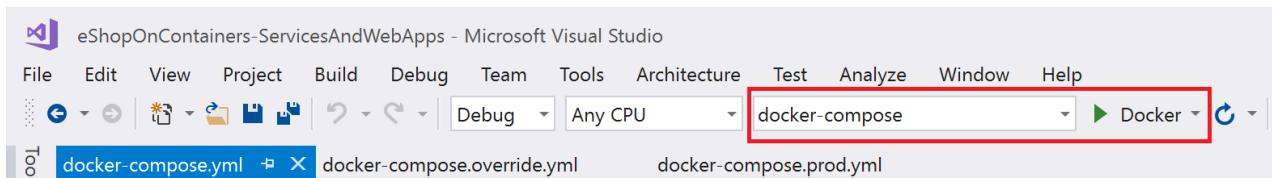


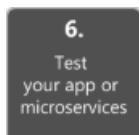
Figura 5-12. Ejecución de aplicaciones de varios contenedores en Visual Studio 2017

Recursos adicionales

- **Implementación de un contenedor de ASP.NET en un host remoto de Docker**
<https://docs.microsoft.com/azure/vs-azure-tools-docker-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 Core 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).

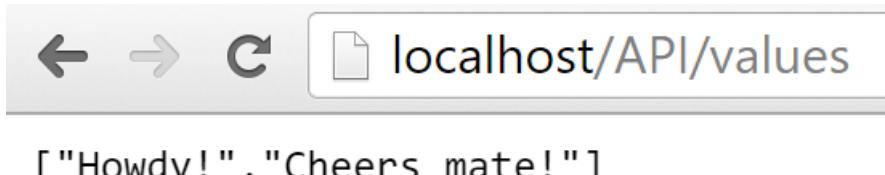


Figura 5-13. Ejemplo de prueba de la aplicación de Docker en local mediante localhost

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.

Tenga en cuenta que 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

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 2017

Al ejecutar y depurar los contenedores con Visual Studio 2017, puede depurar la aplicación de .NET prácticamente de la misma manera que como lo haría al ejecutar sin contenedores.

Pruebas y depuración sin Visual Studio

Si está desarrollando con el enfoque de editor/CLI, la depuración de contenedores es más difícil y se prefiere hacer mediante la generación de seguimientos.

Recursos adicionales

- **Depuración de aplicaciones en un contenedor de Docker local**

<https://docs.microsoft.com/visualstudio/containers/edit-and-refresh>

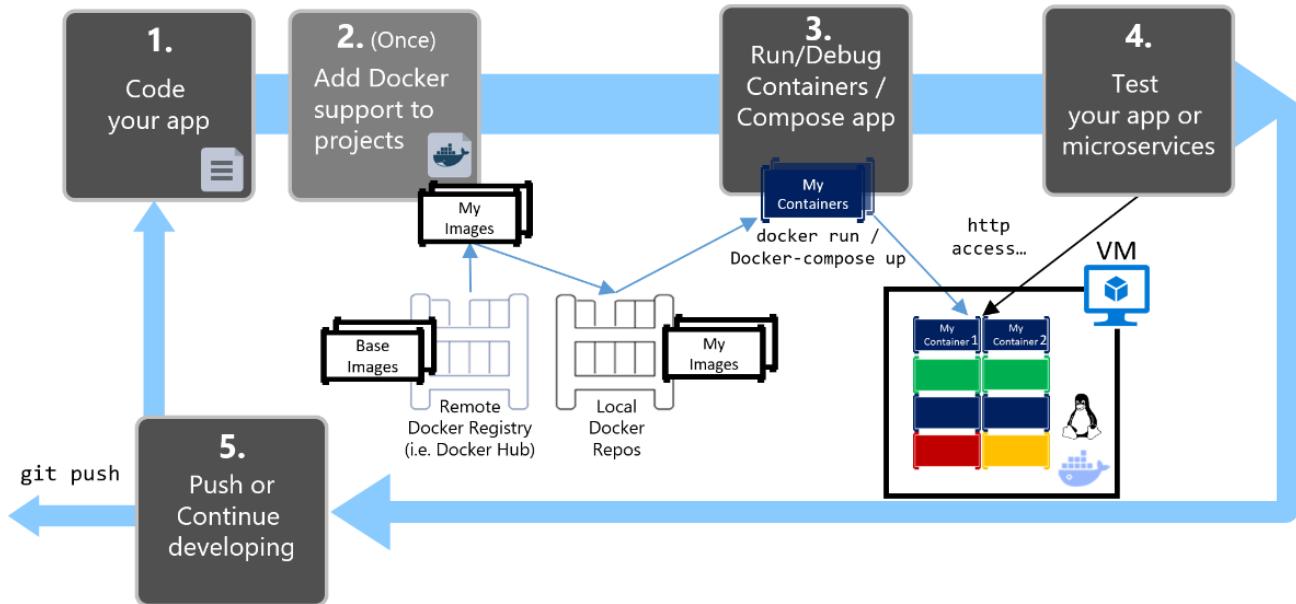
- **Steve Lasker. Compilar, depurar e implementar aplicaciones ASP.NET Core con Docker.** Video.

<https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T115>

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.

VS development workflow for Docker apps



Proceso de desarrollo de aplicaciones de Docker: 1. Programar la aplicación, 2. Escribir Dockerfiles, 3. Crear imágenes definidas en Dockerfiles, 4. (Opcional) Crear servicios en el archivo docker-compose.yml, 5. Ejecutar contenedor o aplicación docker-compose, 6. Probar la aplicación o los microservicios, 7. Insertar en el repositorio y repetir.

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.

Recursos adicionales

- **Desarrollo de Docker de .NET con Visual Studio 2017, de Steve Lasker**
<https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T111>

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:

```
FROM microsoft/windowsservercore
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 4.6 o cualquier otro software de Windows. Por ejemplo, el siguiente comando en un Dockerfile configura ASP.NET 4.5:

```
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-Itsc2016/runtime/Dockerfile>

[ANTERIOR](#)

[SIGUIENTE](#)

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

23/10/2019 • 2 minutes to read • [Edit Online](#)

Si desarrolla aplicaciones de microservicios en contenedor significa que está compilando aplicaciones de varios contenedores. Pero 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 "¿Se necesita Docker para compilar una arquitectura de microservicios?". La respuesta es un no rotundo. Docker es un habilitador y puede proporcionar grandes ventajas, pero los contenedores y Docker no son un requisito imprescindible para los microservicios. Por ejemplo, podría crear una aplicación basada en microservicios con o sin Docker al usar Azure Service Fabric, que es compatible con los microservicios que se ejecutan como procesos simples o como contenedores de Docker.

Pero si sabe cómo diseñar y desarrollar una aplicación basada en microservicios que también se base en contenedores de Docker, podrá diseñar y desarrollar cualquier modelo de aplicación más sencillo. Por ejemplo, podría 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 de los contenedores, esta sección se centra en la implementación de una arquitectura de microservicios con contenedores de Docker.

[ANTERIOR](#)

[SIGUIENTE](#)

Diseño de una aplicación orientada a microservicios

25/11/2019 • 31 minutes to read • [Edit Online](#)

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

Especificaciones de la aplicación

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

La aplicación constará de estos tipos de componentes:

- Componentes de presentación. Son los responsables del control de la interfaz de usuario y el consumo de servicios remotos.
- Lógica de dominio o de negocios. Se trata de la lógica de dominio de la aplicación.
- Lógica de acceso a bases de datos. Son los componentes de acceso a datos responsables de obtener acceso a las bases de datos (SQL o NoSQL).
- Lógica de integración de aplicaciones. Incluye un canal de mensajería, basado principalmente en agentes de mensajes.

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

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

Contexto del equipo de desarrollo

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

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

Elección de una arquitectura

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

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

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

Los microservicios se desarrollan e implementan como contenedores de forma independiente entre ellos. Esto 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 permite separarlo totalmente de otros microservicios. Cuando sea necesario, la coherencia entre las bases de datos de los diferentes microservicios se logra mediante eventos de integración de nivel de aplicación (a través de un bus de eventos lógicos), como se controla en Command and Query Responsibility Segregation (CQRS). Por ese motivo, las restricciones de negocio deben adoptar la coherencia final entre los múltiples microservicios y bases de datos relacionadas.

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

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

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

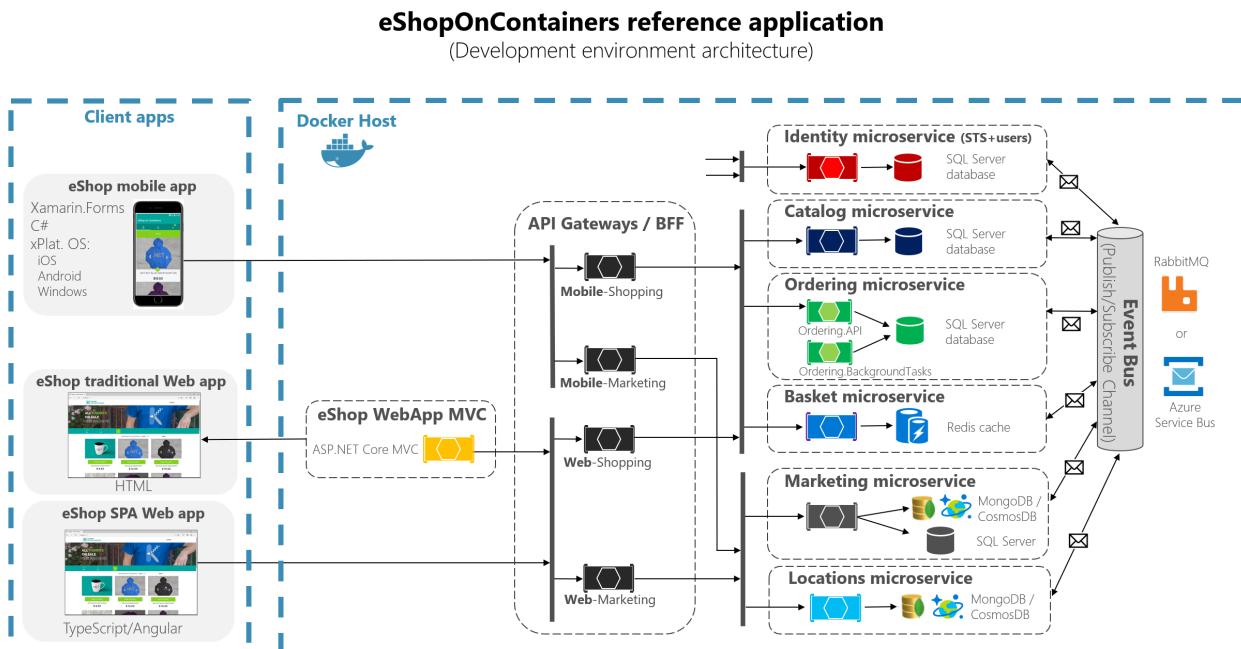


Figura 6-1. La arquitectura de aplicación de referencia de eShopOnContainers para el entorno de desarrollo

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

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

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

- Comunicación de cliente a microservicio de HTTP a través de puertas de enlace de API. Se usa para las consultas y al aceptar los comandos transaccionales o de actualización desde las aplicaciones cliente. El enfoque que usa puertas de enlace de API se explica con detalle en secciones posteriores.
- Comunicación asincrónica basada en eventos. Se realiza a través de un bus de eventos para propagar las actualizaciones entre los microservicios o para la integración con aplicaciones externas. El bus de eventos se puede implementar con cualquier tecnología de infraestructura de agente de mensajería como RabbitMQ, o bien mediante Service Bus de nivel superior (nivel de abstracción) como Azure Service Bus, NServiceBus, MassTransit o Brighter.

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

Propiedad de los datos por microservicio

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

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

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

Recursos adicionales

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

Ventajas de una solución basada en microservicios

Una solución basada en microservicios como esta tiene muchas ventajas:

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

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

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

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

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

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

Desventajas de una solución basada en microservicios

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

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

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

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

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

de recursos normalmente es una ventaja para las grandes aplicaciones a largo plazo.

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

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

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

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

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

Diferencias entre patrones de arquitectura y diseño externos e internos

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

External architecture per application

Internal architecture per microservice

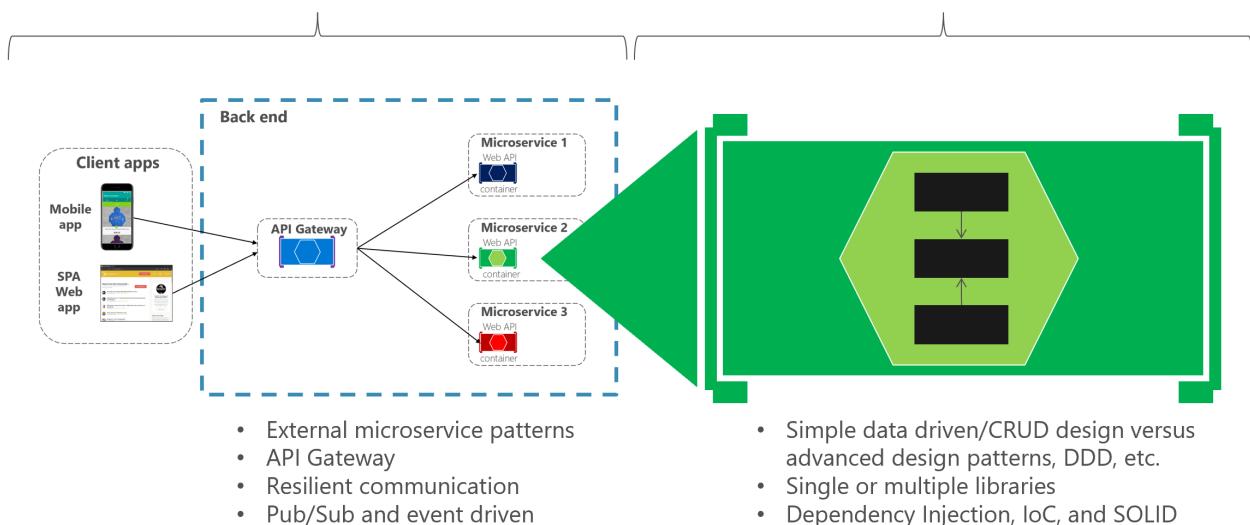


Figura 6-2. Diferencias entre arquitectura y diseño externos e internos

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

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

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

El nuevo mundo: varios modelos arquitectónicos y microservicios políglotas

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

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

- Arquitectura controlada por eventos (EDA).

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

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

The Multi-Architectural-Patterns and polyglot microservices world

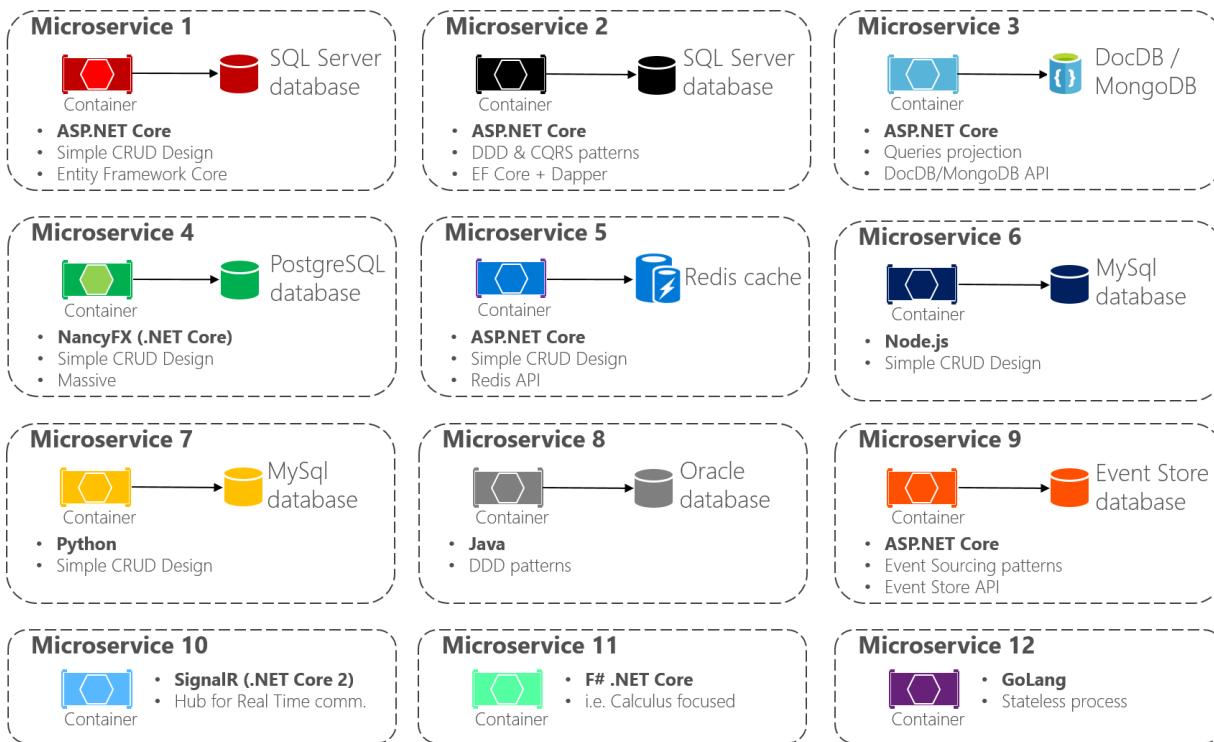


Figura 6-3. Modelos arquitectónicos múltiples y el mundo de los microservicios políglotas

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

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

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

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

[ANTERIOR](#)

[SIGUIENTE](#)

Creación de un microservicio CRUD sencillo controlado por datos

25/11/2019 • 33 minutes to read • [Edit Online](#)

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.

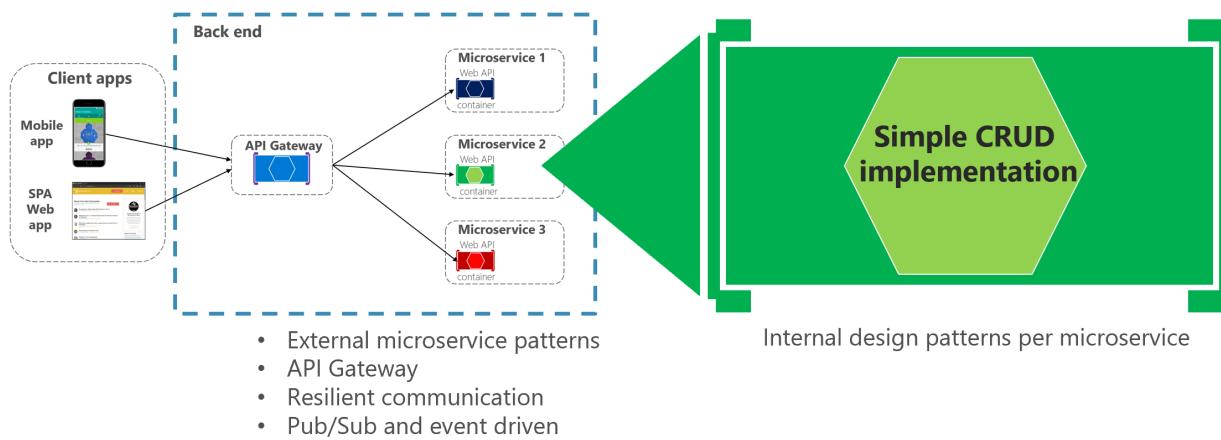


Figura 6-4. Diseño interno de microservicios CRUD sencillos

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.

Data-Driven/CRUD microservice container

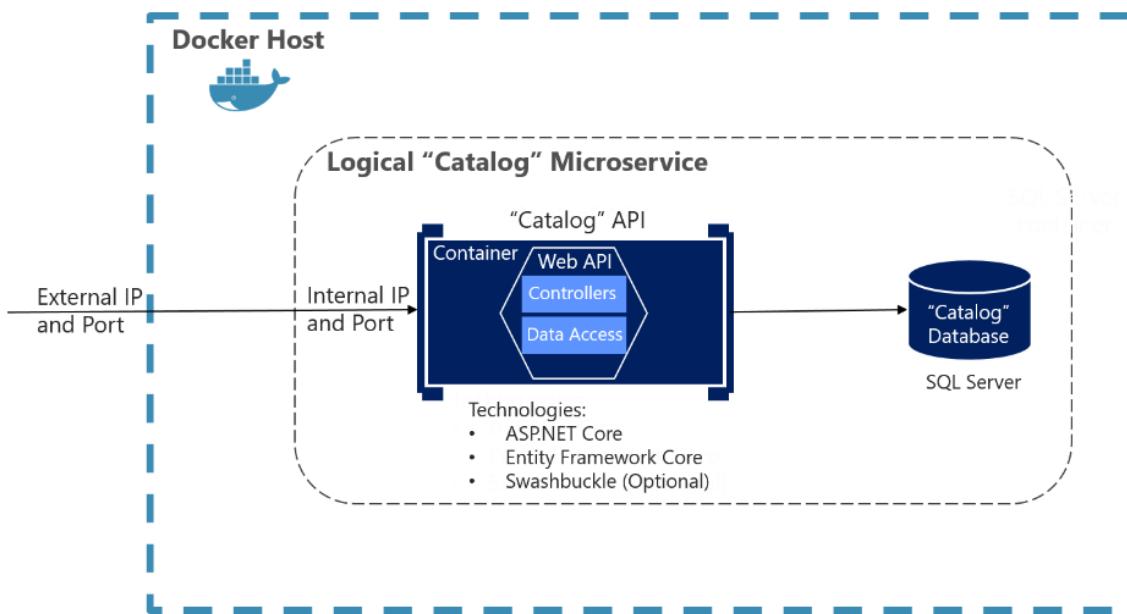


Figura 6-5. Diseño de un microservicio CRUD sencillo controlado por datos

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. Esto resulta muy útil para 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 Core y Visual Studio, primero debe crear un proyecto de API web de ASP.NET Core sencillo (que se ejecute en .NET Core para que pueda ejecutarse en un host de Linux Docker), como se muestra en la Figura 6-6.

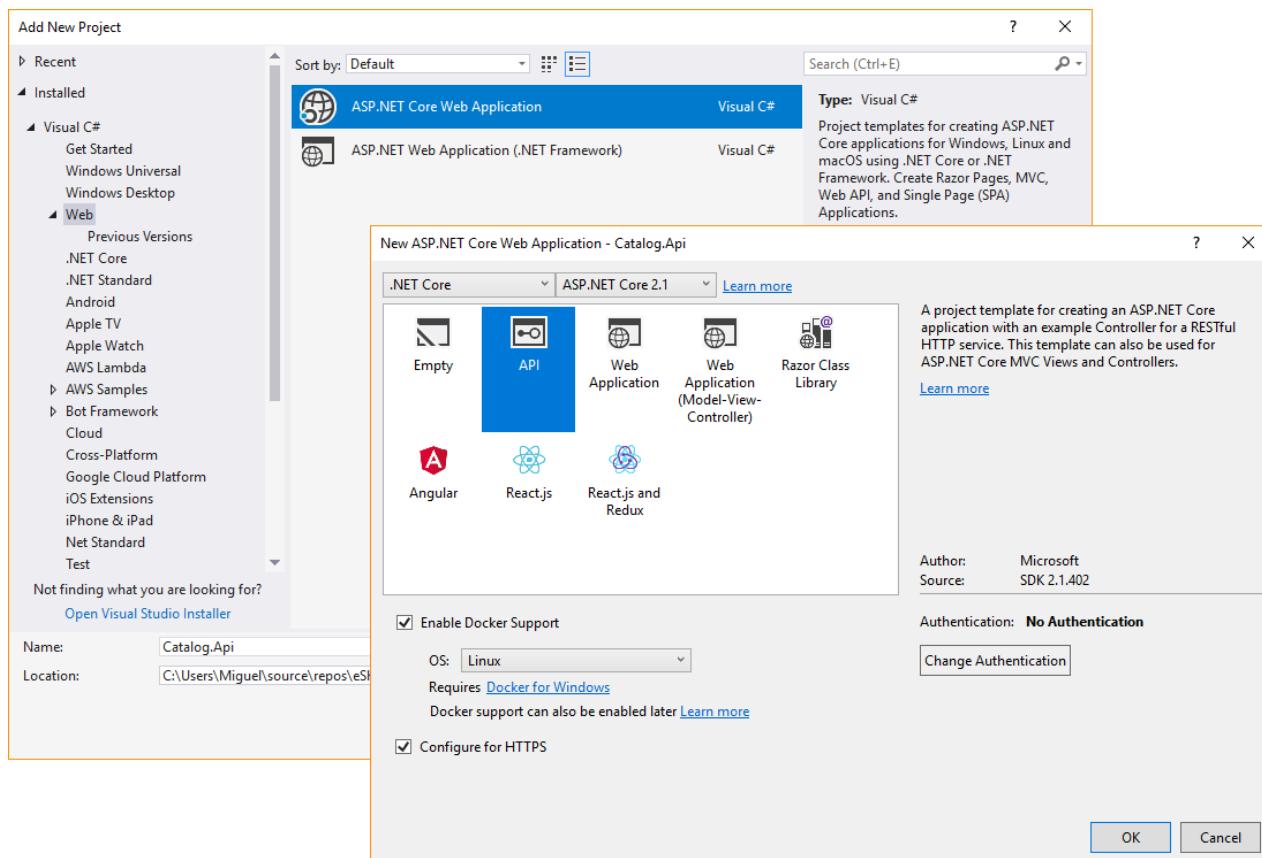


Figura 6-6. Creación de un proyecto de API Web de ASP.NET Core en Visual Studio

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 Core, como se muestra en la Figura 6-7.

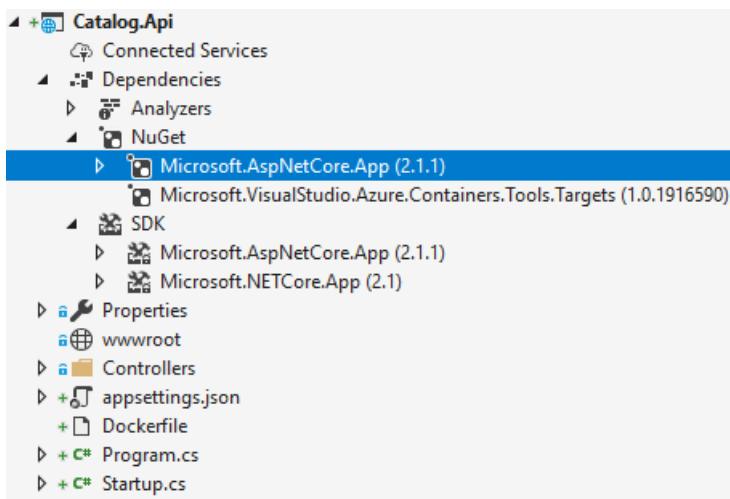


Figura 6-7. Dependencias en un microservicio API Web de CRUD sencillo

El proyecto de API incluye referencias al paquete NuGet *Microsoft.AspNetCore.App*, que 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 migraciones 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, usamos el último enfoque. Puede ver un ejemplo de la clase de entidad `CatalogItem` en el ejemplo de código siguiente, que es una clase de entidad de objeto CLR estándar ([POCO](#)).

```
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:

```
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](#).

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:

```
[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("[action]")]
    [ProducesResponseType(typeof(PaginatedItemsViewModel<CatalogItem>), (int) HttpStatusCode.OK)]
    public async Task<IActionResult> Items([FromQuery] int pageSize = 10,
                                            [FromQuery] int pageIndex = 0)

    {
        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.

```
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 el ejemplo anterior de la clase `CatalogController`, vamos a insertar un objeto del tipo `CatalogContext` junto con otros objetos a través del 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 la clase `Startup`, mediante una llamada al método `services.AddDbContext<DbContext>()` dentro del método `ConfigureServices()`, como se muestra en el ejemplo siguiente:

```
public void ConfigureServices(IServiceCollection services)
{
    // Additional code...

    services.AddDbContext<CatalogContext>(options =>
    {
        options.UseSqlServer(Configuration["ConnectionString"],
            sqlServerOptionsAction: sqlOptions =>
        {
            sqlOptions.MigrationsAssembly(
                typeof(Startup).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));
    });
    //...
}
```

Recursos adicionales

- **Consulta de datos**

<https://docs.microsoft.com/ef/core/querying/index>

- **Guardado de datos**

<https://docs.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:

```
{
  "ConnectionString": "Server=tcp:127.0.0.1,5433;Initial
Catalog=Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=Pass@word",
  "ExternalCatalogBaseUrl": "http://localhost: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).

```
# docker-compose.override.yml

#
catalog.api:
  environment:
    - ConnectionString=Server=sql.data;Database=Microsoft.eShopOnContainers.Services.CatalogDb;User
Id=sa;Password=Pass@word
    # Additional environment variables for this service
  ports:
    - "5101:80"
```

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 desde el código mediante la configuración ["ConnectionString"], tal y como se muestra en el método ConfigureServices de 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 muy detallado del uso de secretos de la aplicación sin necesidad de dejar que nadie los conozca. 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 expidite en el URI (v1 en este caso).

```
[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

- **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://docs.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api>
- **Roy Fielding. Versioning, Hypermedia, and REST** (Control de versiones, hipermedios 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 (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 análisis 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 muy 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.

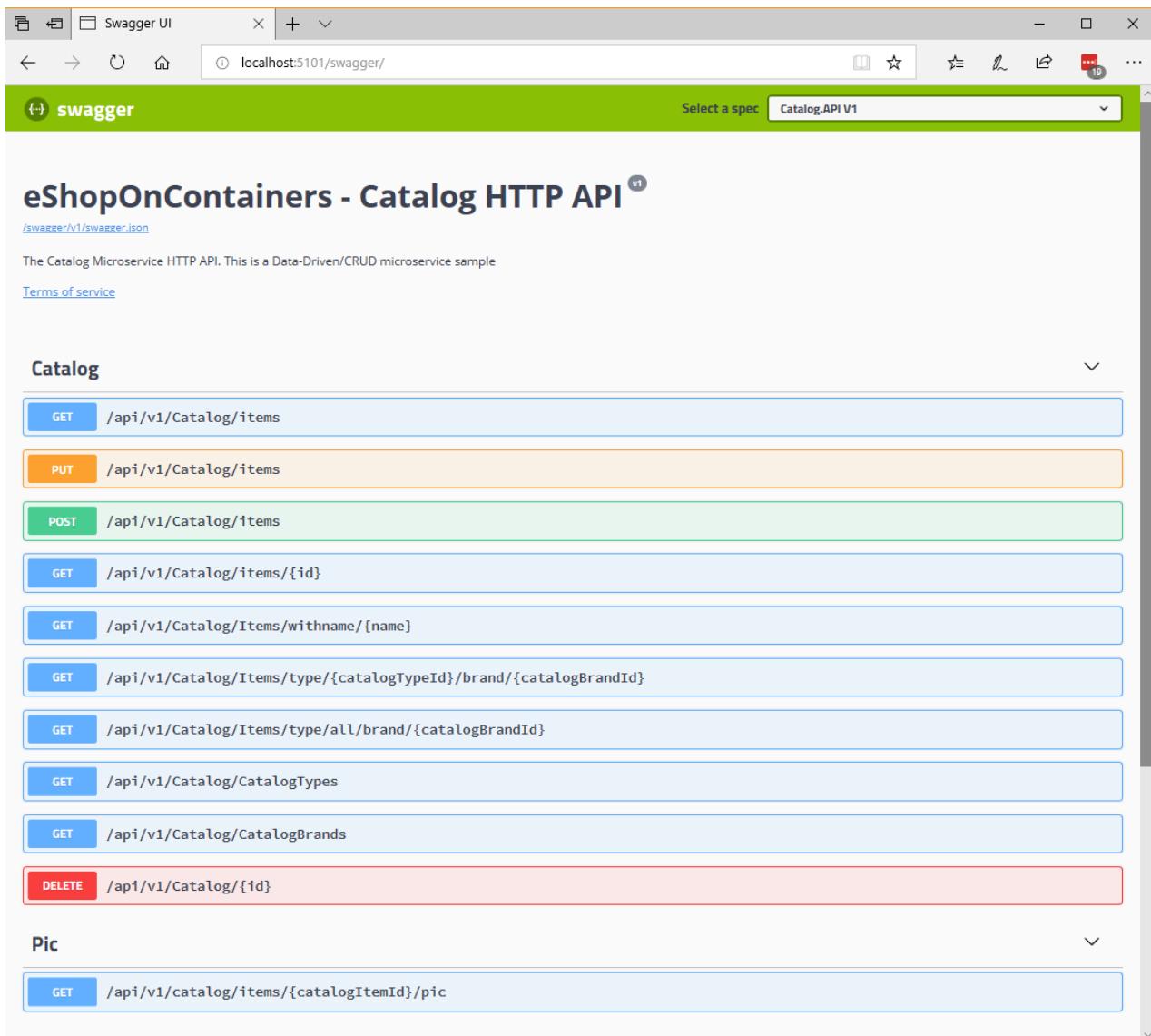


Figura 6-8. Explorador de API de Swashbuckle basado en metadatos de Swagger: microservicio del catálogo eShopOnContainers

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 metapaqete general [Swashbuckle.AspNetCore](#) para las aplicaciones ASP.NET Core.

Después de instalar estos paquetes NuGet en el proyecto de API web, debe configurar Swagger en la clase de inicio, como en el código siguiente (simplificado):

```
public class Startup
{
    public IConfigurationRoot Configuration { get; }
    // Other startup code...

    public void ConfigureServices(IServiceCollection services)
    {
        // Other ConfigureServices() code...

        // Add framework services.
        services.AddSwaggerGen(options =>
        {
            options.DescribeAllEnumsAsStrings();
            options.SwaggerDoc("v1", new Swashbuckle.AspNetCore.Swagger.Info
            {
                Title = "eShopOnContainers - Catalog HTTP API",
                Version = "v1",
                Description = "The Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice sample",
                TermsOfService = "Terms Of Service"
            });
        });

        // Other ConfigureServices() code...
    }

    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env,
        ILoggerFactory loggerFactory)
    {
        // Other Configure() code...
        // ...
        app.UseSwagger()
            .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:

```
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.

The screenshot shows the Swagger UI interface for the Catalog API. The URL is `localhost:5101/swagger/`. The main section displays the `GET /api/v1/Catalog/items` endpoint. In the 'Parameters' table, three parameters are defined: `pageSize` (integer, query), `pageIndex` (integer, query), and `ids` (string, query). Their values are set to 12, 0, and `ids` respectively. Below the table are 'Execute' and 'Clear' buttons. The 'Responses' section includes a 'Curl' example and a 'Request URL' field containing `http://localhost:5101/api/v1/Catalog/items?pageSize=12&pageIndex=0`. The 'Server response' section shows a 200 status code with a JSON response body:

```

{
  "pageIndex": 0,
  "pageSize": 12,
  "count": 12,
  "data": [
    {
      "id": 2,
      "name": ".NET Black & White Mug",
      "description": ".NET Black & White Mug",
      "price": 100,
      "pictureFileName": "2.png",
      "pictureUrl": "http://localhost:5202/api/v1/c/catalog/items/2/pic/",
      "catalogTypeId": 1,
      "catalogType": null,
      "catalogBrandId": 2,
      "catalogBrand": null,
      "availableStock": 100
    }
  ]
}

```

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. En la Figura 6-10 se muestran los metadatos JSON de Swagger generados a partir del microservicio eShopOnContainers (que es lo que las herramientas usan en segundo plano) al solicitar `http://<your-root-url>/swagger/v1/swagger.json` mediante [Postman](#).

The screenshot shows the Postman application interface. On the left, there's a sidebar with tabs for 'Runner', 'Import', and 'Builder'. Below that is a history section showing several API requests made on November 8 and November 6. The main area is titled 'Builder' and shows a request to 'http://localhost:5101/swagger/v1/swagger.json'. The 'Authorization' tab is selected, showing 'No Auth'. The 'Body' tab is selected, displaying the raw JSON response. The response content is as follows:

```
1  {
2   "swagger": "2.0",
3   "info": {
4     "version": "v1",
5     "title": "eShopOnContainers - Catalog HTTP API",
6     "description": "The Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice sample",
7     "termsOfService": "Terms Of Service"
8   },
9   "basePath": "/",
10  "paths": {
11    "/api/v1/Catalog/Items": {
12      "get": {
13        "tags": [
14          "Catalog"
15        ],
16        "operationId": "ApiV1CatalogItemsGet",
17        "consumes": [],
18        "produces": [],
19        "parameters": [
20          {
21            "name": "pageSize",
22            "in": "modelbinding",
23            "required": false,
24            "type": "integer",
25            "format": "int32"
26          }

```

Figura 6-10. Metadatos JSON de Swagger

Es así de sencillo. Y, como se generan automáticamente, los metadatos de Swagger crecerán cuando agregue más funcionalidad a la API.

Recursos adicionales

- **Páginas de ayuda de ASP.NET Core Web API con Swagger**

<https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger>

- **Introducción a Swashbuckle y ASP.NET Core**

<https://docs.microsoft.com/aspnet/core/tutorials/getting-started-with-swashbuckle>

- **Introducción a NSwag y ASP.NET Core**

<https://docs.microsoft.com/aspnet/core/tutorials/getting-started-with-nswag>

[ANTERIOR](#)

[SIGUIENTE](#)

Definir una aplicación de varios contenedores con docker-compose.yml

25/11/2019 • 29 minutes to read • [Edit Online](#)

En esta guía, el archivo [docker-compose.yml](#) se ha introducido en la sección [Paso 4. Definir los servicios en docker-compose.yml al compilar una aplicación de Docker de varios contenedores](#). Pero hay otras formas de usar los archivos docker-compose que merece la pena examinar con más detalle.

Por ejemplo, puede describir explícitamente cómo quiere implementar la aplicación de varios contenedores en el archivo docker-compose.yml. Si quiere, también puede describir cómo va a compilar las imágenes de Docker personalizadas (las imágenes de Docker personalizadas también se pueden compilar con la CLI de Docker).

Básicamente define cada uno de los contenedores que quiere implementar, además de ciertas características para cada implementación de contenedor. Una vez que tenga un archivo de descripción de la implementación de varios contenedores, puede implementar la solución completa en una sola acción organizada por el comando de la CLI [docker-compose up](#) o bien puede implementarla de forma transparente en Visual Studio. En caso contrario, tendría que usar la CLI de Docker para implementar uno a uno los contenedores en varios pasos mediante el comando `docker run` desde la línea de comandos. Por lo tanto, cada servicio definido en el archivo docker-compose.yml debe especificar exactamente una imagen o compilación. El resto de las claves son opcionales y son análogas a sus equivalentes de la línea de comandos de `docker run`.

El siguiente código YAML es la definición de un archivo docker-compose.yml posiblemente global pero único para el ejemplo de eShopOnContainers. Este no es el archivo docker-compose real de eShopOnContainers, sino que es una versión simplificada y consolidada en un único archivo, lo cual no es la mejor manera de trabajar con archivos docker-compose, como se explicará más adelante.

```

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=sql.data;Initial Catalog=CatalogData;User Id=sa;Password=your@password
    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:
      - sql.data

  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=sql.data;Database=Services.OrderingDb;User Id=sa;Password=your@password
    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:
      - sql.data

  basket.api:
    image: eshop/basket.api
    environment:
      - ConnectionString=sql.data
    ports:
      - "5103:80"
    depends_on:
      - sql.data

  sql.data:
    environment:
      - SA_PASSWORD=your@password
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"

  basket.data:
    image: redis

```

La clave raíz de este archivo es "services" (servicios). En esa clave se definen los servicios que se quieren implementar y ejecutar al ejecutar el comando `docker-compose up`, o bien al implementarlos 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.

NOMBRE DEL SERVICIO	DESCRIPCIÓN
webmvc	Contenedor que incluye la aplicación ASP.NET Core MVC que consume los microservicios de C# del lado servidor
catalog.api	Contenedor que incluye el microservicio Catalog de la API web de ASP.NET Core
ordering.api	Contenedor que incluye el microservicio Ordering de la API web de ASP.NET Core
sql.data	Contenedor que ejecuta SQL Server para Linux, que contiene las bases de datos de microservicios
basket.api	Contenedor que incluye el microservicio Basket de la API web de ASP.NET Core
basket.data	Contenedor que ejecuta el servicio Redis Cache, con la base de datos Basket como caché de Redis

Contenedor de la API de servicio web simple

Si nos centramos en un único contenedor, el microservicio de contenedor catalog.api tiene una definición sencilla:

```

catalog.api:
  image: eshop/catalog.api
  environment:
    - ConnectionString=Server=sql.data;Initial Catalog=CatalogData;User Id=sa;Password=your@password
  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:
    - sql.data

```

Este servicio de contenedor tiene la siguiente configuración básica:

- Se basa en la imagen eshop/catalog.api personalizada. Por simplicidad, no hay ninguna configuración compilación: clave en el archivo. Esto significa que la imagen se debe haber compilado previamente (con docker build) o se debe haber descargado (con el comando docker pull) de cualquier registro de Docker.
- Define una variable de entorno denominada `ConnectionString` con la cadena de conexión para que la use Entity Framework para obtener acceso a la instancia de SQL Server que contiene el modelo de datos del catálogo. En este caso, el mismo contenedor de SQL Server contiene varias bases de datos. Por lo tanto, necesitará menos memoria en el equipo de desarrollo para Docker, aunque también podría implementar un contenedor de SQL Server para cada base de datos de microservicio.
- El nombre de SQL Server es `sql.data`, que es el mismo nombre que se usa para el contenedor que ejecuta la instancia de SQL Server para Linux. Esto resulta práctico: poder usar esta resolución de nombres (interna al host de Docker) resolverá la dirección de red, por lo que no necesita saber la dirección IP interna de los contenedores a los que tiene acceso desde otros contenedores.

Dado que la cadena de conexión se define mediante una variable de entorno, podría establecer esa variable mediante otro mecanismo y en otro momento. Por ejemplo, podría establecer una cadena de conexión diferente al efectuar una implementación en producción en los hosts finales, o haciéndolo desde sus canalizaciones de CI/CD

en Azure DevOps Services o en su sistema de DevOps preferido.

- Expone el puerto 80 para el acceso interno al servicio catalog.api dentro del host de Docker. Actualmente, el host es una máquina virtual de Linux porque se basa en una imagen de Docker para Linux, aunque podría configurar el contenedor para que se ejecute en una imagen de Windows.
- Reenvía el puerto 80 expuesto del contenedor al puerto 5101 del equipo host de Docker (la máquina virtual de Linux).
- Vincula el servicio web al servicio sql.data (la base de datos de instancias de SQL Server para Linux que se ejecuta en un contenedor). Al especificar esta dependencia, el contenedor catalog.api no se iniciará hasta que se haya iniciado el contenedor sql.data. Esto es importante porque catalog.api necesita primero que la base de datos de SQL Server esté en ejecución. Pero este tipo de dependencia de contenedor no es suficiente en muchos casos, dado que Docker efectúa comprobaciones únicamente en el nivel de contenedor. A veces es posible que el servicio (en este caso SQL Server) aún no esté listo, por lo que es aconsejable implementar la lógica de reintento con retroceso exponencial en los microservicios de su cliente. De este modo, si un contenedor de dependencia no está listo durante un período de tiempo breve, la aplicación seguirá siendo resistente.
- Está configurado para permitir el acceso a los servidores externos: el valor de configuración extra_hosts le permite obtener acceso a máquinas o servidores externos situados fuera del host de Docker (es decir, fuera de la máquina virtual de Linux predeterminada, que es un host de Docker de desarrollo), como una instancia local de SQL Server en su equipo de desarrollo.

También existen otras opciones más avanzadas de los archivos docker-compose.yml que se exponen en las siguientes secciones.

Usar archivos docker-compose para fijar como objetivo varios entornos

Los archivos docker-compose.yml son archivos de definición que se pueden usar en varias infraestructuras que comprendan ese formato. La herramienta más sencilla y directa es el comando docker-compose.

Por lo tanto, si usa el comando docker-compose, puede fijar como objetivo 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 docker-compose para crear ese entorno o usar Visual Studio, que usa docker-compose en segundo plano.

El archivo docker-compose.yml le permite configurar y documentar todas las dependencias de servicio de la aplicación (otros servicios, la 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 actúan como prácticos archivos de documentación sobre la composición de la aplicación de varios contenedores.

Entornos de prueba

Una parte importante de cualquier proceso de implementación continua (CD) o de integración continua (CI) son las pruebas unitarias y las pruebas de integración. Estas pruebas automatizadas requieren un entorno aislado, por lo que no se ven afectadas por los usuarios ni por ningún otro cambio efectuado 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:

```
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 efectuar una implementación en un motor de Docker remoto. Un caso típico consiste en efectuar una implementación en una única instancia de host de Docker (como una máquina virtual de producción o un servidor aprovisionado con [Docker Machine](#)).

Si usa cualquier otro orquestador (Azure Service Fabric, Kubernetes, etc.), es posible que tenga que agregar valores de configuración de instalación y metadatos como los de docker-compose.yml, pero con el formato necesario para el otro orquestador.

En cualquier caso, docker-compose es una herramienta y un formato de metadatos prácticos 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.

Usar varios archivos docker-compose para controlar distintos entornos

Al fijar como objetivo entornos diferentes, debe usar varios archivos compose. Así puede crear distintas variantes de configuración en función del entorno.

Invalidar el archivo base docker-compose

Podría usar un archivo docker-compose.yml como en los ejemplos simplificados que se muestran en las secciones anteriores, pero no se recomienda para la mayoría de las aplicaciones.

De forma predeterminada, Compose lee dos archivos, un archivo docker-compose.yml y un archivo docker-compose.override.yml opcional. Como se muestra en la figura 6-11, cuando se usa Visual Studio y se habilita la compatibilidad con Docker, Visual Studio también crea un archivo docker-compose.vs.debug.g.yml adicional para depurar la aplicación, como se puede ver en la carpeta obj\ Docker\ de la carpeta de la solución principal.

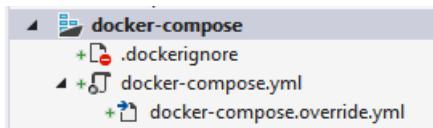


Figura 6-11. Archivos docker-compose en Visual Studio 2017

Estructura de los archivos de un proyecto de **docker-compose**:

- *.dockerignore*: se usa para omitir archivos.
- *docker-compose.yml*: se usa para crear 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 básica y otras opciones estáticas. Esto significa que la configuración del servicio no debería variar según el entorno de implementación que tenga como objetivo.

El archivo docker-compose.override.yml, como su nombre sugiere, contiene valores de configuración que invalidan la configuración básica, como la configuración que depende del entorno de implementación. También puede tener varios archivos de invalidación con nombres diferentes. Los archivos de invalidación 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 compose de manera que puede fijar como objetivo varios entornos (por ejemplo, producción, almacenamiento provisional, integración continua o desarrollo). Para dar cabida a estas diferencias, la configuración de Compose se puede dividir en varios archivos, como se muestra en la figura 6-12.

Multiple docker-compose files

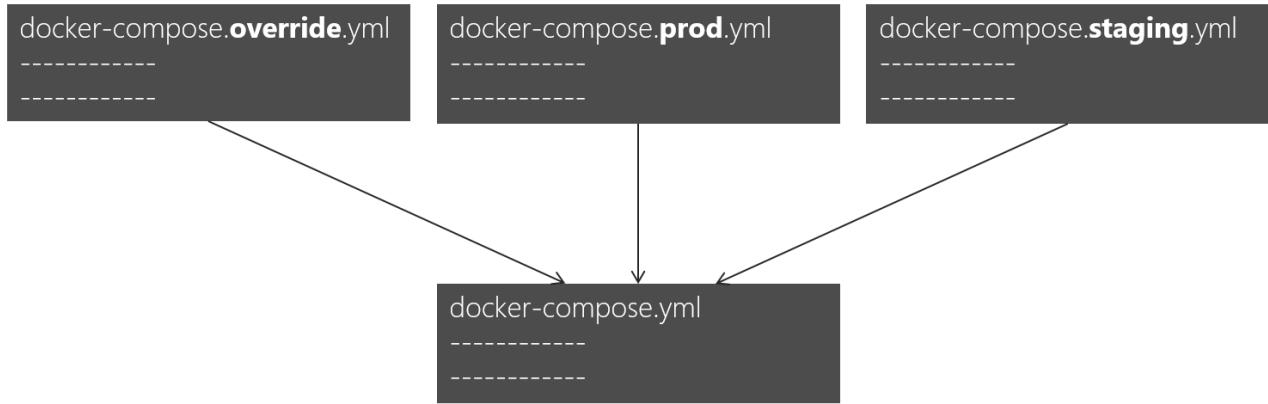


Figura 6-12. Varios archivos docker-compose invalidan los valores del archivo base docker-compose.yml

Se pueden combinar varios archivos docker-compose*.yml para controlar otros entornos. Comienza con el archivo base docker-compose.yml. Este archivo base debe contener los valores de configuración básicos o estáticos que no varían según el entorno. Por ejemplo, eShopOnContainers tiene el siguiente archivo docker-compose.yml (simplificado con menos servicios) como archivo base.

```

#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:
      - basket.data
      - identity.api
      - rabbitmq

  catalog.api:
    image: eshop/catalog.api:${TAG:-latest}
    build:
      context: .
      dockerfile: src/Services/Catalog/Catalog.API/Dockerfile
    depends_on:
      - sql.data
      - rabbitmq

  marketing.api:
    image: eshop/marketing.api:${TAG:-latest}
    build:
      context: .
      dockerfile: src/Services/Marketing/Marketing.API/Dockerfile
    depends_on:
      - sql.data
      - nosql.data
      - 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

  sql.data:
    image: microsoft/mssql-server-linux:2017-latest

  nosql.data:
    image: mongo

  basket.data:
    image: redis

  rabbitmq:
    image: rabbitmq:3-management

```

Los valores del archivo base docker-compose.yml no deberían variar porque haya distintos entornos de implementación de destino.

Si se centra en la definición del servicio webmvc, por ejemplo, puede ver que esa información es la misma con independencia del entorno que fije como objetivo. Dispone de la siguiente información:

- El nombre del servicio: webmvc.

- La imagen personalizada del contenedor: eshop/webmvc.
- El comando para compilar la imagen personalizada de Docker, que indica qué Dockerfile se debe usar.
- Dependencias de otros servicios, por lo que este contenedor no se inicia hasta que se hayan iniciado los otros contenedores de dependencia.

Puede tener otra configuración, pero lo importante es que en el archivo base docker-compose.yml solo establezca la información que es común en todos los entornos. Luego, en el archivo docker-compose.override.yml o en archivos similares de producción o almacenamiento provisional, debería colocar la configuración específica para cada entorno.

Por lo general, el archivo docker-compose.override.yml se usa para el entorno de desarrollo, como se muestra en el siguiente ejemplo de eShopOnContainers:

```
#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=${ESHOP_AZURE_REDIS_BASKET_DB:-basket.data}
- identityUrl=http://identity.api
- IdentityUrlExternal=http://${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=${ESHOP_AZURE_CATALOG_DB:-}
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=Pass@word
- PicBaseUrl=${ESHOP_AZURE_STORAGE_CATALOG_URL:-http://localhost: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=${ESHOP_AZURE_MARKETING_DB:-}
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.MarketingDb;User Id=sa;Password=Pass@word
- MongoConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosql.data}
```

```

- MongoDB=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://localhost: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"

sql.data:
environment:
- SA_PASSWORD=Pass@word
- ACCEPT_EULA=Y
ports:
- "5433:1433"

nosql.data:
ports:
- "27017:27017"

basket.data:
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.

Al ejecutar `docker-compose up` (o al iniciarla en Visual Studio), el comando lee las invalidaciones automáticamente como si se combinaran ambos archivos.

Imagínese que quiere que otro archivo Compose para el entorno de producción, con distintos valores de configuración, puertos o cadenas de conexión. Puede crear otro archivo de invalidación, como el archivo llamado

`docker-compose.prod.yml`, con distintas configuraciones y variables de entorno. Ese archivo podría estar almacenado en otro repositorio de Git o lo podría administrar y proteger 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 otro nombre, puede usar la opción `-f` con el comando docker-compose y especificar los archivos. Cree los archivos de combinaciones 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.

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

Usar variables de entorno en los archivos docker-compose

Resulta práctico, sobre todo en los entornos de producción, poder obtener información de configuración de variables de entorno, como hemos mostrado en ejemplos anteriores. En los archivos docker-compose se puede hacer referencia a una variable de entorno mediante la sintaxis `${MY_VAR}`. En la siguiente línea de un archivo `docker-compose.prod.yml` se muestra cómo hacer referencia al valor de una variable de entorno.

```
IdentityUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5105
```

Las variables de entorno se crean y se inicializan de maneras diferentes, en función de su entorno de host (Linux, Windows, clúster en la nube, etc.), aunque un método práctico consiste en usar un archivo `.env`. Los archivos docker-compose admiten la declaración de variables de entorno predeterminadas en el archivo `.env`. Estos valores de las variables de entorno son los valores predeterminados, pero se pueden invalidar con los valores que haya podido definir en cada uno de sus entornos (sistema operativo host o variables de entorno del clúster). Este archivo `.env` se coloca en la carpeta en la que se ejecuta el comando docker-compose.

En el siguiente ejemplo se muestra un archivo `.env` como el archivo [.env](#) para la aplicación eShopOnContainers.

```
# .env file  
ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost  
ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=10.121.122.92
```

Docker-compose espera que cada línea de los archivos `.env` tenga el formato `<variable>=<valor>`.

Tenga en cuenta que los valores establecidos en el entorno en tiempo de ejecución siempre invalidan los valores definidos en el archivo `.env`. De forma similar, los valores que se pasan a través de argumentos de comando de la 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/extends/#multiple-compose-files>

Compilación de imágenes optimizadas de Docker de ASP.NET Core

Si está explorando Docker y .NET Core en orígenes de Internet, encontrará Dockerfiles que muestran lo fácil que es compilar una imagen de Docker copiando el origen en un contenedor. Estos ejemplos sugieren que, si usa una configuración simple, puede tener una imagen de Docker con el entorno empaquetado con la aplicación. En el ejemplo siguiente se muestra un Dockerfile sencillo en esta misma línea.

```
FROM mcr.microsoft.com/dotnet/core/sdk:2.2
WORKDIR /app
ENV ASPNETCORE_URLS http://+:80
EXPOSE 80
COPY . .
RUN dotnet restore
ENTRYPOINT ["dotnet", "run"]
```

Un Dockerfile como este funcionará, pero puede optimizar considerablemente sus imágenes, sobre todo las imágenes de producción.

En el modelo de microservicios y contenedores están iniciando contenedores constantemente. El método habitual de usar los contenedores no reinicia un contenedor inactivo, porque el contenedor se puede descartar. Los orquestadores (como Kubernetes y Azure Service Fabric) tan solo crean instancias de imágenes. Esto significa que tendría que efectuar una optimización precompilando la aplicación al crearla para que el proceso de creación de instancias sea más rápido. Cuando se inicia el contenedor, tendría que estar preparado para ejecutarse. No se debería restaurar y compilar en tiempo de ejecución, con los comandos `dotnet restore` y `dotnet build` desde la CLI de dotnet, como se puede ver en muchas entradas de blog sobre .NET Core y Docker.

El equipo de .NET ha estado trabajando mucho para convertir .NET Core y ASP.NET Core en un marco optimizado para contenedores. .NET Core no solo es un marco ligero con una superficie de memoria pequeña; el equipo se ha centrado en imágenes de Docker optimizadas para los tres escenarios principales y las han publicado en el registro de Docker Hub en `dotnet/core`, empezando por la versión 2.1:

1. **Desarrollo:** La prioridad es la capacidad de iterar con rapidez y depurar cambios, donde el tamaño es secundario.
2. **Compilación:** La prioridad es compilar la aplicación e incluye los archivos binarios y otras dependencias para optimizar los archivos binarios.
3. **Producción:** El foco es la implementación y el inicio rápido de los contenedores, por lo que estas imágenes se limitan a los archivos binarios y el contenido necesario para ejecutar la aplicación.

Para lograrlo, el equipo de .NET proporciona tres variantes básicas en `dotnet/core` (en Docker Hub):

1. **sdk:** para los escenarios de desarrollo y compilación
2. **aspnet:** para los escenarios de producción de ASP.NET
3. **runtime:** para los escenarios de producción de .NET
4. **runtime-deps:** para los escenarios de producción de [aplicaciones autocontenidoas](#)

Para un inicio más rápido, las imágenes en tiempo de ejecución también configuran automáticamente las direcciones URL_aspnetcore en el puerto 80 y usan Ngen para crear una caché de imágenes nativa de ensamblados.

Recursos adicionales

- **Building Optimized Docker Images with ASP.NET Core** (Compilación de imágenes de Docker optimizadas con ASP.NET Core)
<https://blogs.msdn.microsoft.com/stevelasker/2016/09/29/building-optimized-docker-images-with-asp-net-core/>
- **Creación de imágenes de Docker para aplicaciones de .NET Core**
<https://docs.microsoft.com/dotnet/core/docker/building-net-docker-images>

Uso de un servidor de bases de datos que se ejecuta como un contenedor

02/12/2019 • 12 minutes to read • [Edit Online](#)

Puede tener las bases de datos (SQL Server, PostgreSQL, MySQL, etc.) en servidores independientes regulares, en clústeres locales o en los servicios PaaS en la nube como Azure SQL DB. Pero en los entornos de desarrollo y prueba, el hecho de que las bases de datos se ejecuten como contenedores es conveniente, ya que no tiene ninguna dependencia externa y solo con ejecutar el comando `docker-compose up` ya se inicia toda la aplicación. Tener esas bases de datos como contenedores también es muy útil para las pruebas de integración, porque 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 un microservicio

En eShopOnContainers, hay un contenedor denominado sql.data definido en el archivo `docker-compose.yml` que ejecuta SQL Server para Linux con todas las bases de datos de SQL Server necesarias para los microservicios. (También puede tener un contenedor de SQL Server para cada base de datos, pero eso requiere más memoria asignada a Docker). Lo importante de los microservicios es que cada microservicio posea sus datos relacionados, es decir, su base de datos SQL relacionada en este caso. Pero las bases de datos pueden encontrarse en cualquier lugar.

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 ha recopilado información de configuración 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).

```
sql.data:
  image: microsoft/mssql-server-linux:2017-latest
  environment:
    - SA_PASSWORD=Pass@word
    - ACCEPT_EULA=Y
  ports:
    - "5434:1433"
```

De forma similar, en lugar de usar `docker-compose`, el siguiente comando `docker run` puede ejecutar ese contenedor:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD=Pass@word' -p 5433:1433 -d microsoft/mssql-server-linux:2017-latest
```

Pero, si va a implementar una aplicación de varios contenedores, como eShopOnContainers, resulta más conveniente usar el comando `docker-compose up` para que implemente todos los contenedores necesarios para la aplicación.

Cuando se inicia este contenedor de SQL Server por primera vez, el contenedor inicializa SQL Server con la contraseña que usted 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 de SQL normal, como en SQL Server Management Studio, Visual Studio o código C#.

La aplicación eShopOnContainers inicializa cada base de datos de microservicio con datos de ejemplo que

propaga al inicio, tal como se describe en la sección siguiente.

El hecho de que SQL Server se ejecute como un contenedor no solo es útil para una demo, donde puede que no tenga acceso a una instancia de SQL Server, sino que también es perfecto para entornos de desarrollo y prueba, de manera que puede realizar fácilmente pruebas de integración a partir de una imagen limpia de SQL Server y datos conocidos propagando nuevos datos de ejemplo.

Recursos adicionales

- **Ejecución de imágenes de Docker de SQL Server en Linux, Mac o Windows**

<https://docs.microsoft.com/sql/linux/sql-server-linux-setup-docker>

- **Conexión y consulta de SQL Server en Linux con sqlcmd**

<https://docs.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 similar al siguiente al método Configure en la clase Startup del proyecto de API web:

```
public class Startup
{
    // Other Startup code...
    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env,
        ILoggerFactory loggerFactory)
    {
        // Other Configure code...
        // Seed data through our custom class
        CatalogContextSeed.SeedAsync(app)
            .Wait();
        // Other Configure code...
    }
}
```

El código siguiente en la clase CatalogContextSeed personalizada rellena los datos.

```

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 realizar pruebas de integración, resulta útil disponer de una forma de generar datos coherentes con las pruebas de integración. Poder crear cualquier cosa de cero, incluida una instancia de SQL Server que se ejecuta en un contenedor, es muy útil para los entornos de prueba.

Base de datos de EF Core InMemory frente a SQL Server que se ejecuta como un contenedor

Otra buena opción al realizar pruebas es usar el proveedor de base de datos de Entity Framework InMemory. Puede especificar esa configuración en el método ConfigureServices de la clase Startup en el proyecto de API web:

```

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 que son específicas de una base de datos determinada. Por ejemplo, podría agregar un índice único en una columna en el modelo de EF Core y escribir una prueba en la base de datos en memoria para comprobar que no le permite agregar un valor duplicado. Pero cuando usa la base de datos en memoria, no puede controlar los índices únicos en una columna. Por tanto, la base de datos en memoria no se comporta exactamente igual que una base de datos de SQL Server real: no emula restricciones específicas de la base de datos.

Aun así, una base de datos en memoria también es útil para las pruebas y la creación de prototipos. Pero si quiere crear pruebas de integración precisas que tengan en cuenta el comportamiento de la implementación de una base de datos determinada, debe usar una base de datos real como SQL Server. Para ello, ejecutar SQL Server en un contenedor es una gran opción, más precisa que el proveedor de base de datos de EF Core InMemory.

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 escenarios de prueba de concepto. Este escenario resulta práctico, porque puede hacer que todas las dependencias se ejecuten en contenedores, no solo para las máquinas de desarrollo locales, sino también para los entornos de pruebas en las canalizaciones 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 una PaaS (plataforma como servicio). En el código, solo debe 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 contenedor Redis de Docker ejecutando el siguiente comando de CLI de Docker en el símbolo del sistema:

```
docker run --name some-redis -d redis
```

La imagen de Redis incluye expose:6379 (el puerto que usa Redis), de manera que la vinculación de contenedor estándar hará que esté automáticamente disponible para los contenedores vinculados.

En eShopOnContainers, el microservicio basket.api usa una caché de Redis que se ejecuta como un contenedor. Ese contenedor basket.data se define como parte del archivo de varios contenedores docker-compose.yml, tal como se muestra en el ejemplo siguiente:

```
#docker-compose.yml file  
#...  
basket.data:  
  image: redis  
  expose:  
    - "6379"
```

Este código en el archivo docker-compose.yml define un contenedor denominado basket.data basado en la imagen de Redis que publica el puerto 6379 internamente, lo que significa que estará accesible solo desde los otros contenedores que se ejecutan dentro del host de Docker.

Por último, en el archivo docker-compose.override.yml, el microservicio basket.api para el ejemplo de eShopOnContainers define la cadena de conexión que se usará para ese contenedor de Redis:

```
basket.api:  
  environment:  
    # Other data ...  
    - ConnectionString=basket.data  
    - EventBusConnection=rabbitmq
```

Como se mencionó antes, el DNS de la red interna de Docker resuelve el nombre del microservicio "basket.data".

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación de comunicación basada en eventos entre microservicios (eventos de integración)

09/12/2019 • 15 minutes to read • [Edit Online](#)

Como se describió anteriormente, si utiliza una comunicación basada en eventos, un microservicio publica un evento cuando sucede algo importante, como cuando actualiza una entidad de negocio. Otros microservicios se suscriben a esos eventos. Cuando un microservicio recibe un evento, puede actualizar sus propias entidades de negocio, lo que puede comportar que se publiquen más eventos. Esta es la esencia del concepto de la coherencia final. Este sistema de publicación/suscripción normalmente se realiza mediante una implementación de un bus de eventos. El bus de eventos puede diseñarse como una interfaz con la API necesaria para suscribirse a eventos, cancelar las suscripciones y publicar eventos. También puede tener una o más implementaciones basadas en cualquier comunicación de mensajería o entre procesos, como una cola de mensajes o un bus de servicio que admita la comunicación asíncrona y un modelo de publicación/suscripción.

Puede usar eventos para implementar transacciones de negocio que abarquen varios servicios, lo cual proporciona una eventual coherencia entre dichos servicios. Una eventual transacción coherente consta de una serie de acciones distribuidas. En cada acción, el microservicio actualiza una entidad de negocio y publica un evento que desencadena la siguiente acción. En la figura 6-18 siguiente, se muestra un evento PriceUpdated publicado mediante un bus de eventos para que la actualización de los precios se propague a la cesta y a otros microservicios.

Implementing asynchronous event-driven communication with an event bus

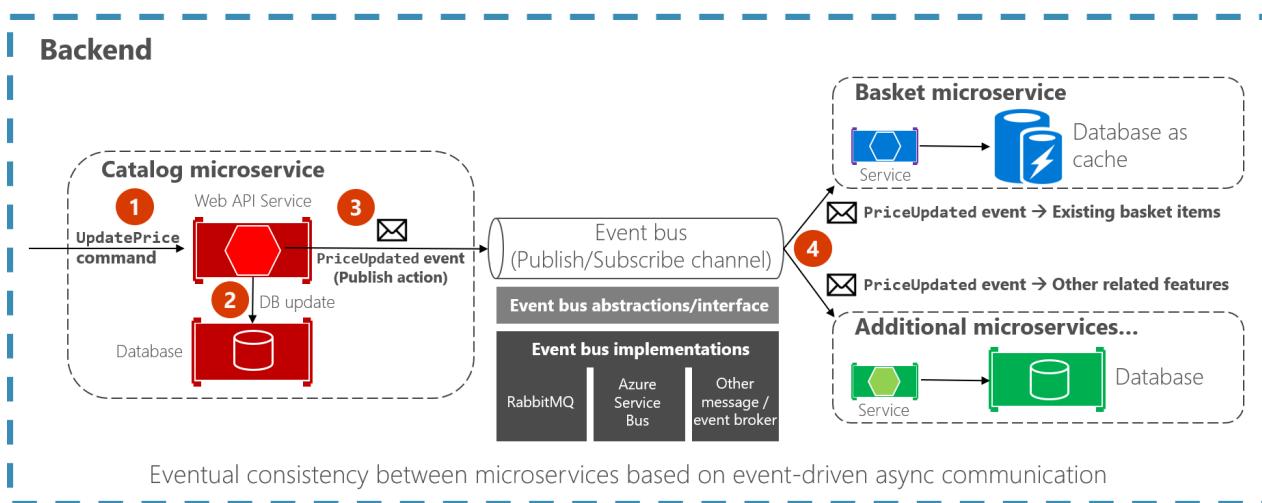


Figura 6-18. Comunicación orientada a eventos basada en un bus de eventos

En esta sección se describe cómo puede implementar este tipo de comunicación con .NET mediante un bus de eventos genéricos, como se muestra en la Figura 6-18. Hay varias implementaciones posibles, cada una de las cuales usa una tecnología o infraestructura distinta, como RabbitMQ, Azure Service Bus o cualquier otro bus de servicio de código abierto de terceros o comercial.

Uso de buses de agentes y servicios de mensajería para sistemas de producción

Como se indicó en la sección de arquitectura, puede escoger entre diferentes tecnologías de mensajería para

implementar el bus de eventos abstractos. Pero estas tecnologías se encuentran en distintos 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 puede trabajar encima de RabbitMQ o Azure Service Bus. La selección que haga del producto depende de la cantidad de características y de la escalabilidad de serie que necesite para la aplicación.

Para implementar solo una prueba de concepto de bus de eventos para su entorno de desarrollo, como en el ejemplo de eShopOnContainers, una implementación sencilla encima de RabbitMQ ejecutándose como contenedor podría ser suficiente. Pero para sistemas importantes y de producción que necesiten alta escalabilidad, se recomienda evaluar y usar Azure Service Bus.

Si necesita abstracciones de alto nivel y características más potentes, como [Sagas](#), para procesos de larga duración que faciliten el desarrollo distribuido, vale la pena tener en cuenta otros buses de servicio comerciales y de código abierto, como NServiceBus, MassTransit y Brighter. 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).

Obviamente, siempre puede crear sus propias características de bus de servicio sobre tecnologías de nivel inferior, como RabbitMQ y Docker, pero el trabajo necesario para "volver a inventar la rueda" puede ser demasiado costoso para una aplicación de empresa personalizada.

Para reiterar: las abstracciones de bus de eventos de ejemplo y la implementación presentada en el ejemplo de eShopOnContainers están diseñadas para usarse solo como una prueba de concepto. Una vez que haya decidido que quiere tener comunicación asíncrona y controlada por eventos, como se explica en la sección actual, debe elegir el producto de bus de servicio que mejor se adapte a sus necesidades de producción.

Eventos de integración

Los eventos de integración se utilizan para sincronizar el estado de dominio en varios microservicios o sistemas externos. Esto se realiza mediante la publicación de eventos de integración fuera del microservicio. Cuando se publica un evento en varios microservicios de receptor (en tantos microservicios como estén suscritos al evento de integración), el controlador de eventos correspondiente en cada microservicio de receptor controla el evento.

Un evento de integración es básicamente una clase de almacenamiento de datos, como en el ejemplo siguiente:

```
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 pueden definirse en el nivel de aplicación de cada microservicio, por lo que se separan de otros microservicios de una forma comparable a cómo se define ViewModels en el servidor y en el cliente. Lo que no se recomienda es compartir una biblioteca de eventos de integración común entre varios microservicios. De hacerlo, podría estar acoplando esos microservicios a una biblioteca de datos de definición de eventos únicos. Esto no le interesa por el mismo motivo que no le interesa compartir un modelo de dominio común entre varios

microservicios: los microservicios deben ser completamente autónomos.

Solo hay unos cuantos tipos de bibliotecas que debería compartir entre microservicios. Por un lado, las bibliotecas que son bloques de aplicaciones finales, como la [API de cliente de bus de eventos](#), como en eShopOnContainers. Por otro lado, las bibliotecas que constituyen herramientas que también se podrían compartir como componentes de NuGet, igual que los serializadores JSON.

El bus de eventos

Un bus de eventos permite una comunicación de estilo de suscripción/publicación entre microservicios, sin requerir que los componentes se reconozcan entre sí, como se muestra en la Figura 6-19.

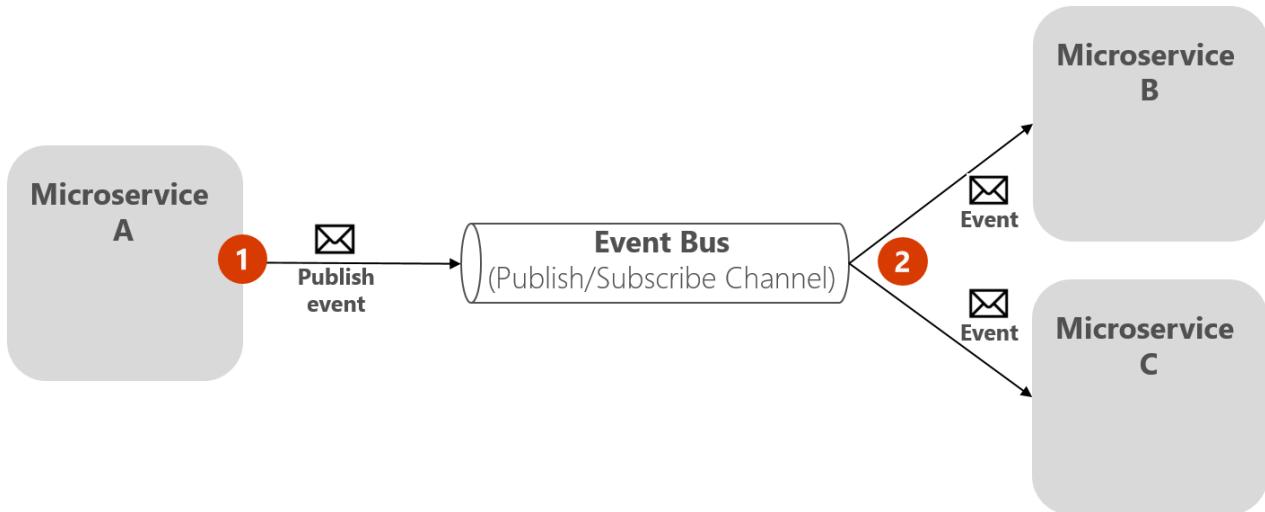


Figura 6-19. Aspectos básicos de publicación/suscripción con un bus de eventos

En el diagrama anterior se muestra que el microservicio A se publica en el bus de eventos, que lo distribuye a los microservicios B y C suscritos, sin que el editor tenga que conocer a los suscriptores. El bus de eventos está relacionado con el patrón de observador y con el patrón de publicación/suscripción.

Patrón de observador

En el [patrón de observador](#), su objeto principal (conocido como "Observable") proporciona información pertinente (eventos) a otros objetos interesados (conocidos como "Observadores").

Patrón de publicación/suscripción (Pub/Sus)

El propósito del [patrón de publicación/suscripción](#) es el mismo que el del modelo de observador: informar a otros servicios de la realización de determinados eventos. Pero hay una diferencia importante entre los patrones Observador y Pub/Sus. En el patrón de observador, la difusión se realiza directamente desde el objeto observable a los observadores, por lo que "se reconocen" entre sí. Pero cuando se usa un patrón Pub/Sus, hay un tercer componente, denominado "agente", "mensaje de agente" o "bus de eventos", que tanto el publicador como el suscriptor conocen. Por lo tanto, al utilizar el patrón Pub/Sus, el publicador y los suscriptores se desvinculan precisamente gracias al bus de eventos o al mensaje de agente mencionados.

El intermediario o bus de eventos

¿Cómo se consigue el anonimato entre el publicador y el suscriptor? Una forma sencilla de hacerlo es permitir que un intermediario se ocupe de toda la comunicación. Un bus de eventos es un intermediario de este tipo.

Normalmente, los buses de eventos están compuestos de dos partes:

- La abstracción o interfaz.
- Una o varias implementaciones.

En la Figura 6-19 puede ver cómo, desde el punto de vista de la aplicación, el bus de eventos no es más que un

canal de Pub/Sus. La forma de implementar este tipo de comunicación asincrónica puede variar. Puede tener varias implementaciones para intercambiarlas dependiendo 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 de mensajería.

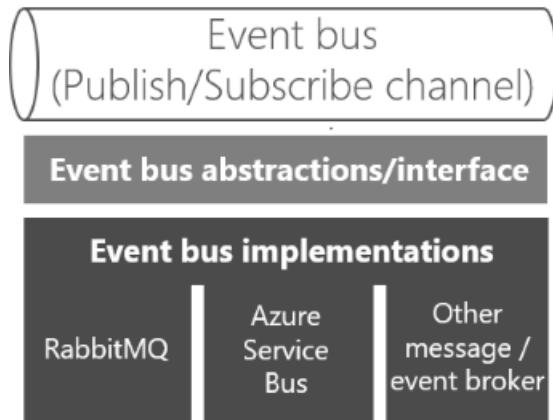


Figura 6-20. Varias implementaciones de un bus de eventos

Es conveniente definir el bus de eventos través de una interfaz, de forma que pueda implementarse con varias tecnologías, como RabbitMQ y Azure Service Bus entre otras. Pero, como se ha mencionado anteriormente, usar sus propias abstracciones (la interfaz del bus de eventos) solo es una buena opción si necesita características de bus de eventos compatibles con sus abstracciones. Si necesita características más completas del bus de servicio, probablemente tendrá que usar la API y las abstracciones proporcionadas por el bus de servicio comercial que prefiera, en vez de usar sus propias abstracciones.

Definición de una interfaz de bus de eventos

Comencemos con código de implementación para la interfaz de bus de eventos y las posibles implementaciones para fines de exploración. La interfaz debe ser sencilla y genérica, como la interfaz siguiente.

```
public interface IEventBus
{
    void Publish(IntegrationEvent @event);

    void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIIntegrationEventHandler<T>;

    void SubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void UnsubscribeDynamic<TH>(string eventName)
        where TH : IDynamicIntegrationEventHandler;

    void Unsubscribe<T, TH>()
        where TH : IIIntegrationEventHandler<T>
        where T : IntegrationEvent;
}
```

El método `Publish` es sencillo. El bus de eventos difunde el evento de integración que ha recibido a cualquier microservicio, o incluso a una aplicación externa, que se haya suscrito a ese evento. El microservicio que está publicando el evento utiliza este método.

Los microservicios que quieren recibir eventos utilizan los métodos `Subscribe` (puede tener varias implementaciones dependiendo de los argumentos). Este método tiene dos argumentos. El primero es el evento

de integración para suscribirse a (`IntegrationEvent`). El segundo es el controlador del evento de integración (o el método de devolución de llamada), denominado `IIntegrationEventHandler<T>`, que se ejecuta cuando el microservicio receptor recibe ese mensaje de evento de integración.

Recursos adicionales

Algunas soluciones de mensajería listas para producción:

- **Azure Service Bus**

<https://docs.microsoft.com/azure/service-bus-messaging/>

- **NServiceBus**

<https://particular.net/nservicebus>

- **MassTransit**

<https://masstransit-project.com/>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación de un bus de eventos con RabbitMQ para el entorno de desarrollo o de prueba

09/12/2019 • 7 minutes to read • [Edit Online](#)

Para empezar, hay que decir que si crea el bus de eventos personalizado basado en RabbitMQ que se ejecuta en un contenedor, como hace la aplicación eShopOnContainers, debe usarse únicamente para los entornos de desarrollo y prueba. No debe usarlo para el entorno de producción, salvo que lo cree como parte de un bus de servicio para entornos de producción. En un bus de eventos personalizado simple pueden faltar muchas de las características críticas para entornos de producción que tiene un bus de servicio comercial.

Una de las implementaciones personalizadas de bus de eventos en eShopOnContainers es básicamente una biblioteca que usa la API de RabbitMQ (hay otra implementación basada en Azure Service Bus).

La implementación del bus de eventos con RabbitMQ permite que los microservicios se suscriban a eventos, los publiquen y los reciban, tal como se muestra en la figura 6-21.

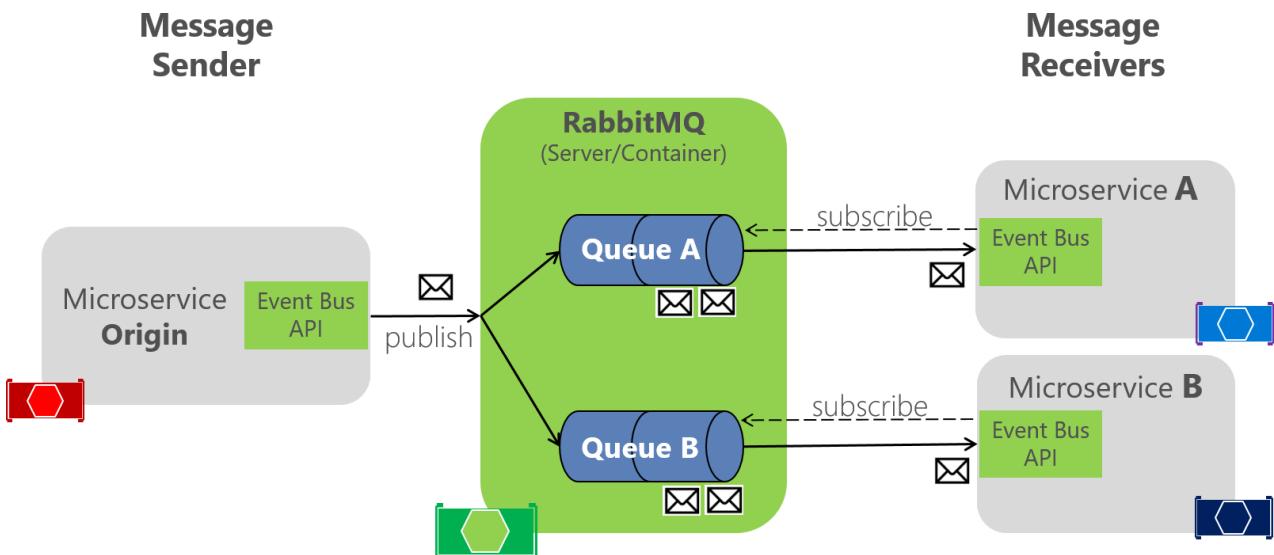


Figura 6-21. Implementación de RabbitMQ de un bus de eventos

Para controlar la distribución, RabbitMQ funciona como intermediario entre el publicador de mensajes y los suscriptores. En el código, la clase `EventBusRabbitMQ` implementa la interfaz genérica de `IEventBus`. Esto se basa en la inserción de dependencias para que pueda cambiar de esta versión de desarrollo/pruebas a una versión de producción.

```
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Implementation using RabbitMQ API
    //...
}
```

La implementación de RabbitMQ de un bus de eventos de desarrollo/pruebas de ejemplo es un código reutilizable. Tiene que controlar la conexión con el servidor RabbitMQ y proporcionar código para publicar un evento de mensaje a las colas. También debe implementar un diccionario de las colecciones de controladores de eventos de integración para cada tipo de evento; estos tipos de eventos pueden tener una instancia diferente y diferentes suscripciones para cada microservicio receptor, tal como se muestra en la figura 6-21.

Implementar un método de publicación sencillo con RabbitMQ

El código siguiente es una versión **simplificada** de una implementación de bus de eventos de RabbitMQ con el objetivo de presentar 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](#).

```
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 ha mejorado con una directiva de reintentos de [Polly](#), que vuelve a intentar la tarea un número determinado de veces en caso de que el contenedor RabbitMQ no esté listo. Esto puede ocurrir cuando docker-compose inicia los contenedores; por ejemplo, el contenedor de RabbitMQ puede iniciarse más lentamente que los otros contenedores.

Como se ha mencionado anteriormente, hay muchas configuraciones posibles en RabbitMQ, por lo que este código debe usarse únicamente para entornos de desarrollo y pruebas.

Implementar el código de suscripción con la API de RabbitMQ

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. Una vez más, normalmente no deberá cambiarlo a menos que lo esté mejorando.

```

public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods ...
    // ...

    public void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIIntegrationEventHandler<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. Puede tener tantos controladores de eventos por tipo de canal y evento como sea necesario.

El método `Subscribe` acepta un objeto `IIIntegrationEventHandler`, que es similar a un método de devolución de llamada en el microservicio actual, además de su objeto `IntegrationEvent` relacionado. Después, el código agrega ese controlador de eventos a la lista de controladores de eventos que puede tener cada tipo de evento de integración por microservicio cliente. Si el código cliente ya no se ha suscrito al evento, el código crea un canal para el tipo de evento de forma que pueda recibir eventos en un estilo de inserción de RabbitMQ cuando ese evento se publique desde cualquier otro servicio.

Como se mencionó anteriormente, el bus de eventos implementado en eShopOnContainers solo tiene fines educativos, ya que únicamente controla los escenarios principales y, por tanto, no está listo para la producción.

En escenarios de producción, compruebe los recursos adicionales siguientes, específicos para RabbitMQ, y la sección [Implementación de comunicación basada en eventos entre microservicios](#).

Recursos adicionales

Soluciones listas para la producción compatibles con RabbitMQ.

- **EasyNetQ**: cliente de la API de .NET de código abierto para RabbitMQ
<http://easynetq.com/>
- **MassTransit**
<https://masstransit-project.com/>

Suscripción a eventos

25/11/2019 • 34 minutes to read • [Edit Online](#)

El primer paso para usar el bus de eventos es suscribir los microservicios a los eventos que quieren recibir. Eso debe realizarse en los microservicios de receptor.

En el siguiente código simple se muestra lo que cada microservicio de receptor debe implementar al iniciar el servicio (es decir, en la clase `Startup`) para que se suscriba a los eventos que necesita. En este caso, el microservicio `basket.api` necesita suscribirse a los mensajes `ProductPriceChangedIntegrationEvent` y `OrderStartedIntegrationEvent`.

Por ejemplo, la suscripción al evento `ProductPriceChangedIntegrationEvent` hace que el microservicio de cesta sea consciente de los cambios en el precio del producto y le permite advertir al usuario sobre el cambio si ese producto está en la cesta de la compra del usuario.

```
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 algún mensaje de tipo `ProductPriceChangedIntegrationEvent`, el código invoca el controlador de eventos que se le pasa y procesa el evento.

Publicación de eventos a través del bus de eventos

Por último, el remitente del mensaje (el microservicio de origen) publica los eventos de integración con código similar al del ejemplo siguiente. (Es un ejemplo simplificado que no tiene en cuenta la atomicidad). Debería implementar un código similar cada vez que un evento se tenga que propagar entre varios microservicios, normalmente inmediatamente 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 en un Service Bus) se insertará en el constructor del controlador, como se muestra en el código siguiente:

```
[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;
    }
    // ...
}
```

Después, se usa desde los métodos del dispositivo, como en el método UpdateProduct:

```
[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, como 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 clase `CommandHandler`, dentro del método `Handle()`.

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 cambia el precio del producto y, después, publica un mensaje `ProductPriceChangedIntegrationEvent`. En principio, es posible que parezca fundamental que estas dos operaciones se realicen de forma atómica. Pero si está usando una transacción distribuida que implique la base de datos y el agente de mensajes, como se hace en sistemas anteriores como [Microsoft Message Queuing \(MSMQ\)](#), no se recomienda por las razones descritas por el [Teorema CAP](#).

Básicamente, los microservicios se usan para crear sistemas escalables y de alta disponibilidad. Para simplificarlo de algún modo, el teorema CAP afirma que no se puede crear una base de datos (distribuida), o un microservicio que posea su modelo, que esté continuamente disponible, tenga coherencia fuerte y sea tolerante a cualquier partición. Debe 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 tanto, en las aplicaciones basadas en microservicios más modernas, normalmente no le interesaría usar transacciones distribuidas en la mensajería, como haría al implementar [transacciones distribuidas](#) basadas en el Coordinador de transacciones distribuidas (DTC) de Windows con [MSMQ](#).

Volvamos al problema inicial y su ejemplo. Si el servicio se bloquea después de que se actualice 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 global puede volverse incoherente. Esto podría ser crítico para la empresa, según la operación empresarial específica con la que se esté tratando.

Como se mencionó anteriormente en la sección sobre arquitectura, puede tener varios enfoques para solucionar este problema:

- Uso del [patrón de orígenes de eventos](#) completo.
- Uso de la [minería del registro de transacciones](#).
- Uso del [patrón de bandeja de salida](#). Se trata de una tabla transaccional para almacenar los eventos de integración (extendiendo la transacción local).

En este escenario, el uso del modelo de orígenes de evento (ES) completo es uno de los mejores métodos, si no *el mejor*. Pero en muchas situaciones, es posible que no pueda implementar un sistema de ES completo. Con los orígenes de evento solo se almacenan los 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. Pero la implementación de un sistema de ES completo requiere que se cambie la arquitectura de la mayor parte del sistema y presenta otras muchas complejidades y requisitos. Por ejemplo, le interesaría usar una base de datos creada específicamente para los orígenes de eventos, como [Event Store](#), o bien 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 de usar la minería del registro de transacciones parece muy transparente en un principio. Pero para usar este enfoque, el microservicio debe acoplarse al registro de transacciones de RDBMS, como el registro de transacciones de SQL Server. Esto probablemente no sea deseable. Otra desventaja es que es posible que las actualizaciones de bajo nivel en el registro de transacciones no estén al mismo nivel que los eventos de integración generales. En ese caso, el proceso de utilización de técnicas de ingeniería inversa en esas operaciones de registro de transacciones puede ser complicado.

Un enfoque equilibrado es una combinación de una tabla de base de datos transaccional y un patrón de ES simplificado. Puede usar un estado como "listo para publicar el evento" que se establece en el evento original cuando se confirma en la tabla de eventos de integración. Después, intente publicar el evento en el bus de eventos. Si la acción de publicación de evento se realiza correctamente, inicie otra transacción en el servicio de origen y cambie el estado de "listo para publicar el evento" a "evento ya publicado".

Si se produce un error en la acción de publicación del evento en el bus de eventos, los datos todavía no serán incoherentes en el microservicio de origen (seguirán marcados como "listo 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 para cada microservicio de

origen y solo los eventos que le interesa comunicar con otros microservicios o sistemas externos. Por el contrario, en un sistema de ES completo, también se almacenan todos los eventos de dominio.

Por tanto, este enfoque equilibrado es un sistema de 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 tendrá que almacenar todos los datos de dominio como eventos en la base de datos transaccional, tal y como haría en un sistema de ES completo.

Si ya usa una base de datos relacional, puede usar una tabla transaccional para almacenar los eventos de integración. Para lograr la atomicidad en la aplicación, se usa un proceso de dos pasos basado en transacciones locales. Básicamente, dispone de una tabla `IntegrationEvent` en la misma base de datos donde se encuentren 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 el siguiente:

1. La aplicación inicia una transacción de base de datos local.
2. Después, actualiza el estado de las entidades de dominio e inserta un evento en la tabla de eventos de integración.
3. Finalmente, confirma la transacción, por lo que obtiene la atomicidad deseada.
4. A continuación, publique el evento de algún modo.

Al implementar los pasos necesarios para publicar los eventos, dispone de las opciones siguientes:

- Publicar el evento de integración justo después de confirmar la transacción y usar otra transacción local para marcar los eventos en la tabla como "en proceso de publicación". Después, usar la tabla como si fuera un artefacto para realizar el seguimiento de los eventos de integración en el caso de que se produzcan problemas en los microservicios remotos y realizar acciones de compensación en función de los eventos de integración almacenados.
- Usar la tabla como una especie de cola. Un proceso o subprocesso de aplicación independiente consulta la tabla de eventos de integración, publica los eventos en el bus de eventos y, después, 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.

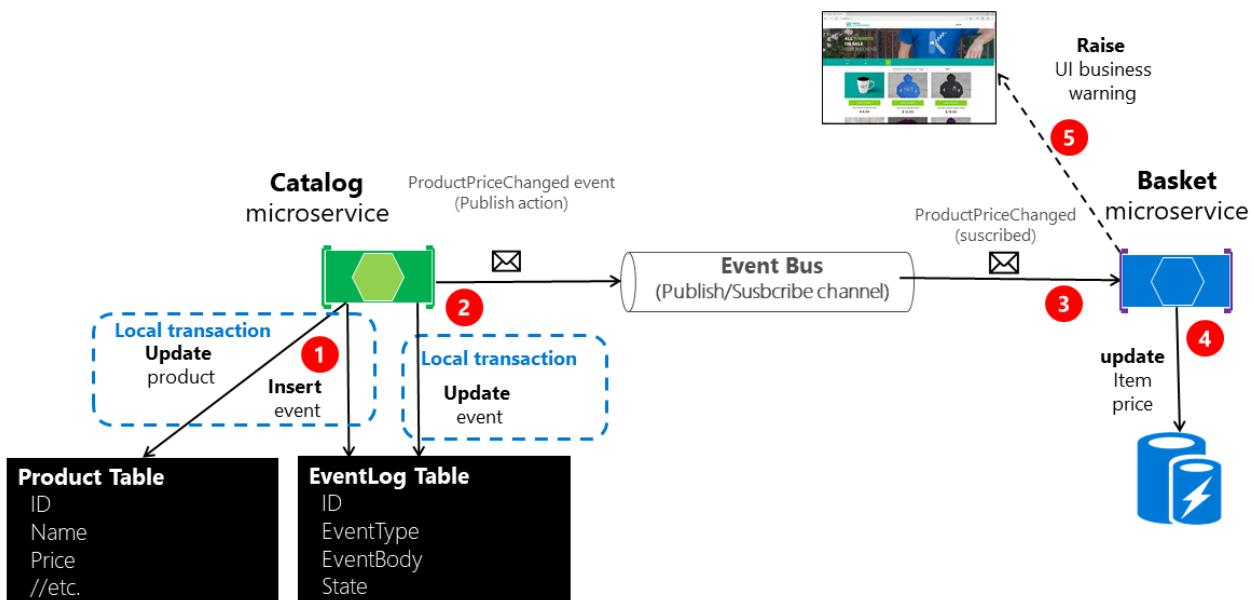


Figura 6-22. Atomicidad al publicar eventos en el bus de eventos

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.

Sobre el segundo enfoque: se usa la tabla EventLog como una cola y siempre se usa 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 cuando se publican los eventos.

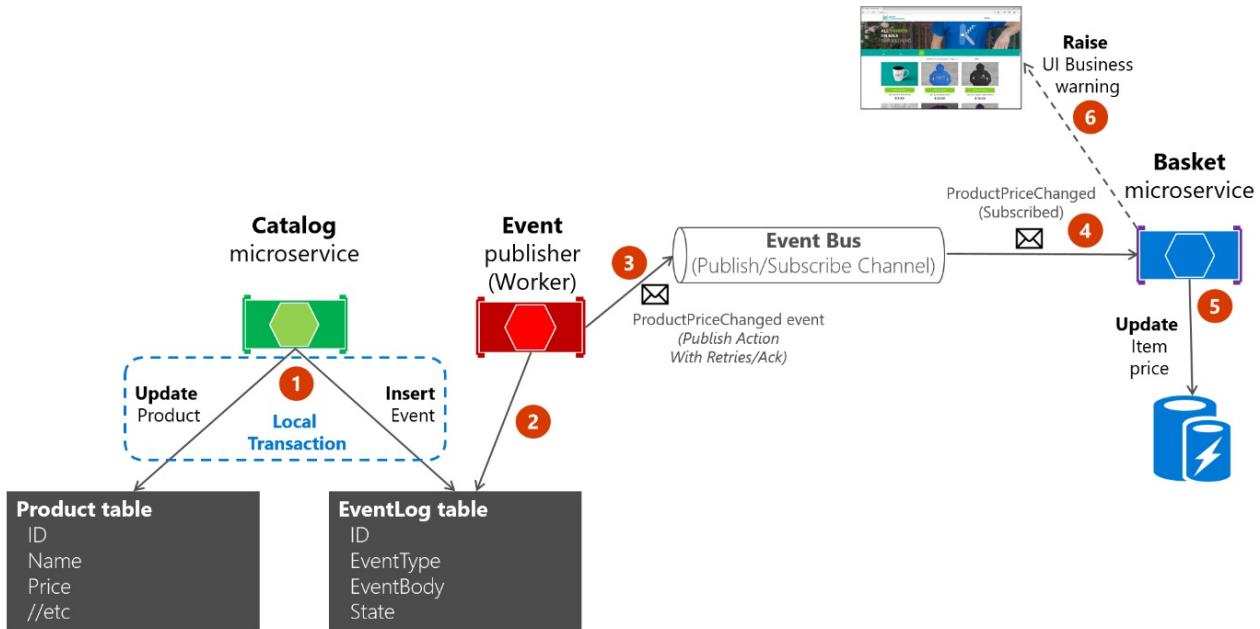


Figura 6-23. Atomicidad al publicar eventos en el bus de eventos con un microservicio de trabajo

Para simplificar, en el ejemplo eShopOnContainers se usa el primer enfoque (sin procesos adicionales ni microservicios de comprobador) junto con el bus de eventos. Pero en eShopOnContainers no se controlan todos los casos de error posibles. En una aplicación real implementada en la nube, debe asumir el hecho de que con el tiempo van a surgir problemas, 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 la forma de crear una única transacción que implica varios objetos `DbContext`, un contexto relacionado con los datos originales que se van a actualizar y el segundo relacionado con la tabla `IntegrationEventLog`.

Tenga en cuenta que la transacción en el código de ejemplo siguiente no será resistente si las conexiones a la base de datos tienen algún problema cuando se ejecute el código. Esto puede ocurrir en sistemas de servidor basados en la nube como Azure SQL DB, en los que es posible que las bases de datos se muevan entre servidores. Para implementar transacciones resistentes entre varios contextos, vea la sección [Implementación de conexiones resistentes de Entity Framework Core SQL](#) más adelante en esta guía.

Para evitar confusiones, en el ejemplo siguiente se muestra el proceso completo en un único fragmento de código. Pero la implementación de eShopOnContainers realmente se refactoriza y divide esta lógica en varias clases para que sea más fácil de mantener.

```

// 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();

            // Save to EventLog only if product price changed
            if(raiseProductPriceChangedEvent)
                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 (la actualización del elemento de catálogo) también incluye la persistencia del evento en la tabla `EventLog`. Esto crea una única transacción y siempre se podrá comprobar si los mensajes de eventos se han enviado.

La tabla de 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 cualquiera de las operaciones, se inicia una excepción y la transacción revierte cualquier operación completada, lo que mantiene la coherencia entre las

operaciones de dominio y los mensajes de eventos que se guardan en la tabla.

Recepción de mensajes desde suscripciones: controladores de eventos en microservicios de receptor

Además de la lógica de suscripción de eventos, debe implementar el código interno para los controladores de eventos de integración (por ejemplo, un método de devolución de llamada). En el controlador de eventos se especifica dónde se reciben y procesan los mensajes de eventos de un tipo determinado.

Un controlador de eventos recibe por primera vez una instancia de evento desde el bus de eventos. Después, busca el componente que se va a procesar relacionado con ese evento de integración y lo propaga y conserva como un cambio de estado en el microservicio de receptor. Por ejemplo, si un evento ProductPriceChanged se origina en el microservicio de catálogo, se controla en el microservicio de cesta de la compra y también cambia el estado en este microservicio de receptor, como se muestra en el código siguiente.

```
namespace Microsoft.eShopOnContainers.Services.Basket.API.IntegrationEvents.EventHandling
{
    public class ProductPriceChangedIntegrationEventHandler : 
        IIIntegrationEventHandler<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 la cesta de la compra. También actualiza el precio del artículo para cada artículo de línea de la cesta de la compra relacionado. Por último, crea una alerta que se mostrará al usuario sobre el cambio de precio, como se muestra en la figura 6-24.

The screenshot shows a shopping cart page from 'eSHOP onCONTAINERS' at localhost:5100/Cart. The cart contains three items:

PRODUCT	PRICE	QUANTITY	COST
	\$ 8.50	<input type="text" value="1"/>	\$ 8.50
	\$ 21.00	<input type="text" value="1"/>	\$ 21.00
	\$ 19.50	<input type="text" value="1"/>	\$ 19.50

A note in the middle of the cart states: "Note that the price of this article changed in our Catalog. The old price when you originally added it to the basket was 12.00 \$". The original price of \$21.00 is highlighted with a red box.

At the bottom right of the cart, there are two buttons: [UPDATE] and [CHECKOUT].

The footer of the page includes the eShoponContainers logo and the text "e-ShoponContainers. All right reserved".

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. Es necesario asegurarse de que las actualizaciones son idempotentes o que proporcionan información suficiente para garantizar que un duplicado se pueda detectar, descartarlo y devolver una sola respuesta.

Como se indicó anteriormente, idempotencia 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 bien al modo en que el sistema controla el evento. La idempotencia de mensajes es importante en cualquier aplicación en la que se use la mensajería, no solo en las aplicaciones que implementan el patrón de bus de eventos.

Un ejemplo de una operación idempotente es una instrucción SQL que inserta datos en una tabla solo si esos datos no están ya en la tabla. No importa cuántas veces se ejecute esa instrucción SQL de inserción; el resultado será el mismo: la tabla contendrá esos datos. Este tipo de idempotencia también puede ser necesaria cuando se trabaja con mensajes si existe la posibilidad de que se envíen y, por tanto, se procesen más de una vez. Por ejemplo, si la lógica de reintentos hace que un remitente envíe exactamente el mismo mensaje más de una vez,

tendrá que asegurarse de que sea idempotente.

Se pueden diseñar mensajes idempotentes. Por ejemplo, puede crear un evento que indique "establecer el precio del producto en 25 USD" en lugar de "sumar 5 USD al precio del producto". Podría procesar sin riesgos el primer mensaje cualquier número de veces y el resultado sería el mismo. Esto no es cierto para el segundo mensaje. Pero incluso en el primer caso, es posible que no le interese procesar el primer evento, porque el sistema también podría haber enviado un evento de cambio de precio más reciente y se podría sobrescribir el precio de nuevo.

Otro ejemplo podría ser un evento de pedido completado que se propaga a varios suscriptores. Es importante que la información del pedido se actualice una sola vez en otros sistemas, incluso si hay eventos de mensaje duplicados para el mismo evento de pedido completado.

Es conveniente tener algún tipo de identidad por evento para poder crear lógica que exija que cada evento se procese solo una vez por cada receptor.

Algún procesamiento de mensajes es idempotente de forma inherente. Por ejemplo, si un sistema genera imágenes en miniatura, es posible que no importe cuántas veces se procesa el mensaje sobre la miniatura generada; el resultado es que las miniaturas se generan y son iguales cada vez. Por otra parte, las operaciones como la llamada a una pasarela de pagos para cobrar una tarjeta de crédito no pueden ser idempotentes. En estos casos, debe asegurarse de que el procesamiento repetido de un mensaje tiene el efecto que se espera.

Recursos adicionales

- **Honoring message idempotency** (Respeto de la idempotencia de los mensajes)

[https://docs.microsoft.com/previous-versions/msp-n-p/jj591565\(v=pandp.10\)#honoring-message-idempotency](https://docs.microsoft.com/previous-versions/msp-n-p/jj591565(v=pandp.10)#honoring-message-idempotency)

Desduplicación de mensajes de eventos de integración

Puede asegurarse de que los eventos de mensajes se envían y se procesan una sola vez por cada suscriptor en niveles diferentes. Una manera de hacerlo consiste en usar una característica de desduplicación que ofrece la infraestructura de mensajería en uso. Otra consiste en implementar lógica personalizada en el microservicio de destino. Lo mejor es tener validaciones en el nivel de transporte y el nivel de aplicación.

Desduplicación de eventos de mensaje en el nivel de controlador de eventos

Una manera de asegurarse de que un evento se procesa solo una vez por cualquier receptor es mediante la implementación de cierta lógica al procesar los eventos de mensaje en controladores de eventos. Por ejemplo, ese es el enfoque que se usa en la aplicación eShopOnContainers, como se aprecia en el [código fuente de la clase UserCheckoutAcceptedIntegrationEventHandler](#) cuando recibe un evento de integración UserCheckoutAcceptedIntegrationEvent. (En este caso se encapsula CreateOrderCommand con un elemento IdentifiedCommand, usando eventMsg.RequestId como un identificador, antes de enviarlo al controlador de comandos).

Desduplicación de mensajes cuando se usa RabbitMQ

Cuando se producen errores de red intermitentes, los mensajes se pueden duplicar y el receptor del mensaje 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 "entregado de nuevo", el receptor debe tenerlo en cuenta, dado que es posible que el mensaje ya se haya procesado. Pero eso no está garantizado; es posible que el mensaje nunca llegara al receptor después de salir del agente de mensajes, quizás debido a problemas de red. Por otro lado, si no se estableció la marca "entregado de nuevo", se garantiza que el mensaje no se ha 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 [Particular Software]**
<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 de publicación y suscripción**
<https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- **Comunicación entre contextos delimitados**
[https://docs.microsoft.com/previous-versions/msp-n-p/jj591572\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-n-p/jj591572(v=pandp.10))
- **Coherencia de los eventos**
https://en.wikipedia.org/wiki/Eventual_consistency
- **Philip Brown. Estrategias para la integración de contextos delimitados**
<https://www.culttt.com/2014/11/26/strategies-integrating-bounded-contexts/>
- **Chris Richardson. 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 Richardson. Patrón de orígenes de eventos**
<https://microservices.io/patterns/data/event-sourcing.html>
- **Introducción a los orígenes de eventos**
[https://docs.microsoft.com/previous-versions/msp-n-p/jj591559\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-n-p/jj591559(v=pandp.10))
- **Base de datos Event Store.** Sitio oficial.
<https://geteventstore.com/>
- **Patrick Nommensen. Administración de datos orientada a eventos 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>
- **Manual de coherencia de datos**
[https://docs.microsoft.com/previous-versions/msp-n-p/dn589800\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-n-p/dn589800(v=pandp.10))
- **Rick Saling. Teorema CAP: por qué "todo es diferente" con la nube e Internet**
<https://blogs.msdn.microsoft.com/rickatmicrosoft/2013/01/03/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/>
- **Eric Brewer. ¿cómo han cambiado las "normas"? CAP doce años más tarde:**
<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>
- **Azure Service Bus. Mensajería asincrónica: Detección de duplicados**
<https://code.msdn.microsoft.com/Brokered-Messaging-c0acea25>

- **Guía de confiabilidad**[documentación de RabbitMQ]

<https://www.rabbitmq.com/reliability.html#consumer>

[ANTERIOR](#)

[SIGUIENTE](#)

Probar aplicaciones web y servicios ASP.NET Core

25/11/2019 • 13 minutes to read • [Edit Online](#)

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. Esto garantiza que los componentes individuales de la aplicación funcionan según lo previsto. Las aserciones prueban la API del componente.
- Pruebas de integración. Esto garantiza que las interacciones del componente 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. Esto garantiza que la aplicación funcione según lo esperado desde la perspectiva del usuario.
- Pruebas de servicio. Esto garantiza 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 up).

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 exemplificar 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.

```

[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 de 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. Esto se puede comparar a una solicitud HTTP, pero se ejecuta con mayor rapidez.

```

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. Controladores de pruebas** (ASP.NET Core)
<https://docs.microsoft.com/aspnet/core/mvc/controllers/testing>
- **Steve Smith. Pruebas de integración** (ASP.NET Core)
<https://docs.microsoft.com/aspnet/core/test/integration-tests>
- **Pruebas unitarias en .NET Core con dotnet test**
<https://docs.microsoft.com/dotnet/core/testing/unit-testing-with-dotnet-test>
- **xUnit.net.** Sitio oficial.
<https://xunit.github.io/>
- **Unit Test Basics** (Conceptos básicos de prueba unitaria).
<https://docs.microsoft.com/visualstudio/test/unit-test-basics>
- **Moq.** Repositorio de GitHub.
<https://github.com/moq/moq>
- **NUnit.** Sitio oficial.
<https://www.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 up (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**.
4. **Pruebas de carga**, que se centran en los tiempos de respuesta para cada microservicio. Estas pruebas se encuentran en el proyecto **LoadTest** y necesitan Visual Studio 2017 Enterprise Edition.

Las pruebas unitarias y de integración por microservicio se incluyen en una carpeta de prueba en cada microservicio y las pruebas de carga y aplicación se incluyen en la carpeta de pruebas de la carpeta de soluciones, como se muestra en la figura 6-25.

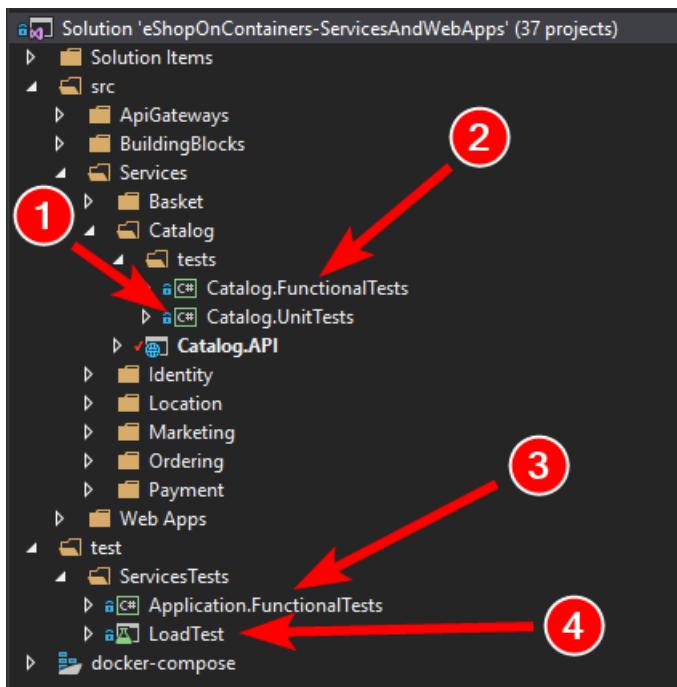


Figura 6-25. Estructura de carpetas de prueba en eShopOnContainers

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, por medio de un conjunto de archivos docker-compose incluidos en la carpeta de prueba de la solución:

docker-compose-test.yml

```
version: '3.4'

services:
  redis.data:
    image: redis:alpine
  rabbitmq:
    image: rabbitmq:3-management-alpine
  sql.data:
    image: microsoft/mssql-server-linux:2017-latest
  nosql.data:
    image: mongo
```

docker-compose-test.override.yml

```
version: '3.4'

services:
  redis.data:
    ports:
      - "6379:6379"
  rabbitmq:
    ports:
      - "15672:15672"
      - "5672:5672"
  sql.data:
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
  nosql.data:
    ports:
      - "27017:27017"
```

Por tanto, para ejecutar las pruebas de integración y funcionales primero debe ejecutar este comando, desde la carpeta de prueba de la solución:

```
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

- **Archivo LÉAME de las pruebas** en el repositorio de eShopOnContainers en GitHub
<https://github.com/dotnet-architecture/eShopOnContainers/tree/dev/test>
- **Archivo LÉAME de las pruebas de carga** en el repositorio de eShopOnContainers en GitHub
<https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/test/ServicesTests/LoadTest/>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementar tareas en segundo plano en microservicios con `IHostedService` y la clase `BackgroundService`

25/11/2019 • 16 minutes to read • [Edit Online](#)

Las tareas en segundo plano y los trabajos programados son algo que quizás tenga que implementar a la larga en una aplicación basada en microservicio o en cualquier tipo de aplicación. La diferencia al utilizar una arquitectura de microservicios es que permite implementar un proceso/contenedor único de microservicio para hospedar estas tareas en segundo plano, por lo que se puede escalar o reducir verticalmente según sea necesario, e incluso es posible asegurarse de que se ejecuta una sola instancia de ese proceso o contenedor de microservicio.

Desde un punto de vista genérico, en .NET Core este tipo de tareas se llaman *Servicios hospedados*, puesto que son servicios o lógica que se hospedan en el host, la aplicación o el microservicio. Observe que, en este caso, el servicio hospedado simplemente significa una clase con la lógica de la tarea de segundo plano.

Desde la versión 2.0 de .NET Core, el marco proporciona una nueva interfaz denominada `IHostedService` que le ayuda a implementar fácilmente servicios hospedados. La idea básica es que pueda registrar varias tareas en segundo plano (servicios hospedados), que se ejecutan en segundo plano mientras se ejecuta el host o host web, tal como se muestra en la imagen 6-26.

Implementing background tasks with `IHostedService` in .NET Core

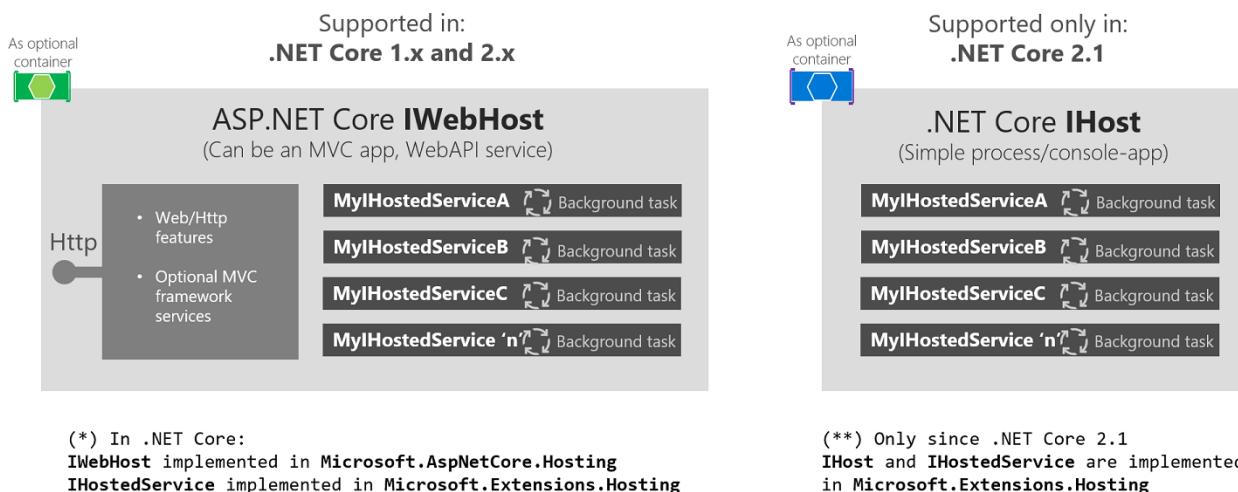


Figura 6-26. Uso de `IHostedService` en un `WebHost` frente a un `Host`

ASP.NET Core 1.x y 2.x admiten `IWebHost` para los procesos en segundo plano en aplicaciones web. .NET Core 2.1 admite `IHost` para los procesos en segundo plano con aplicaciones de consola planas. Observe la diferencia entre `WebHost` y `Host`.

`WebHost` (clase base que implementa `IWebHost`) en ASP.NET Core 2.0 es el artefacto de infraestructura que se utiliza para proporcionar características de servidor HTTP al proceso, por ejemplo, si se va a 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, insertar middleware en la canalización de solicitudes, etc., además de utilizar de manera precisa `IHostedServices` para tareas en segundo plano.

Un `Host` (clase base que implementa `IHost`) se presentó en .NET Core 2.1. Básicamente, `Host` permite disponer

de una infraestructura similar a la que se tiene con `WebHost` (inserción de dependencias, servicios hospedados, etc.), pero en este caso tan solo quiere tener un proceso sencillo y más ligero como host, sin ninguna relación con las características de servidor HTTP, MVC o API Web.

Por lo tanto, puede elegir y crear un proceso de host especializado con `IHost` para controlar los servicios hospedados y nada más, como por ejemplo un microservicio hecho solo para hospedar `IHostedServices`, o bien puede ampliar un `WebHost` de ASP.NET Core existente, como por ejemplo una aplicación MVC o API Web de ASP.NET Core.

Cada enfoque tiene ventajas e inconvenientes dependiendo de sus necesidades empresariales y de escalabilidad. La conclusión es básicamente que, si las tareas en segundo plano no tienen nada que ver con HTTP (`IWebHost`), debe usar `IHost`.

Registro de servicios hospedados en Host o WebHost

Vamos a profundizar más en la interfaz `IHostedService` puesto que su uso es muy 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:

- Una tarea en segundo plano que sondea una base de datos en busca de cambios.
- Una tarea programada que actualiza una caché periódicamente.
- Una implementación de `QueueBackgroundWorkItem` que permite que una tarea se ejecute 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 descargar cualquiera de esas acciones a una tarea en segundo plano basada en `IHostedService`.

La forma de agregar uno o varios elementos `IHostedServices` en `WebHost` o `Host` es registrarlos a través del método de extensión `AddHostedService` en un elemento `WebHost` de ASP.NET Core (o en un elemento `Host` en .NET Core 2.1 y versiones posteriores). Básicamente, tiene que registrar los servicios hospedados dentro del conocido método `ConfigureServices()` de la clase `Startup`, como se muestra en el código siguiente de un `WebHost` de ASP.NET típico.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    //Other DI registrations;

    // Register Hosted Services
    services.AddHostedService<GracePeriodManagerService>();
    services.AddHostedService<MyHostedServiceB>();
    services.AddHostedService<MyHostedServiceC>();
    //...
}
```

En el código, el servicio hospedado `GracePeriodManagerService` es código real del microservicio de negocios de pedido en `eShopOnContainers`, mientras que los otros dos son solo dos ejemplos adicionales.

La ejecución de la tarea en segundo plano `IHostedService` se coordina con la duración de la aplicación (host o microservicio para este propósito). Las tareas se registran cuando se inicia la aplicación y, cuando se esté cerrando la aplicación, tendrá la oportunidad de limpiar o realizar alguna acción correcta.

Si no se usa `IHostedService`, siempre se puede iniciar un subproceso en segundo plano para ejecutar cualquier tarea. La diferencia está precisamente en el tiempo de cierre de la aplicación, cuando ese subproceso simplemente terminaría sin tener la oportunidad de ejecutar las acciones de limpieza correcta.

Interfaz de `IHostedService`

Al registrar un `IHostedService`, .NET Core llamará a los métodos `StartAsync()` y `StopAsync()` de su tipo `IHostedService` durante el inicio y la detención de la aplicación respectivamente. En concreto, se llama a `start` después de que el servidor se inicie y se desencadene `IApplicationLifetime.ApplicationStarted`.

`IHostedService`, tal como se define en .NET Core, se parece a lo siguiente.

```
namespace Microsoft.Extensions.Hosting
{
    //
    // Summary:
    //     Defines methods for objects that are managed by the host.
    public interface IHostedService
    {
        //
        // Summary:
        // Triggered when the application host is ready to start the service.
        Task StartAsync(CancellationToken cancellationToken);
        //
        // Summary:
        // Triggered when the application host is performing a graceful shutdown.
        Task StopAsync(CancellationToken cancellationToken);
    }
}
```

Como puede imaginarse, es posible crear varias implementaciones de `IHostedService` y registrarlas en el método `ConfigureService()` en el contenedor de DI, tal y como se ha mostrado anteriormente. Todos los servicios hospedados se iniciarán y detendrán junto con la aplicación o microservicio.

Los desarrolladores son responsables de controlar la acción de detención o los servicios cuando el host activa el método `StopAsync()`.

Implementación de `IHostedService` con una clase de servicio hospedado personalizado que se deriva de la clase base `BackgroundService`

Puede seguir adelante y crear una clase de servicio hospedado personalizado desde cero e implementar `IHostedService`, tal y como se debe hacer cuando se usa .NET Core 2.0.

Pero como la mayoría de las tareas en segundo plano tienen necesidades similares en relación con la administración de tokens de cancelación y otras operaciones habituales, hay una clase base abstracta práctica denominada `BackgroundService` de la que puede derivar (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 de `BackgroundService` tal y como se implementa en .NET Core.

```

// 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, y gracias a la implementación heredada, solo tiene que implementar el método `ExecuteAsync()` en su clase de servicio hospedado personalizado propio, como en el siguiente ejemplo simplificado de código de eShopOnContainers, en el que se sondea una base de datos y se publican eventos de integración en el bus de eventos cuando es necesario.

```

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();

            await Task.Delay(_settings.CheckUpdateTime, stoppingToken);
        }

        _logger.LogDebug($"GracePeriod background task is stopping.");
    }

    .../...
}

```

En este caso concreto de eShopOnContainers, se ejecuta un método de aplicación que consulta una tabla de base de datos en la que busca pedidos con un estado específico y al aplicar los cambios, está publicando eventos de integración a través del bus de eventos (de forma subyacente puede estar utilizando RabbitMQ o Azure Service Bus).

Por supuesto, en su lugar puede ejecutar cualquier otra tarea en segundo plano empresarial.

De forma predeterminada, el token de cancelación se establece con un tiempo de espera de 5 segundos, aunque se puede cambiar ese valor al compilar su `WebHost` mediante la extensión `UseShutdownTimeout` de `IWebHostBuilder`. Esto significa que se espera que nuestro servicio se cancele en 5 segundos o, en caso contrario, se terminará de manera repentina.

El código siguiente cambiaría ese tiempo a 10 segundos.

```

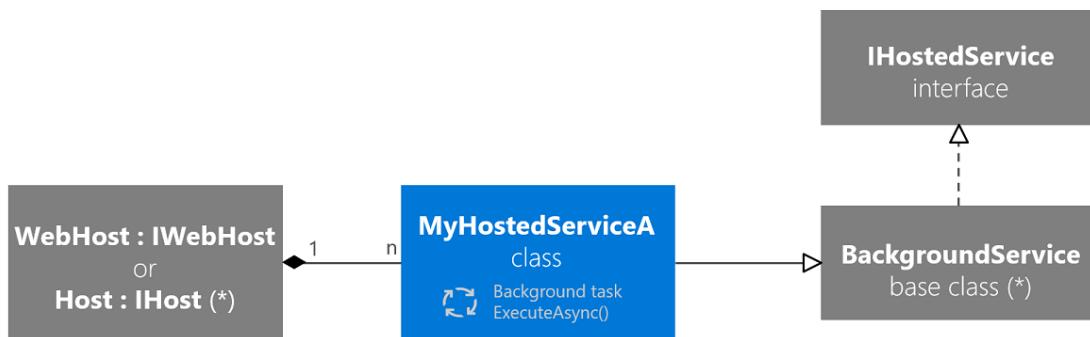
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...

```

Diagrama de clases de resumen

En la siguiente ilustración se muestra un resumen visual de las clases y las interfaces implicadas al implementar `IHostedServices`.

Class diagram with a custom `IHostedService` and related classes and interfaces



(**) `IHost` and `BackgroundService` are implemented in `Microsoft.Extensions.Hosting` only since .NET Core 2.1

Figura 6-27. Diagrama de clases que muestra las distintas clases e interfaces relacionadas con `IHostedService`

Diagrama de clases: `IWebHost` y `IHost` pueden hospedar muchos servicios, que heredan de `BackgroundService`, que implementa `IHostedService`.

Impresiones y consideraciones sobre implementación

Es importante tener en cuenta que la forma de implementar su ASP.NET Core `WebHost` o .NET Core `Host` puede 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 implementar el host como un contenedor en un orquestador como Kubernetes o Service Fabric, puede controlar el número garantizado de instancias activas del host. Además, podría considerar otros métodos en la nube pensados especialmente para estos escenarios, como Azure Functions. Por último, si necesita que el servicio se ejecute todo el tiempo y se implemente en 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 interfaz `IHostedService` proporciona una manera cómoda de iniciar tareas en segundo plano en una aplicación web de ASP.NET (en .NET Core 2.0) 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.stevegordon.co.uk/asp-net-core-2-ihostedservice>
- **Ejemplo de `GenericHost` mediante ASP.NET Core 2.1**
<https://github.com/aspnet/Hosting/tree/release/2.1/samples/GenericHostSample>

Implementación de puertas de enlace de API con Ocelot

25/11/2019 • 41 minutes to read • [Edit Online](#)

En la aplicación de microservicio de referencia [eShopOnContainers](#) se usa [Ocelot](#), una puerta de enlace de API simple y ligera que se puede implementar en cualquier lugar junto con los microservicios y contenedores, como en cualquiera de los entornos siguientes que se usan en eShopOnContainers.

- Host de Docker, en el equipo PC de desarrollo local, en el entorno local o en la nube.
- Clúster de Kubernetes, de forma local o en la nube administrada como Azure Kubernetes Service (AKS).
- Clúster de Service Fabric local o en la nube.
- Malla de Service Fabric, como PaaS o sin servidor en Azure.

Arquitectura y diseño de las puertas de enlace de API

En el diagrama de arquitectura siguiente se muestra cómo se implementan las puertas de enlace de API con Ocelot en eShopOnContainers.

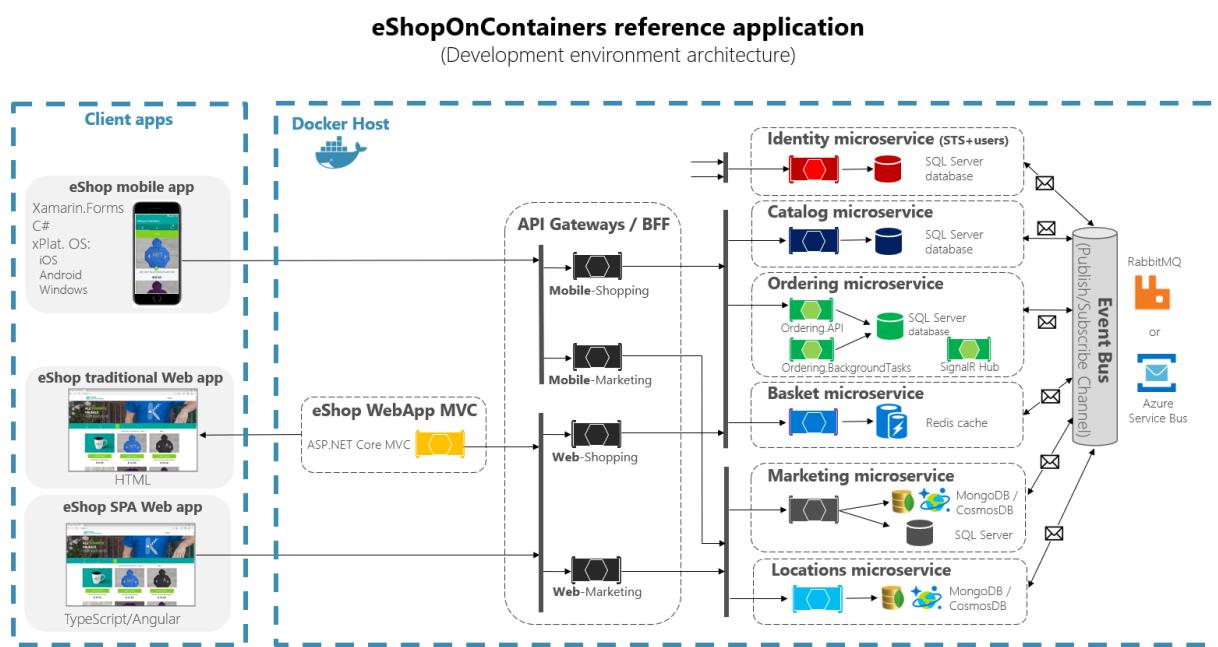


Figura 6-28. Arquitectura de eShopOnContainers con puertas de enlace de API

En ese diagrama se muestra cómo se implementa toda la aplicación en un único host de Docker o PC de desarrollo con "Docker para Windows" o "Docker para Mac". Pero la implementación en cualquier orquestador sería bastante similar y cualquiera de los contenedores del diagrama se podría escalar horizontalmente en el orquestador.

Además, los recursos de infraestructura como bases de datos, caché y agentes de mensajes se deberían descargar 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 se puede observar en el diagrama, tener varias puertas de enlace de API permite que varios equipos de desarrollo sean autónomos (en este caso, las características de Marketing frente a las de Shopping) al

desarrollar e implementar sus microservicios además de sus propias puertas de enlace de API relacionadas.

Si tuviera una sola puerta de enlace de API monolítica, sería un único punto para actualizar por varios equipos de desarrollo, que podrían acoplar todos los microservicios con un único elemento de la aplicación.

Ampliando mucho más el diseño, en ocasiones una puerta de enlace de API específica también se puede limitar a un microservicio empresarial individual en función de la arquitectura elegida. El tener los límites de la puerta de enlace de API dictados por el negocio o el dominio le ayudará a lograr un mejor diseño.

Por ejemplo, la especificidad del nivel de puerta de enlace de API puede ser especialmente útil para aplicaciones de interfaz de usuario compuesta más avanzadas que se basan en microservicios, dado que el concepto de una puerta de enlace de API específica es similar a un servicio de composición de interfaz de usuario.

Nos adentramos en más detalles en la sección anterior, [Creación de interfaz de usuario compuesta basada en microservicios](#).

Como punto clave, para muchas aplicaciones de tamaño medio y grande, el uso de una puerta de enlace de API personalizada suele ser un enfoque adecuado, pero no como un único agregador monolítico ni una única puerta de enlace de API personalizada central, a menos que esa puerta de enlace de API permita varias áreas de configuración independientes para que los diferentes equipos de desarrollo creen microservicios autónomos.

Microservicios y contenedores de ejemplo para redistribuir entre las puertas de enlace de API

Como ejemplo, eShopOnContainers tiene aproximadamente seis tipos de microservicio internos que se tienen que publicar entre las puertas de enlace de API, como se muestra en la imagen siguiente.

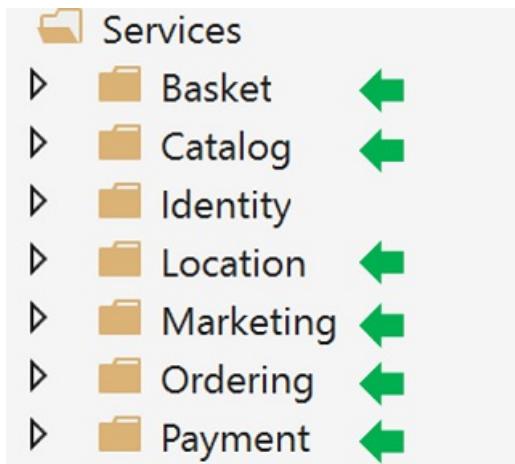


Figura 6-29. Carpetas de microservicio en la solución eShopOnContainers en Visual Studio

En cuanto al servicio Identity, en el diseño se excluye del enrutamiento de puerta de enlace de API, porque es el único interés transversal del sistema, aunque con Ocelot también es posible incluirlo como parte de las listas de reenrutamiento.

Todos estos servicios se implementan actualmente como servicios de API web de ASP.NET Core, como se desprende del código. Centrémonos en uno de los microservicios, como el código del microservicio Catalog.

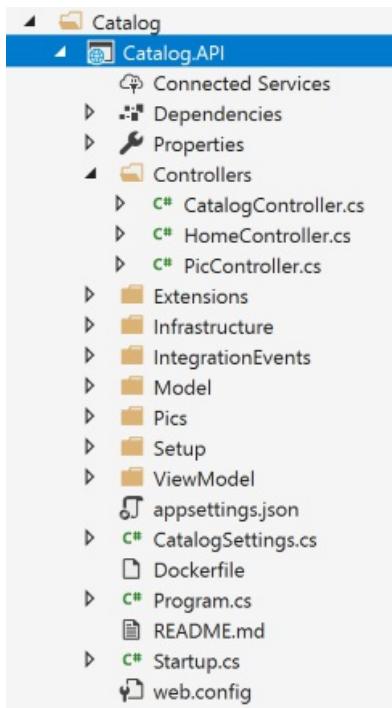


Figura 6-30. Microservicio de API web de ejemplo (microservicio Catalog)

Puede ver que el microservicio Catalog es un proyecto de API web de ASP.NET Core típico con varios controladores y métodos, como en el código siguiente.

```
[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 además de cualquier otra acción requerida.

En lo que respecta a la dirección URL del microservicio, cuando los contenedores se implementan en el equipo de desarrollo local (el host de Docker local), el contenedor de cada microservicio siempre tiene un puerto interno (normalmente el puerto 80) que se especifica en su Dockerfile, como se muestra en el Dockerfile siguiente:

```
FROM microsoft/aspnetcore:2.0.5 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 existen) publicados al implementar con `docker-compose`.

Eso puertos externos no se deben publicar al implementar en un entorno de producción. Por esto precisamente se va a usar la puerta de enlace de API, para evitar la comunicación directa entre las aplicaciones cliente y los microservicios.

Pero durante el desarrollo, le interesa acceder directamente al contenedor o microservicio, y ejecutarlo a través de Swagger. Por eso en eShopOnContainers se siguen especificando los puertos externos, aunque la puerta de enlace de API o las aplicaciones cliente no los vayan a usar.

Este es un ejemplo del archivo `docker-compose.override.yml` para el microservicio Catalog:

```
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 Catalog es el puerto 80, pero el puerto para el acceso externo es 5101. Pero la aplicación no debería usar este puerto cuando se utiliza una puerta de enlace de API, solo para depurar, ejecutar y probar el microservicio Catalog.

Normalmente, no se puede implementar con `docker-compose` en un entorno de producción porque el entorno de implementación de producción correcto para los microservicios es un orquestador como Kubernetes o Service Fabric. Cuando se implementa en esos entornos, se usan otros archivos de configuración en los que no se publica directamente ningún puerto externo para los microservicios, pero siempre se usa al proxy inverso de la puerta de enlace de API.

Ejecute el microservicio Catalog en el host de Docker local ya sea mediante la ejecución de la solución eShopOnContainers completa desde Visual Studio (ejecutará todos los servicios de los archivos `docker-compose`) o iniciando el microservicio con el comando `docker-compose` siguiente en CMD o PowerShell desde la carpeta donde están `docker-compose.yml` y `docker-compose.override.yml`.

```
docker-compose run --service-ports catalog.api
```

Este comando solo ejecuta el contenedor del servicio `catalog.api` además de las dependencias que se especifican en el archivo `docker-compose.yml`. En este caso, el contenedor de SQL Server y el contenedor de RabbitMQ.

Después, puede acceder directamente al microservicio Catalog y ver sus métodos a través de la interfaz de usuario de Swagger, a la que se accede directamente a través de ese puerto "externo", en este caso `http://localhost:5101/swagger`:

The screenshot shows the Swagger UI for the Catalog HTTP API. At the top, there's a navigation bar with icons for back, forward, search, and other settings. The title is "eShopOnContainers - Catalog HTTP API v1". Below the title, it says "Select a spec" and "Catalog.API V1". A link to "/swagger/v1/swagger.json" is also present. A note states: "The Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice sample". There's a "Terms of service" link and an "Authorize" button with a lock icon.

Catalog

- GET /api/v1/Catalog/items**
- PUT /api/v1/Catalog/items**
- POST /api/v1/Catalog/items**
- GET /api/v1/Catalog/items/{id}**
- GET /api/v1/Catalog/Items/withname/{name}**
- GET /api/v1/Catalog/Items/type/{catalogTypeId}/brand/{catalogBrandId}**
- GET /api/v1/Catalog/CatalogTypes**
- GET /api/v1/Catalog/CatalogBrands**
- DELETE /api/v1/Catalog/{id}**

Figura 6-31. Probar el microservicio Catalog con su interfaz de usuario de Swagger

En este momento, 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 comando `docker-compose down`.

Pero la comunicación de acceso directo al microservicio, en este caso a través del puerto externo 5101, es precisamente lo que se quiere evitar en la aplicación. Y se puede evitar si se establece el nivel adicional de direccionamiento indirecto de la puerta de enlace de API (en este caso, Ocelot). De ese 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 software intermedio que se puede aplicar en un orden específico.

Ocelot está diseñado para trabajar solamente con ASP.NET Core. Está destinado a .NET Standard 2.0, por lo que se puede usar en cualquier lugar donde se admita .NET Standard 2.0, incluido el runtime de .NET Core 2.0 y el de .NET Framework 4.6.1 o superior.

Ocelot y sus dependencias se instalan en el proyecto de ASP.NET Core con el paquete [NuGet de Ocelot](#), desde Visual Studio.

```
Install-Package Ocelot
```

En eShopOnContainers, su implementación de puerta de enlace de API es un proyecto ASP.NET Core WebHost simple, y el software intermedio de Ocelot controla todas las características de la puerta de enlace de API, como se muestra en la imagen siguiente:

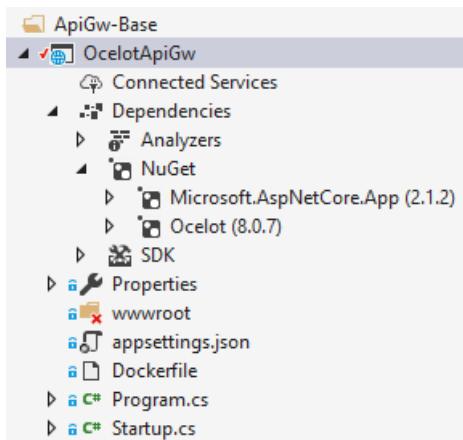


Figura 6-32. El proyecto base de OcelotApiGw en eShopOnContainers

Este proyecto ASP.NET Core WebHost se compila básicamente con dos archivos sencillos: `Program.cs` y `Startup.cs`.

El archivo `Program.cs` solo tiene que crear y configurar el típico `BuildWebHost` de ASP.NET Core.

```
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;
        }
    }
}
```

Aquí, lo importante para Ocelot es el archivo `configuration.json` que se debe proporcionar al generador a través del método `AddJsonFile()`. Ese archivo `configuration.json` es donde se especifican todas las redistribuciones de la puerta de enlace de API, es decir, los puntos de conexión externos con puertos específicos y los puntos de conexión internos correlacionados, que normalmente usan otros puertos.

```
{
    "ReRoutes": [],
    "GlobalConfiguration": {}
}
```

En la configuración hay dos secciones. Una matriz de redistribuciones y `GlobalConfiguration`. Las redistribuciones son los objetos que indican a Ocelot cómo procesar una solicitud de nivel superior. La configuración Global

permite reemplazos de la configuración específica de redistribución. Es útil si no quiere administrar una gran cantidad de configuración específica de redistribución.

Este es un ejemplo simplificado del [archivo de configuración de redistribución](#) de una de las puertas de enlace de API de eShopOnContainers.

```
{
  "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 consiste en aceptar solicitudes HTTP entrantes y reenviarlas a un servicio de nivel inferior, actualmente como otra solicitud HTTP. Ocelot describe el enrutamiento de una solicitud a otra como una redistribución.

Por ejemplo, vamos a centrarnos en una de las redistribuciones del archivo configuration.json anterior, la configuración para el microservicio Basket.

```
{
  "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": []
  }
}
```

DownstreamPathTemplate, Scheme y DownstreamHostAndPorts forman la dirección URL de microservicio interna a la que se va a reenviar esta solicitud.

El puerto es el puerto interno que usa el servicio. Al usar contenedores, el puerto se especifica en su Dockerfile.

Host es un nombre de servicio que depende de la resolución de nombres de servicio que se use. Cuando se usa docker-compose, los nombres de los servicios los proporciona el host de Docker, que usa los nombres de servicio proporcionados en los archivos docker-compose. Si se usa un orquestador como Kubernetes o Service Fabric, ese nombre se debe resolver mediante DNS o la resolución de nombres proporcionada por cada orquestador.

DownstreamHostAndPorts es una matriz que contiene el host y el puerto de los servicios de nivel inferior a los que se quieren reenviar las solicitudes. Normalmente esto solo contendrá una entrada, pero a veces es posible que quiera equilibrar la carga de las solicitudes a los servicios de nivel inferior y Ocelot permite agregar más de una entrada y después seleccionar un equilibrador de carga. Pero si se usa Azure y un orquestador, probablemente una idea mejor sea equilibrar la carga con la infraestructura de nube y de orquestador.

UpstreamPathTemplate es la dirección URL que Ocelot usará para identificar qué DownstreamPathTemplate se va a usar para una solicitud determinada desde el 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) con uno o [varios archivos configuration.json combinados](#), o bien almacenar la configuración en un [almacén de Consul KV](#).

Pero como se mencionó en las secciones de arquitectura y diseño, si realmente quiere tener microservicios autónomos, podría ser mejor dividir esa única puerta de enlace de API monolítica en varias puertas de enlace de API o BFF (back-end para front-end). Para ello, vamos a ver cómo se implementa este enfoque con contenedores de Docker.

Uso de una sola imagen de contenedor de Docker para ejecutar varios tipos de contenedor de puerta de enlace de API y BFF

En eShopOnContainers, se usa una sola imagen de contenedor de Docker con la puerta de enlace de API de Ocelot pero después, en tiempo de ejecución, se crean otros contenedores y servicios para cada tipo de puerta de enlace de API o BFF proporcionando otro archivo configuration.json, mediante un volumen de Docker para acceder a una carpeta distinta del equipo para cada servicio.

Containers API Gateways / BFF

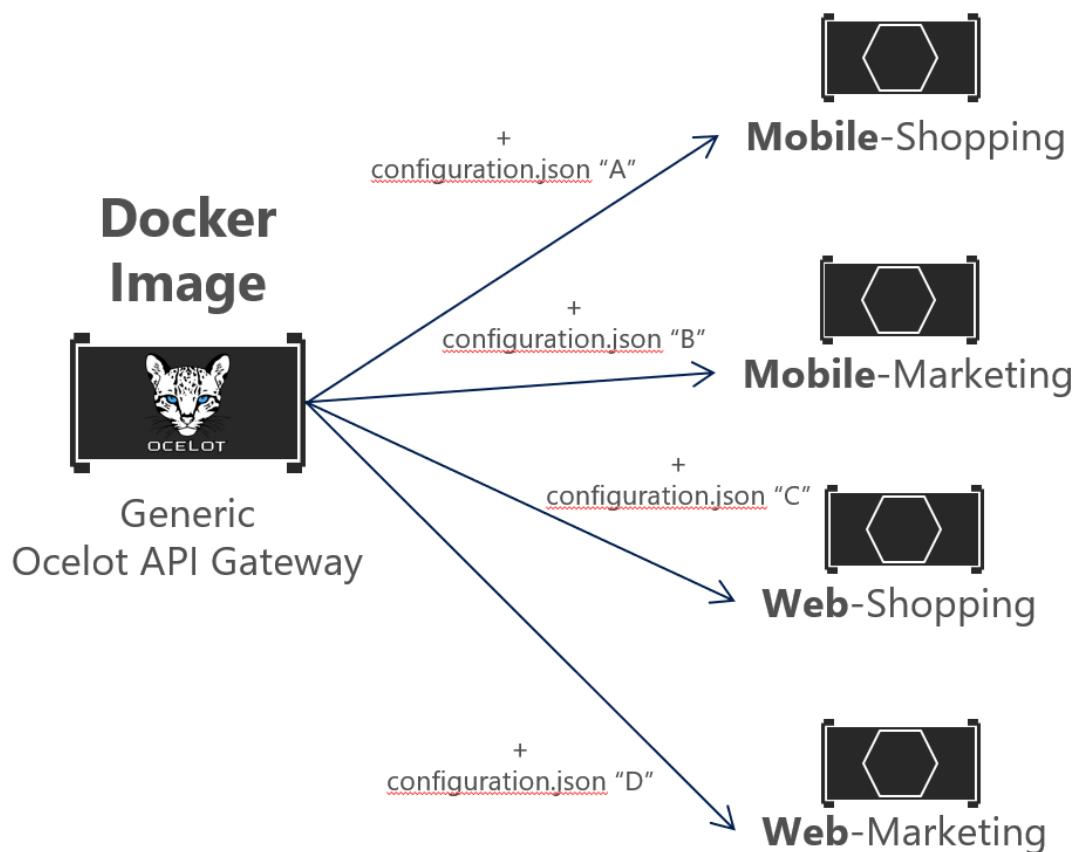


Figura 6-33. Volver a usar una única imagen de Docker de Ocelot entre varios tipos de puerta de enlace de API

En eShopOnContainers, la "imagen de Docker de puerta de enlace de API genérica" se crea con el proyecto "OcelotApiGw" y el nombre de imagen "eshop/ocelotapigw" que se especifica en el archivo docker-compose.yml. Después, al implementar en Docker, habrá cuatro contenedores de puerta de enlace de API que se crean a partir de esa misma imagen de Docker, como se muestra en el extracto siguiente del archivo docker-compose.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 se puede ver en el archivo docker-compose.override.yml siguiente, 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 que se especifica en tiempo de ejecución a través de un volumen de Docker.

```

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.Marketeting/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.Marketeting/apigw:/app/configuration

```

Debido al código anterior, y como se muestra en el Explorador de Visual Studio a continuación, el único archivo necesario para definir cada puerta de enlace de API empresarial específica o BFF es simplemente un archivo configuration.json, dado que las cuatro puertas de enlace de API se basan en la misma imagen de Docker.

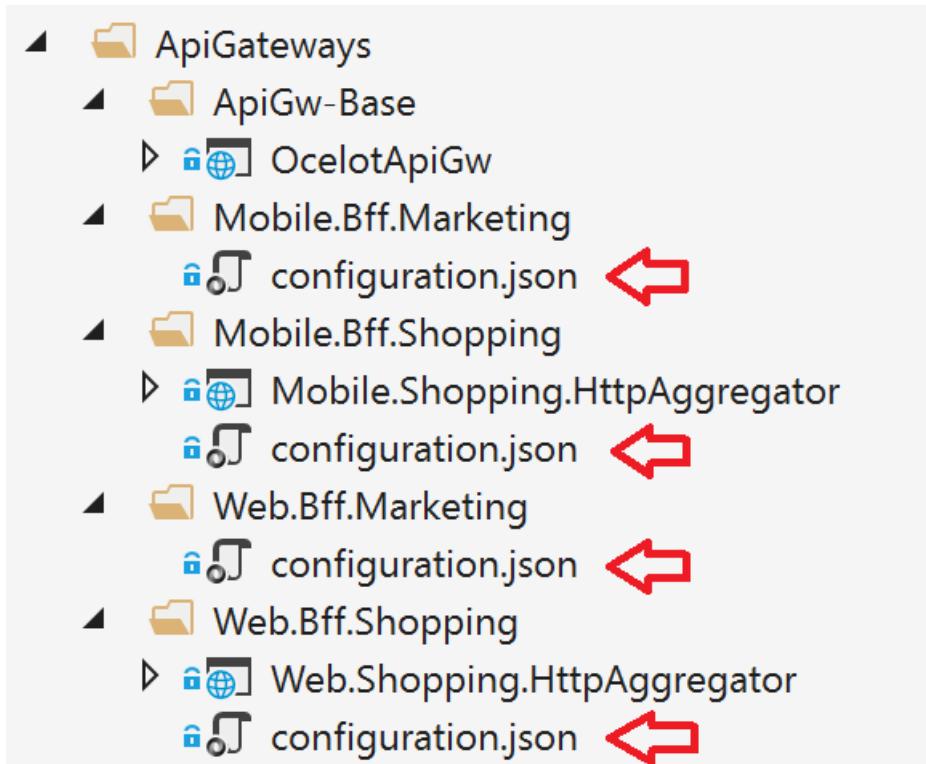


Figura 6-34. El único archivo necesario para definir cada puerta de enlace de API y BFF con Ocelot es un archivo de configuración

Al dividir la puerta de enlace de API en varias, cada equipo de desarrollo centrado en otros subconjuntos de microservicios puede administrar sus propias puertas de enlace de API mediante archivos de configuración de Ocelot independientes. 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 Visual Studio al abrir la solución eShopOnContainers ServicesAndWebApps.sln o al ejecutar "docker-compose up"), se ejecutarán las rutas de ejemplo siguientes.

Por ejemplo, cuando se visita la dirección URL de nivel superior <http://localhost:5202/api/v1/c/catalog/items/2/> que proporciona la puerta de enlace de API webshoppingapigw, se obtiene el mismo resultado de la dirección URL de nivel inferior interna <http://catalog.api/api/v1/2> dentro del host de Docker, como se muestra en el explorador siguiente.

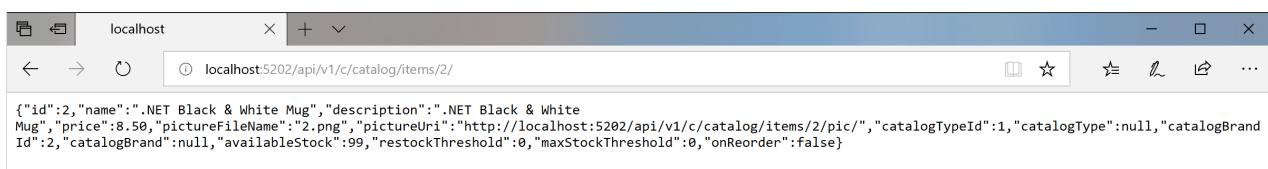


Figura 6-35. Acceso a un microservicio a través de una dirección URL proporcionada por la puerta de enlace de API

Por motivos de pruebas o depuración, si quiere acceder directamente al contenedor de Docker Catalog (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 (la detección de servicios la controlan los nombres de servicio de docker-compose), la única manera de tener acceso directo al contenedor es a través del puerto externo publicado en el archivo docker-compose.override.yml, que solo se proporciona para pruebas de desarrollo, como

<http://localhost:5101/api/v1/Catalog/items/1> en el explorador siguiente.

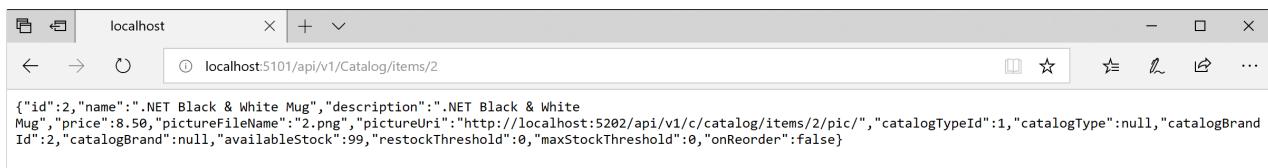


Figura 6-36. Acceso directo a un microservicio con fines de prueba

Pero la aplicación está configurada para que acceda a todos los microservicios a través de las puertas de enlace de API, no a través de "atajos" de puerto directos.

El patrón de agregación de puertas de enlace en eShopOnContainers

Como se mencionó anteriormente, una manera flexible de implementar la agregación de solicitudes consiste en usar servicios personalizados, mediante código. También se podría implementar la agregación de solicitudes con la [característica de agregación de solicitudes en Ocelot](#), pero es posible que no sea tan flexible como se necesita. Por tanto, el método seleccionado para implementar la agregación en eShopOnContainers es mediante un servicio de API web de ASP.NET Core explícito para cada agregador.

Según ese enfoque, el diagrama de composición de las puertas de enlace de API es en realidad más amplio si se tienen en cuenta los servicios de agregador que no se mostraron en el diagrama de arquitectura global simplificado anterior.

En el diagrama siguiente, también se puede ver cómo funcionan los servicios de agregador con sus puertas de enlace de API relacionadas.

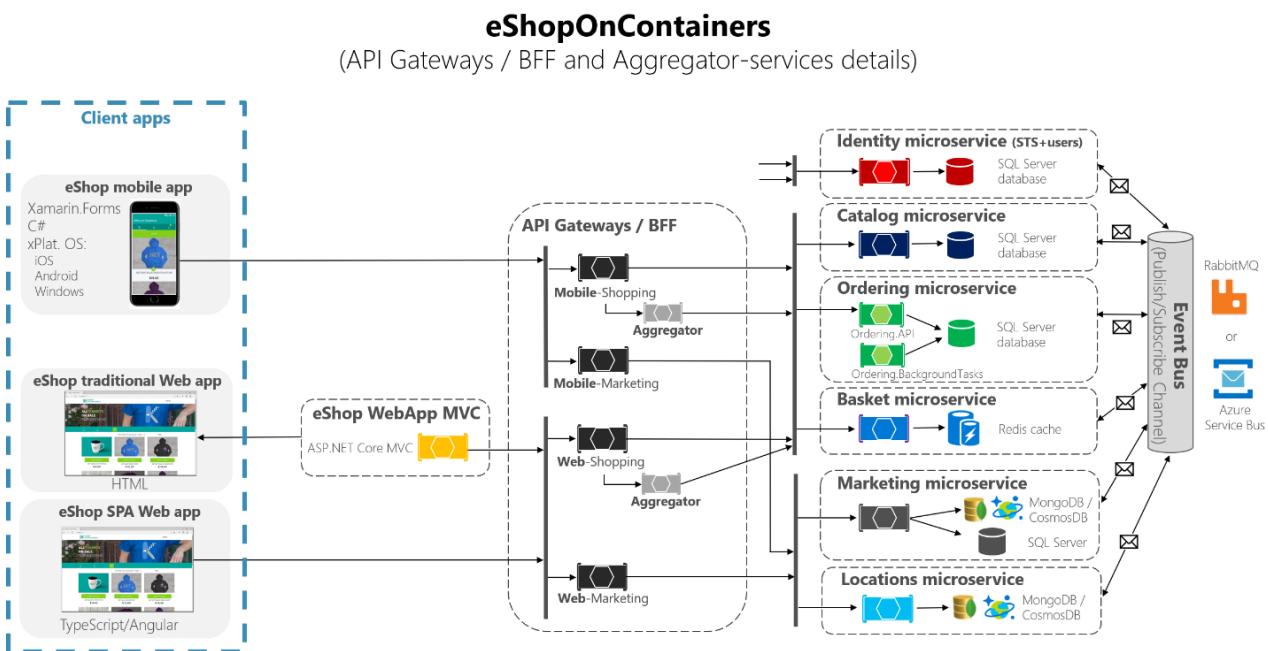


Figura 6-37. Arquitectura de eShopOnContainers con los servicios de agregador

Al ampliar más el área empresarial "Shopping" de la imagen siguiente, se puede ver que al usar los servicios agregadores de las puertas de enlace de API se reduce el intercambio de mensajes entre las aplicaciones cliente y los microservicios.

eShopOnContainers

(API Gateways / BFF and Aggregator-services zoom-in)

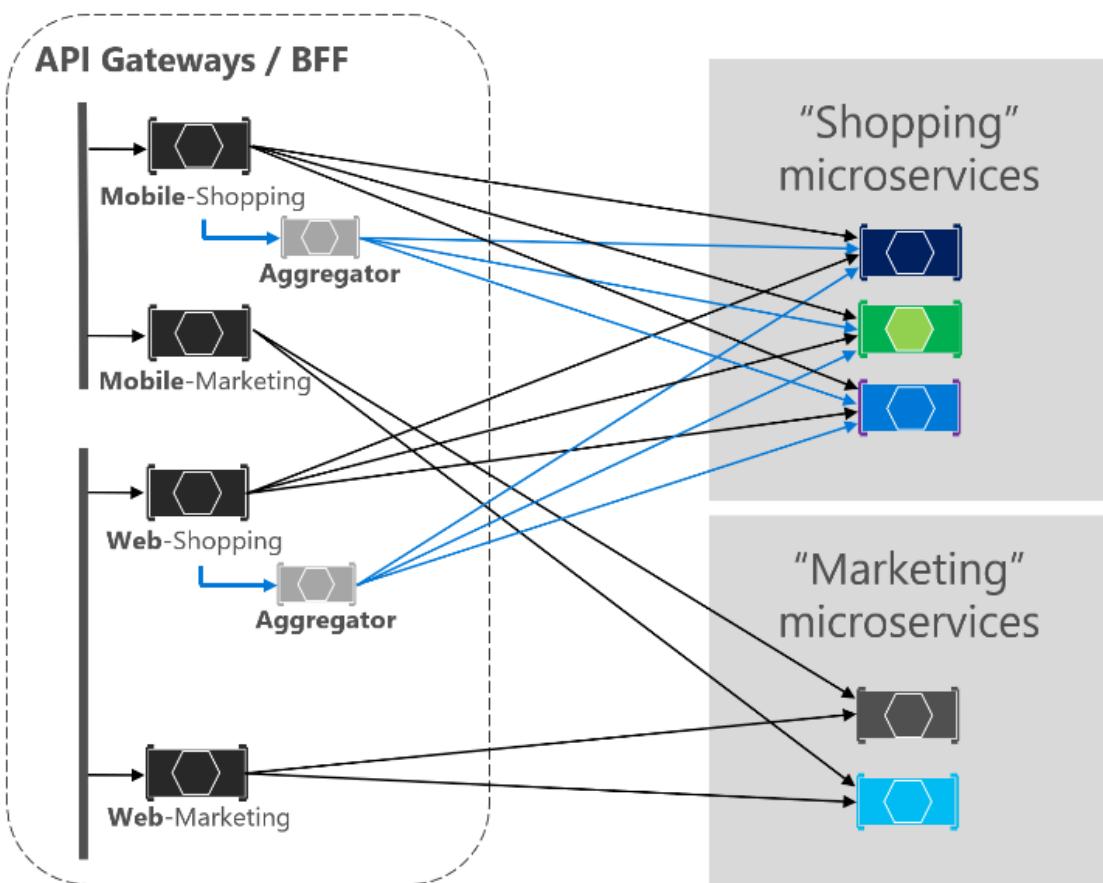


Figura 6-38. Visión ampliada de los servicios de agregador

Se puede observar la complejidad del diagrama cuando se muestran las posibles solicitudes procedentes de las puertas de enlace de API. Aunque se puede ver cómo se simplificarían las flechas de color azul, desde la perspectiva de las aplicaciones cliente, al usar el patrón de agregadores mediante la reducción del intercambio de mensajes y la latencia de la comunicación, en última instancia se mejora de forma significativa la experiencia del usuario, especialmente para las aplicaciones remotas (aplicaciones móviles y SPA).

El caso del área empresarial "Marketing" y los microservicios es un caso de uso muy simple, por lo que no hubo necesidad de usar agregadores, pero podría ser posible, si fuera necesario.

Autenticación y autorización en las puertas de enlace de API de Ocelot

En una puerta de enlace de API de Ocelot se puede ubicar el servicio de autenticación, como un servicio de API web de ASP.NET Core con [IdentityServer](#) para proporcionar el token de autenticación, fuera o dentro de la puerta de enlace de API.

Dado que en eShopOnContainers se usan varias puertas de enlace de API con límites basados en BFF y áreas de negocio, el servicio Identity/Auth se excluye de las puertas de enlace de API, como se resalta en color amarillo en el diagrama siguiente.

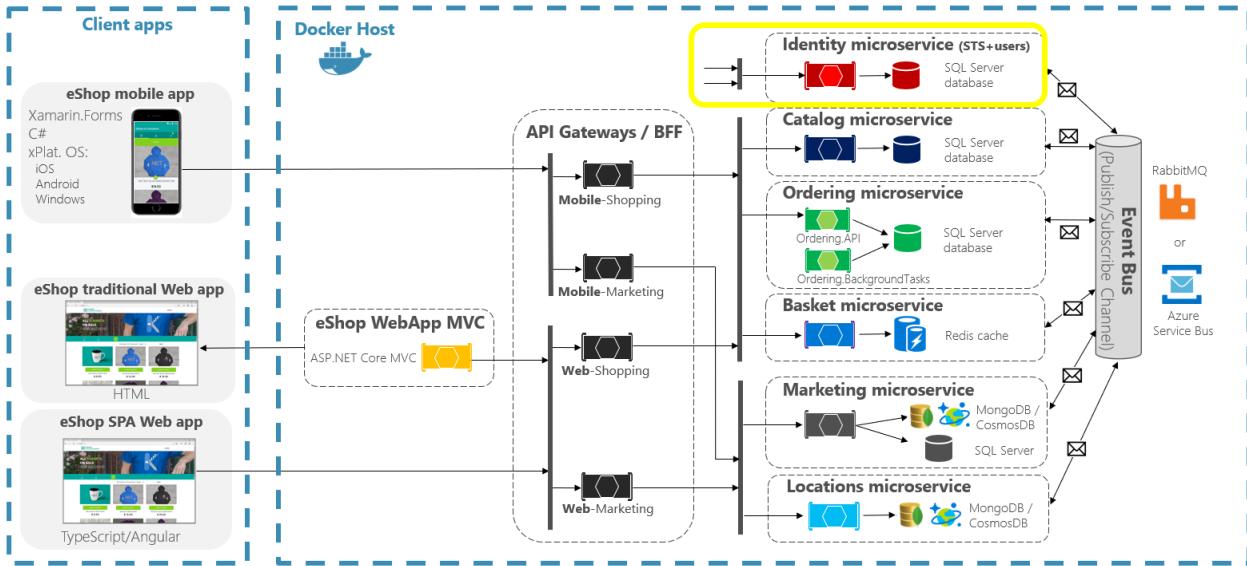


Figura 6-39. Posición del servicio Identity en eShopOnContainers

Pero Ocelot también admite que el microservicio Identity/Auth se sitúe dentro de los límites de la puerta de enlace de API, como se muestra en este otro diagrama.

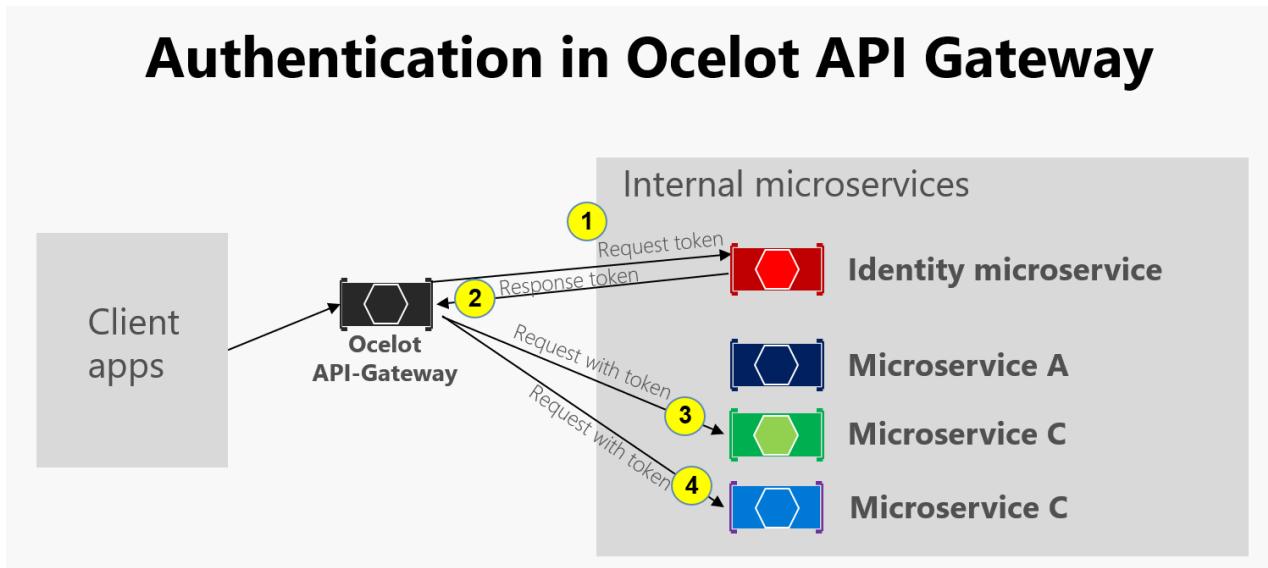


Figura 6-40. Autenticación en Ocelot

Tal y como se muestra en el diagrama anterior, cuando el microservicio Identity está por debajo de la puerta de enlace de API (AG): 1) La puerta de enlace de API solicita un token de autenticación del microservicio Identity; 2) el microservicio Identity devuelve las solicitudes de token a la puerta de enlace de API; 3-4) solicitudes de los microservicios a la puerta de enlace de API mediante el token de autenticación. Como en la aplicación eShopOnContainers se ha dividido la puerta de enlace de API en varios 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 los intereses transversales. Esa opción sería razonable en una arquitectura basada en microservicios más compleja con varios microservicios de intereses transversales. Como en eShopOnContainers solo hay un interés transversal, se decidió controlar solamente el servicio de seguridad fuera del territorio 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 en primer lugar cuando se intenta usar cualquier microservicio protegido. Eso redirige la solicitud HTTP al microservicio Identity o de autenticación para obtener el token de acceso para que los servicios protegidos se puedan visitar con el token de acceso.

La forma de proteger con autenticación cualquier servicio en el nivel de la puerta de enlace de API consiste en

establecer AuthenticationProviderKey en su configuración relacionada en el archivo configuration.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 ejecuta Ocelot, consultará la redistribución AuthenticationOptions.AuthenticationProviderKey y comprobará que hay que un proveedor de autenticación registrado con la clave especificada. Si no lo hay, Ocelot no se iniciará. Si lo hay, la redistribución usará ese proveedor cuando se ejecute.

Como elWebHost de Ocelot está configurado con `authenticationProviderKey = "IdentityApiKey"`, eso requerirá la autenticación cada vez que el servicio tenga alguna solicitud sin ningún token de autenticación.

```
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" }
                };
            });
            //...
        }
    }
}
```

Después, también tendrá que establecer la autorización con el atributo [Authorize] en cualquier recurso al que se vaya a acceder como los microservicios, como en el siguiente controlador del microservicio Basket.

```

namespace Microsoft.eShopOnContainers.Services.Basket.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class BasketController : Controller
    {
        //...
    }
}

```

ValidAudiences como "basket" se ponen en correlación con el público definido en cada microservicio con `AddJwtBearer()` en el método `ConfigureServices()` de la clase `Startup`, como se muestra en el código siguiente.

```

// 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 `Basket`) con una dirección URL de redistribución basada en la puerta de enlace de API como `http://localhost:5202/api/v1/b/basket/1`, obtendrá un error "401 No autorizado" a menos que proporcione un token válido. Por otro lado, si una dirección URL de redistribución está autenticada, Ocelot invocará cualquier esquema de nivel inferior con el que esté asociada (la dirección URL de microservicio interna).

Autorización en el nivel de redistribuciones de Ocelot. Ocelot admite la autorización basada en notificaciones que se evalúa después de la autenticación. La autorización se establece en un nivel de ruta mediante la adición de las líneas siguientes a la configuración de redistribución.

```

"RouteClaimsRequirement": {
    "UserType": "employee"
}

```

En ese ejemplo, cuando se llama al software intermedio de autorización, Ocelot comprobará si el usuario tiene el tipo de notificación "UserType" en el token y si el valor de esa notificación es "employee". En caso contrario, el usuario no tendrá autorización y la respuesta será el error "403 Prohibido".

Uso de entrada de Kubernetes con las puertas de enlace de API de Ocelot

Al usar Kubernetes (como en un clúster de Azure Kubernetes Service) normalmente todas las solicitudes HTTP se unifican a través de la [capa de entrada de Kubernetes](#) basada en *Nginx*.

En Kubernetes, si no se usa ningún enfoque de entrada, los servicios y pods tienen direcciones IP que solo son enrutables por la red de clústeres.

Pero si usa un enfoque de entrada, tendrá una capa intermedia entre Internet y los servicios (incluidas las puertas de enlace de API), que actúa como un proxy inverso.

Como definición, una entrada es una colección de reglas que permiten que las conexiones entrantes lleguen a los servicios de clúster. Normalmente, una entrada se configura para proporcionar a los servicios direcciones URL accesibles de forma externa, tráfico con equilibrio de carga, terminación SSL y mucho más. Los usuarios solicitan la entrada mediante la publicación del recurso de entrada en el servidor de API.

En eShopOnContainers, al desarrollar de forma local y usar solamente el equipo de desarrollo como el host de Docker, no se usa ninguna entrada, solo las diferentes puertas de enlace de API.

Pero cuando el destino es un entorno de "producción" basado en Kubernetes, en eShopOnContainers se usa una entrada delante de las puertas de enlace de API. De este modo, los clientes pueden seguir llamando a la misma dirección URL base, pero las solicitudes se enrutan a varias puertas de enlace de API o BFF.

Tenga en cuenta que las puertas de enlace de API actúan de front-end o fachadas en las que solo se exponen los servicios, pero no las aplicaciones web que suelen estar fuera de su ámbito. Además, es posible que las puertas de enlace de API oculten ciertos microservicios internos.

Pero la entrada simplemente redirige las solicitudes HTTP pero no intenta ocultar ningún microservicio ni aplicación web.

Tener un nivel Nginx de entrada en Kubernetes delante de las aplicaciones web además de las distintas puertas de enlace de API de Ocelot o BFF es la arquitectura ideal, como se muestra en el diagrama siguiente.

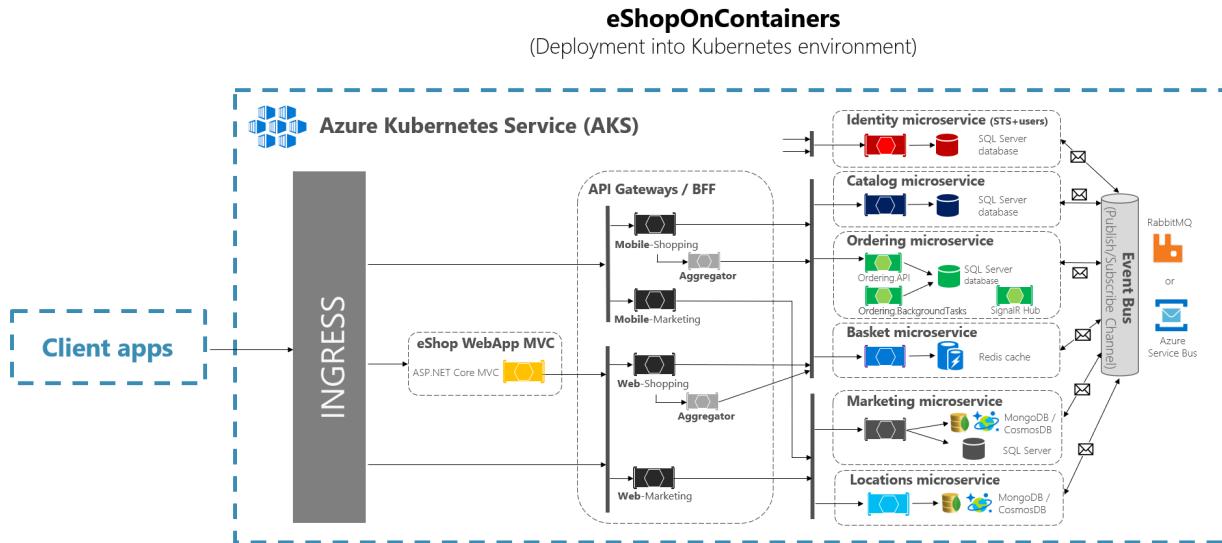


Figura 6-41. El nivel de entrada en eShopOnContainers cuando se implementa en Kubernetes

Una entrada de Kubernetes actúa como un proxy inverso para todo el tráfico a la aplicación, incluidas las aplicaciones web, que normalmente están fuera del ámbito de la puerta de enlace de la API. Al implementar eShopOnContainers en Kubernetes, solo expone algunos servicios o puntos de conexión a través de la *entrada*, básicamente la lista siguiente 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 en la que se muestra el estado o las comprobaciones de estado
- `/webshoppingapigw` para el BFF web y los procesos empresariales de compra
- `/webmarketingapigw` para el BFF web y los procesos empresariales de marketing
- `/mobileshoppingapigw` para el BFF para dispositivos móviles y los procesos empresariales de compra
- `/mobilemarketingapigw` para el BFF para dispositivos móviles y los procesos empresariales de marketing

Al implementar en Kubernetes, cada puerta de enlace de API Ocelot usa un archivo "configuration.json" diferente para cada *pod* en el que se ejecutan las puertas de enlace de API. Dichos archivos "configuration.json" se proporcionan mediante el montaje (originalmente con el script `deploy.ps1`) de 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 `/app/configuration` del contenedor.

En los archivos de código fuente de eShopOnContainers, los archivos "configuration.json" originales se encuentran en la carpeta `k8s/ocelot/`. Hay un archivo para cada BFF o puerta de enlace de API.

Características transversales adicionales en una puerta de enlace de API de Ocelot

Cuando se usa una puerta de enlace de API de Ocelot hay otras características importantes para investigar y utilizar, como se describe en los vínculos siguientes.

- **Detección de servicios en el lado cliente mediante la integración de 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 e interruptores) en el nivel de puerta de enlace de API**
<https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html>
- **Limitación de velocidad**
<https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html>

[ANTERIOR](#)

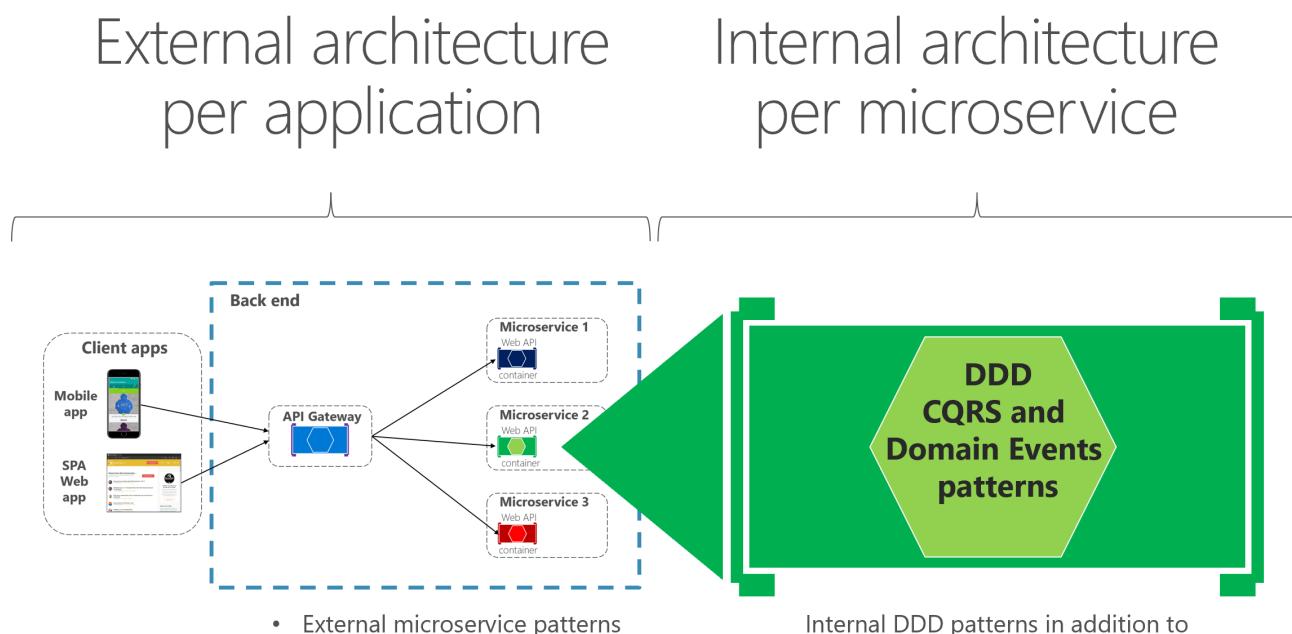
[SIGUIENTE](#)

Abordar la complejidad empresarial en un microservicio con patrones DDD y CQRS

25/11/2019 • 4 minutes to read • [Edit Online](#)

Diseñe un modelo de dominio para cada microservicio o contexto limitado que refleje un conocimiento del ámbito empresarial.

Esta sección se centra en microservicios más avanzados que se implementan cuando se deben abordar subsistemas complejos y en microservicios derivados de los conocimientos de expertos en el dominio con reglas de negocios cambiantes. Los patrones de arquitectura que se usan en esta sección se basan en los enfoques de diseño guiado por el dominio (DDD) y segregación de responsabilidades de comandos y consultas (CQRS), como se ilustra en la figura 7-1.



Diferencia entre la arquitectura externa: patrones de microservicio, puertas de enlace de API, comunicaciones resistentes, pub/sub, etc., y la arquitectura interna: orientada a datos/CRUD, patrones de DDD, inserción de dependencias, varias bibliotecas, etc.

Figura 7-1. Arquitectura de microservicios externa frente a patrones de arquitectura interna para cada microservicio.

Pero la mayoría de las técnicas para microservicios orientados a datos, (por ejemplo, cómo implementar un servicio ASP.NET Core Web API o cómo exponer metadatos de Swagger con Swashbuckle o NSwag) también son aplicables a los microservicios más avanzados que se implementan internamente con patrones DDD. Esta sección es una ampliación de las secciones anteriores, ya que la mayoría de las prácticas explicadas anteriormente también se aplican aquí o a cualquier tipo de microservicio.

Esta sección proporciona en primer lugar detalles sobre los patrones CQRS simplificados que se usan en la aplicación de referencia eShopOnContainers. Más adelante, obtendrá información general sobre las técnicas DDD que le permiten encontrar patrones comunes que puede volver a usar en sus aplicaciones.

DDD es un tema amplio con numerosos recursos para obtener más información. Puede empezar con libros como [Domain-Driven Design](#) (Diseño guiado por el dominio), de Eric Evans, y materiales adicionales de Vaughn Vernon,

Jimmy Nilsson, Greg Young, Udi Dahan, Jimmy Bogard y muchos otros expertos en DDD y 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. Domain Language (Lenguaje de dominio)**

<https://domainlanguage.com/>

- **Martin Fowler. Domain-Driven Design (Diseño orientado al dominio)**

<https://martinfowler.com/tags/domain%20driven%20design.html>

- **Jimmy Bogard. Strengthening your domain: a primer (Reforzar el dominio: conceptos básicos)**

<https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/>

Libros sobre DDD

- **Eric Evans. Diseño orientado al dominio: Tackling Complexity in the Heart of Software (Diseño orientado al dominio: abordar la complejidad en el corazón del software)**

<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

- **Eric Evans. Domain-Driven Design Reference: Definitions and Pattern Summaries (Referencia del diseño orientado al dominio: 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 de un diseño orientado al dominio)**

<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 - Patterns, Principles, and Practices of Domain-Driven Design (Patrones, principios y procedimientos del diseño orientado al dominio)**

<http://www.wrox.com/WileyCDA/WroxTitle/Patterns-Principles-and-Practices-of-Domain-Driven-Design.productCd-1118714709.html>

Aprendizaje de DDD

- **Julie Lerman y Steve Smith. Domain-Driven Design Fundamentals (Fundamentos del diseño orientado al dominio)**

<https://bit.ly/PS-DDD>

ANTERIOR

SIGUIENTE

Aplicación de patrones CQRS y DDD simplificados en un microservicio

23/10/2019 • 6 minutes to read • [Edit Online](#)

CQRS es un patrón de arquitectura que separa los modelos para leer y escribir datos. Bertrand Meyer definió originalmente el término relacionado [Separación de consultas y comandos \(CQS\)](#) en su libro *Construcción de software orientada a objetos*. La idea básica es que puede dividir las operaciones de un sistema en dos categorías claramente separadas:

- Consultas. Devuelven un resultado sin cambiar el estado del sistema y no tienen efectos secundarios.
- Comandos. Cambian el estado de un sistema.

CQS es un concepto simple: se trata de métodos dentro del mismo objeto, que son consultas o comandos. Cada método devuelve o transforma el estado, pero no ambas cosas. Incluso un único objeto de patrón de repositorio puede cumplir con CQS. CQS puede considerarse un principio fundamental para CQRS.

Greg Young introdujo el concepto [Segregación de responsabilidad de consultas y comandos \(CQRS\)](#), que también lo promocionaron mucho Udi Dahan y otros. Se basa en el principio CQS, aunque es más detallado. Se puede considerar un patrón basado en comandos y eventos y, opcionalmente, en mensajes asincrónicos. En muchos casos, CQRS está relacionado con escenarios más avanzados, como tener una base de datos física para operaciones de lectura (consultas) distinta que para operaciones de escritura (actualizaciones). Además, un sistema CQRS más evolucionado podría implementar [Event-Sourcing \(ES\)](#) para la base de datos de las actualizaciones, de modo que solo almacenaría eventos en el modelo del dominio en lugar de almacenar los datos de estado actual. Sin embargo, este no es el enfoque usado en esta guía; estamos usando el enfoque CQRS más sencillo, que consiste simplemente en separar las consultas de los comandos.

La separación que CQRS persigue se consigue mediante la agrupación de las operaciones de consulta en una capa y de los comandos en otra. Cada capa tiene su propio modelo de datos (tenga en cuenta que decimos modelo, no necesariamente una base de datos diferente) y se basa en su propia combinación de patrones y tecnologías. Lo más importante es que las dos capas pueden estar dentro del mismo nivel o microservicio, como en el ejemplo (microservicio de pedidos) usado para esta guía. O pueden implementarse en diferentes microservicios o procesos para que se puedan optimizar y escalar horizontalmente por separado sin que una afecte a la otra.

CQRS significa tener dos objetos para una operación de lectura/escritura cuando en otros contextos solo hay uno. Hay razones para tener una base de datos para las operaciones de lectura sin normalizar, de la cual puede obtener información en la bibliografía sobre CQRS más avanzada. Pero aquí no vamos a usar ese enfoque, ya que el objetivo es tener más flexibilidad en las consultas en lugar de limitar las consultas con las restricciones de patrones de DDD como los agregados.

Un ejemplo de este tipo de servicio es el microservicio de pedidos de la aplicación de referencia de eShopOnContainers. Este servicio implementa un microservicio basado en un enfoque simplificado de CQRS. Usa un solo origen de datos o base de datos, pero dos modelos lógicos, además de patrones de DDD para el dominio transaccional, como se muestra en la figura 7-2.

Simplified CQRS and DDD microservice

High level design

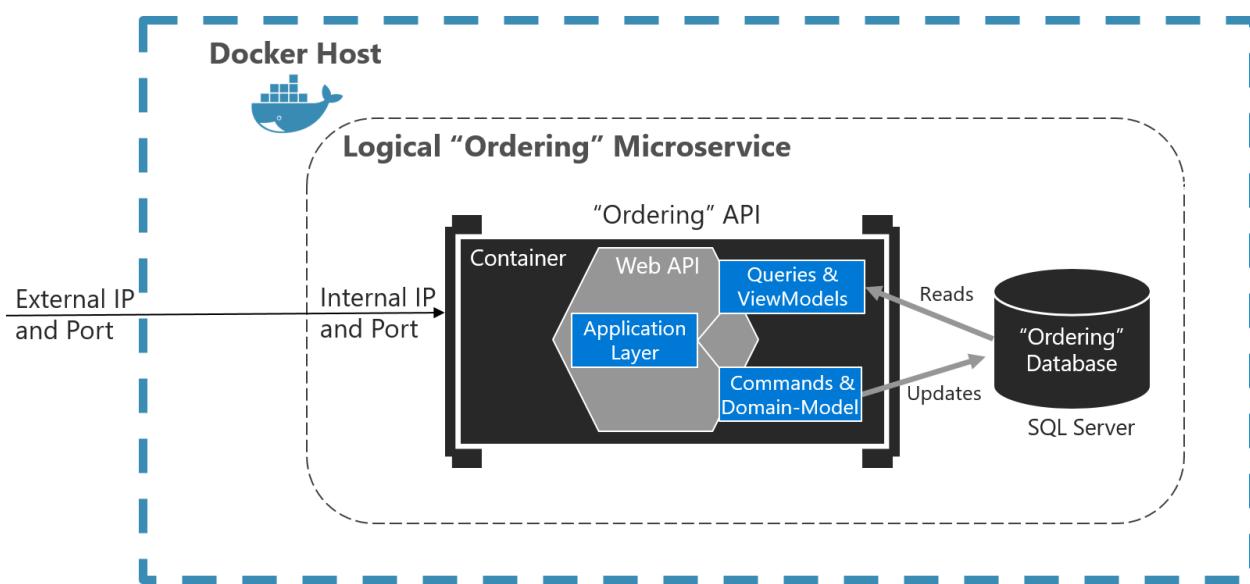


Figura 7-2. Microservicio con CQRS simplificado y basado en DDD

El microservicio "Ordering" lógico incluye su base de datos Ordering, que puede estar, pero no tiene que estar, en el mismo host de Docker. La presencia de la base de datos en el mismo host de Docker es buena para el desarrollo, pero no para producción.

El nivel de aplicación puede ser la propia API web. Aquí, el aspecto de diseño importante es que el microservicio, siguiendo el patrón de CQRS, ha dividido las consultas y los ViewModels (modelos de datos creados especialmente para las aplicaciones cliente) de los comandos, del modelo del dominio y de las transacciones. Este enfoque mantiene las consultas independientes de las restricciones procedentes de los patrones DDD que solo tienen sentido para las transacciones y las actualizaciones, como se explica en secciones posteriores.

Recursos adicionales

- **Greg Young. Versioning in an Event Sourced System** (Control de versiones en un sistema con abastecimiento de eventos, libro electrónico en línea gratuito)
<https://leanpub.com/esversioning/read>

[ANTERIOR](#)

[SIGUIENTE](#)

Aplicación de enfoques CQRS y CQS en un microservicio DDD en eShopOnContainers

23/10/2019 • 6 minutes to read • [Edit Online](#)

El diseño del microservicio Ordering de la aplicación de referencia eShopOnContainers se basa en los principios CQRS. Pero usa el enfoque más sencillo, que consiste simplemente en separar las consultas de los comandos y usar la misma base de datos para ambas acciones.

La esencia de esos patrones y el punto importante es que las consultas son idempotentes: el estado del sistema no cambia independientemente de las veces que se consulte a ese sistema. Es decir, las consultas no sufren 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 Ordering usen la misma base de datos. Por lo tanto, se trata de un enfoque CQRS simplificado.

Por otro lado, los comandos, que producen transacciones y actualizaciones de datos, cambian el estado del sistema. Debe tener cuidado con los comandos cuando trabaje con la complejidad y las reglas de negocio cambiantes. Ahí es donde se prefieren aplicar técnicas de DDD para tener un sistema mejor modelado.

Los patrones DDD presentados en esta guía no se deben aplicar de forma general, ya que establecen restricciones en el diseño. Esas restricciones proporcionan ventajas como una mayor calidad con el tiempo, especialmente en comandos y otro código que modifican el estado del sistema. Pero las restricciones agregan complejidad con menos ventajas para leer y consultar datos.

Un patrón de este tipo es el patrón Aggregate, que se analiza en más detalle en secciones posteriores. En pocas palabras, en el patrón Aggregate, muchos objetos de dominio se tratan 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 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, esta guía sugiere usar patrones DDD solo en el área transaccional o de actualizaciones del microservicio (es decir, como se desencadena con comandos). Las consultas pueden seguir un enfoque más simple y deben separarse de los comandos, según un enfoque 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 Ordering), se ha optado por implementar consultas directas mediante un micro ORM como [Dapper](#). Esto permite implementar cualquier consulta basada en instrucciones SQL para obtener el mejor rendimiento, gracias a un marco de trabajo ligero con muy poca sobrecarga.

Observe que al usar este enfoque, las actualizaciones del modelo que afectan a la conservación de 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 las consultas.

Los patrones CQRS y DDD no son arquitecturas de nivel superior

Es importante entender que CQRS y la mayoría de los patrones DDD (como las capas DDD o un modelo de dominio con agregados) no son estilos arquitectónicos, sino simplemente patrones de arquitectura. Los microservicios, SOA y la arquitectura orientada a eventos (EDA) son ejemplos de estilos de arquitectura.

Describen un sistema de muchos componentes, por ejemplo, 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 enlazados usan distintos patrones. Tienen responsabilidades diferentes y eso da lugar a distintas soluciones. Merece la pena resaltar que la aplicación del mismo patrón en todos los sitios da lugar a errores. No use patrones CQRS y DDD en cualquier lugar. Muchos subsistemas, contextos enlazados o microservicios son más sencillos y se pueden implementar con más facilidad mediante servicios CRUD simples o con otro enfoque.

Solo hay una arquitectura de aplicación: la arquitectura del sistema o la aplicación de un extremo a otro que se está diseñando (por ejemplo, la arquitectura de microservicios). Pero el diseño de cada contexto enlazado o microservicio de esa aplicación refleja sus propias compensaciones y decisiones de diseño internas en un nivel de patrones de arquitectura. No intente aplicar los mismos patrones arquitectónicos, como CQRS o DDD, en cualquier lugar.

Recursos adicionales

- **Martin Fowler. CQRS**

<https://martinfowler.com/bliki/CQRS.html>

- **Greg Young. Documentos de CQRS**

https://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf

- **Udi Dahan. CQRS aclarado**

<http://udidahan.com/2009/12/09/clarified-cqrs/>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación de lecturas/consultas en un microservicio CQRS

25/11/2019 • 15 minutes to read • [Edit Online](#)

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. Esto 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.

High level “Queries-side” in a simplified CQRS

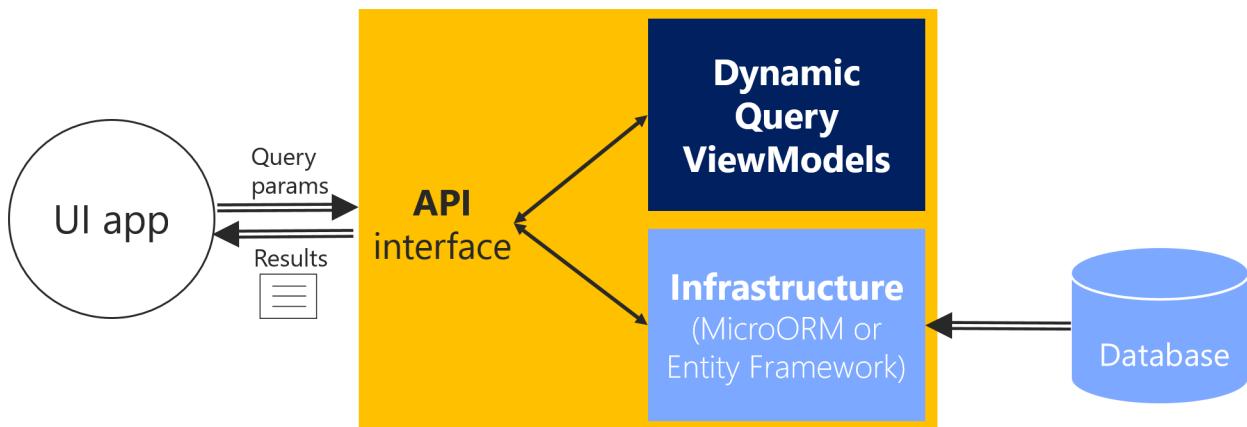


Figura 7-3. El enfoque más sencillo para las consultas en un microservicio CQRS

El enfoque más sencillo para el lado de las consultas en un enfoque CQRS simplificado se puede implementar simplemente 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. Basta con consultar 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 estéticamente 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](#). Esto se muestra en la Figura 7-4. Las consultas se definen en el proyecto de microservicio **Ordering.API** dentro de la solución eShopOnContainers.

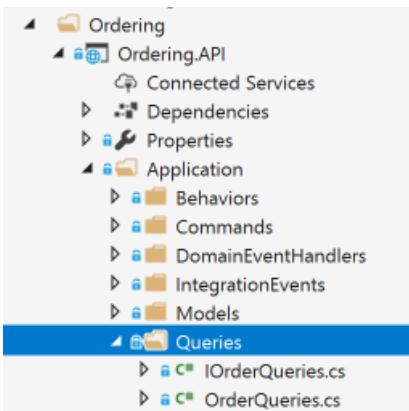


Figura 7-4. Consultas (Queries) en el microservicio de pedidos (Ordering) en eShopOnContainers

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.

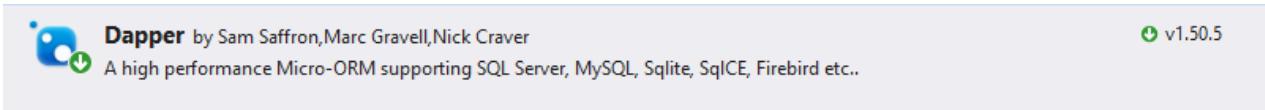
Los datos devueltos (ViewModel) pueden ser el resultado de combinar datos de varias entidades o tablas de la base de datos, o incluso de varios agregados definidos en el modelo de dominio para el área transaccional. En este caso, dado que va a crear consultas independientes del modelo de dominio, se ignoran completamente 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 ViewModel pueden ser tipos estáticos definidos en las clases. O bien, se pueden crear dinámicamente en función de las consultas realizadas (tal y como se implementa en el microservicio de pedidos), lo que resulta muy 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 muy 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 instrucción `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 `System.Data.SqlClient`. Mediante el método `QueryAsync` y otros métodos de extensión que extienden la clase `SqlConnection`, simplemente 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 ViewModel o DTO pueden definirse explícitamente (como clases de contenedor de datos) como la clase `OrderSummary` que se muestra en un fragmento de código más adelante, o simplemente se podrían devolver ViewModel o DTO dinámicos únicamente 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.

```
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 bastante 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, incluso si solo los utiliza 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 muy 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.

```
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). Estos se administran en las anotaciones de datos y en los comentarios XML.

Si 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:

```

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:

```

public class OrderSummary
{
    public int ordernumber { get; set; }
    public DateTime date { get; set; }
    public string status { get; set; }
    public double total { get; set; }
}

```

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/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://docs.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://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio>

[ANTERIOR](#)
SIGUIENTE

Diseño de un microservicio orientado a DDD

25/11/2019 • 19 minutes to read • [Edit Online](#)

El diseño guiado por el dominio (DDD) propone un modelado basado en la realidad de negocio con relación a sus casos de uso. En el contexto de la creación de aplicaciones, DDD hace referencia a los problemas como dominios. Describe áreas con problemas independientes como contextos delimitados (cada contexto delimitado está correlacionado con un microservicio) y resalta un lenguaje común para hablar de dichos problemas. También sugiere muchos patrones y conceptos técnicos, como entidades de dominio con reglas de modelos enriquecidos (no [modelos de dominio anémico](#)), objetos de valor, agregados y raíz agregada (o entidad raíz) para admitir la implementación interna. En esta sección se explica el diseño y la implementación de estos patrones internos.

A veces, estos patrones y reglas técnicas de DDD se perciben como obstáculos con una curva de aprendizaje pronunciada a la hora de implementar opciones de DDD. Pero lo importante no son los patrones en sí, sino organizar el código para que esté en línea con los problemas del negocio y utilizar los mismos términos empresariales (lenguaje ubicuo). Además, las opciones de DDD solo deben aplicarse en el caso de implementar microservicios complejos con reglas de negocio importantes. Las responsabilidades más sencillas, como el servicio CRUD, se pueden administrar con enfoques más sencillos.

La clave está en dónde situar los límites al diseñar y definir un microservicio. Los patrones de DDD le ayudan a comprender la complejidad del dominio. En el modelo de dominio de cada contexto delimitado, debe identificar y definir las entidades, los objetos de valor y los agregados que modelan el dominio. Debe crear y perfeccionar un modelo de dominio que se encuentre dentro de un límite definido por su contexto. Y esto se hace claramente patente en la forma de un microservicio. Los componentes situados dentro de esos límites acaban siendo sus microservicios, aunque, en algunos casos, los contextos delimitados o los microservicios pueden estar compuestos de varios servicios físicos. El DDD afecta a los límites y, por lo tanto, a los microservicios.

Mantener los límites de contexto del microservicio relativamente estrechos

Determinar dónde colocar los límites entre contextos delimitados equilibra dos objetivos contrapuestos. En primer lugar, le interesa crear los microservicios más pequeños posibles, aunque su principal objetivo no debe ser este, sino el de crear un límite alrededor de elementos que deban estar cohesionados. En segundo lugar, le interesa evitar comunicaciones locuaces entre microservicios. Estos objetivos pueden ser contrapuestos. Para encontrar el equilibrio entre ellos, debe descomponer el sistema en tantos microservicios pequeños como pueda hasta que los límites de la comunicación crezcan rápidamente con cada intento adicional de separar un nuevo contexto delimitado. La cohesión es clave en un único contexto delimitado.

Se parece a una [inadecuada intuición de código de cercanía](#) al implementar las clases. Si dos microservicios necesitan colaborar mucho entre sí, probablemente sean el mismo microservicio.

Otra manera de enfocarlo es observando la autonomía. Si un microservicio debe depender de otro servicio para satisfacer directamente una solicitud, no es realmente autónomo.

Niveles en microservicios de DDD

La mayoría de aplicaciones de empresa con una significativa complejidad empresarial y técnica se definen a partir de múltiples niveles. Los niveles son un elemento lógico y no están relacionados con la implementación del servicio, sino que sirven para ayudar a los desarrolladores a administrar la complejidad del código. Los diferentes niveles (por ejemplo, el nivel de modelo de dominio frente al nivel de presentación, etc.) pueden ser de diferentes tipos, lo que exige su traducción.

Por ejemplo, una entidad se puede cargar desde la base de datos. Puede ser que se envíe parte de esa información, o un agregado de información que incluya datos adicionales de otras entidades, a la interfaz de usuario del cliente a través de una API web de REST. La cuestión es que la entidad de dominio se debe situar dentro del nivel de modelo de dominio y no puede propagarse a otras áreas a las que no pertenece, como al nivel de presentación.

Además, debe tener entidades siempre válidas (consulte la sección [Diseño de validaciones en el nivel de modelo de dominio](#)) y controladas por raíces agregadas (entidades raíz). Por lo tanto, las entidades no pueden estar enlazadas a vistas de cliente, porque puede ser que algunos datos no estén validados en el nivel de la interfaz de usuario. Para esto sirve el modelo ViewModel. El modelo ViewModel es un modelo de datos exclusivo para las necesidades del nivel de presentación. Las entidades de dominio no pertenecen directamente al modelo ViewModel. En cambio, debe traducir entre ViewModels y entidades de dominio, y viceversa.

Al hablar de complejidad, es importante tener un modelo de dominio controlado por raíces agregadas que garanticen que todas las invariantes y reglas relacionadas con ese grupo de entidades (agregadas) se realicen a través de un punto de entrada o puerta únicos: la raíz agregada.

En la Figura 7-5 se muestra cómo se implementa un diseño por niveles en la aplicación eShopOnContainers.

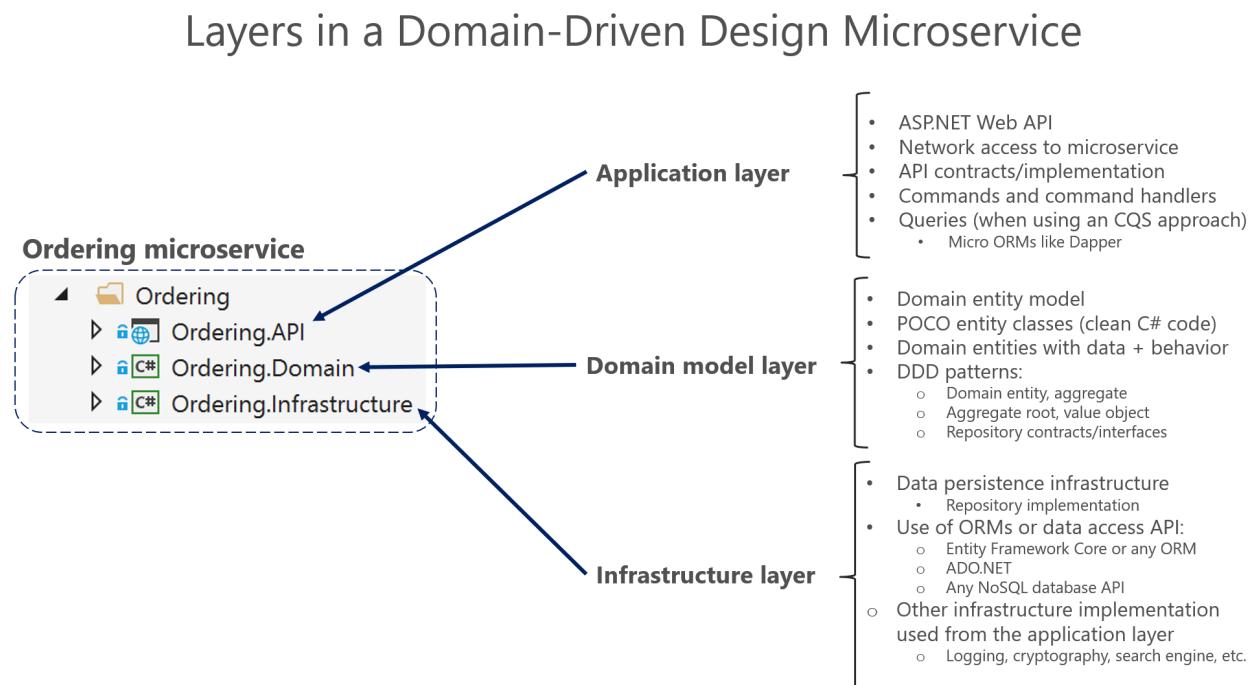


Figura 7-5. Niveles de DDD en el microservicio de ordenación en eShopOnContainers

Las tres capas en un microservicio DDD como Ordering. Cada capa es un proyecto de VS: la capa de aplicación es Ordering.API, el nivel de dominio es Ordering.Domain y el nivel de infraestructura es Ordering.Infrastructure. Le recomendamos que diseñe el sistema de modo que cada nivel se comunique solamente con otros niveles determinados. Esto puede ser más fácil de aplicar si los niveles se implementan como bibliotecas de clase distintas, porque puede identificar claramente qué dependencias se establecen entre bibliotecas. Por ejemplo, el nivel de modelo de dominio no debe depender de ningún otro nivel (las clases del modelo de dominio deben ser clases de objetos CLR estándar o [POCO](#)). Como se muestra en la Figura 7-6, la biblioteca de nivel **Ordering.Domain** solo tiene dependencias en las bibliotecas .NET Core o en los paquetes NuGet, pero no en otras bibliotecas personalizadas, como la biblioteca de datos o de persistencia.

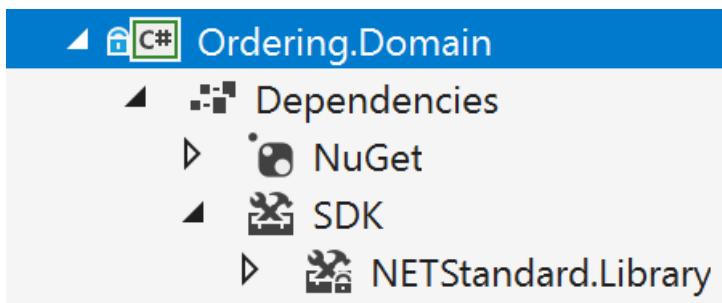


Figura 7-6. Los niveles implementados como bibliotecas permiten controlar mejor las dependencias entre niveles

El nivel de modelo de dominio

En el fantástico libro de Eric Evans, [Domain Driven Design](#) (Diseño guiado por el dominio), se explica lo siguiente sobre el nivel de modelo de dominio y el nivel 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 empresarial está controlado y se usa aquí, aunque los detalles técnicos de su almacenaje se delegan a la infraestructura. Este nivel es el núcleo del software empresarial.

En el nivel de modelo de dominio es donde se expresa el negocio. Al implementar un nivel de modelo de dominio de microservicio en .NET, este nivel se codifica como una biblioteca de clases con las entidades de dominio que capturan datos y comportamiento (métodos con lógica).

Siguiendo los principios de [omisión de persistencia](#) y [omisión de infraestructura](#), este nivel debe omitir completamente los detalles de persistencia de datos. Las tareas de persistencia deben estar realizadas por el nivel de infraestructura. Por lo tanto, este nivel no debe tener dependencias directas en la infraestructura, lo que significa que una regla de importante es que las clases de entidad del modelo de dominio deben ser [POCO](#).

Las entidades de dominio no deben depender directamente (como derivarse de una clase base) de ningún marco de infraestructura de acceso a los datos, como Entity Framework o NHibernate. Lo ideal es que las entidades de dominio no se deriven de ningún tipo definido en ningún marco de infraestructura ni lo implementen.

Los marcos ORM más modernos, como Entity Framework Core, permiten este enfoque, de forma que las clases de modelo de dominio no se acoplan a la infraestructura. Pero no siempre se puede disponer de entidades POCO al usar marcos y bases de datos NoSQL determinados, como actores y colecciones de confianza en Azure Service Fabric.

Incluso cuando es importante seguir el principio de omisión de persistencia en el modelo de dominio, no debe ignorar los problemas de persistencia. Sigue siendo muy importante comprender el modelo de datos físicos y cómo se asigna a un modelo de objetos entidad. En caso contrario, puede crear diseños imposibles.

Además, esto no significa que pueda tomar un modelo diseñado para una base de datos relacional y moverla directamente a un NoSQL o a una base de datos orientada a un documento. En algunos modelos de entidad, es posible que el modelo encaje, pero normalmente no lo hace. Sigue habiendo restricciones que el modelo de entidad debe cumplir, basándose en la tecnología de almacenamiento y en la tecnología ORM.

El nivel de aplicación

Si pasamos al nivel de aplicación, podemos citar de nuevo el libro de Eric Evans [Domain Driven Design](#) (Diseño guiado por el dominio):

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 que son responsabilidad de este nivel son significativas para la empresa o necesarias para la interacción con los niveles de aplicación de otros sistemas. Este nivel debe mantenerse estrecho. No contiene reglas de negocios ni conocimientos, sino que solo coordina tareas y delega trabajo a colaboraciones de objetos de dominio en el siguiente nivel. No tiene ningún estado que refleje la situación empresarial, pero puede tener un estado que refleje el progreso de una tarea para el usuario o el

programa.

Normalmente, el nivel de aplicación de microservicios en .NET se codifica como un proyecto de ASP.NET Core Web API. El proyecto implementa la interacción del microservicio, el acceso a redes remotas y la API web externa utilizada desde aplicaciones cliente o de interfaz de usuario. Incluye consultas si se utiliza un enfoque de CQRS, comandos aceptados por el microservicio e incluso comunicación guiada por eventos entre microservicios (eventos de integración). La ASP.NET Core Web API que representa el nivel de aplicación no puede contener reglas de negocios ni conocimientos del dominio (especialmente reglas de dominio para transacciones o actualizaciones); estos deben pertenecer a la biblioteca de clases del modelo de dominio. El nivel de aplicación solo debe coordinar tareas y no puede contener ni definir ningún estado de dominio (modelo de dominio). Delega la ejecución de reglas de negocios a las mismas clases de modelo de dominio (raíces agregadas y entidades de dominio) que, en última instancia, actualizarán los datos en esas entidades de dominio.

Básicamente, la lógica de la aplicación es el lugar en el que 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 del dominio en el nivel de modelo de dominio, sus invariables, el modelo de datos y las reglas de negocios relacionadas sean totalmente independientes de los niveles de presentación y aplicación. Sobre todo, el nivel de modelo de dominio no puede depender directamente de ningún marco de infraestructura.

El nivel de infraestructura

El nivel de infraestructura es la forma en que los datos que inicialmente se conservan en las entidades de dominio (en la memoria) se guardan en bases de datos o en otro almacén permanente. Un ejemplo sería usar código de Entity Framework Core para implementar las clases del patrón de repositorio que usan DbContext para conservar los datos en una base de datos relacional.

De conformidad con los principios [Omisión de persistencia](#) y [Omisión de infraestructura](#) mencionados anteriormente, el nivel de infraestructura no puede "contaminar" el nivel de modelo de dominio. No puede depender demasiado de los marcos para mantener las clases de entidad de modelo de dominio apartadas de la infraestructura que utiliza para conservar datos (EF o cualquier otro marco). La biblioteca de clases de nivel de modelo de dominio solo debe tener el código de dominio, solo clases de entidad [POCO](#) que implementen la esencia del software y debe estar completamente desacoplada de tecnologías de infraestructura.

Así, los proyectos y bibliotecas de clases o niveles dependerán, en última instancia, del nivel de modelo de dominio (biblioteca) y no al revés, como se muestra en la Figura 7-7.

Dependencies between Layers in a Domain-Driven Design service

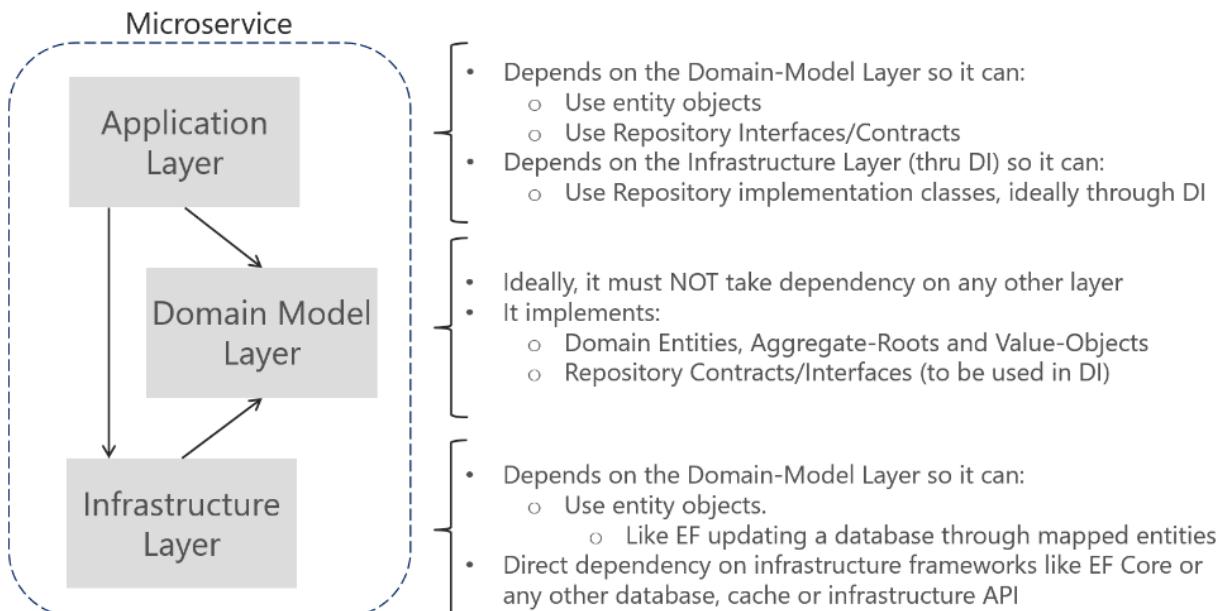


Figura 7-7. Dependencias existentes entre niveles en DDD

Dependencias en un servicio de 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 nivel debe ser independiente para cada microservicio. Como se indicó anteriormente, puede implementar microservicios más complejos siguiendo patrones DDD, al mismo tiempo que puede implementar microservicios guiados por datos más simples (un único CRUD en un solo nivel) de una forma más sencilla.

Recursos adicionales

- **DeviQ. Principio de omisión de persistencia**

<https://deviq.com/persistence-ignorance/>

- **Oren Eini. Omisión de infraestructura**

<https://ayende.com/blog/3137/infrastructure-ignorance>

- **Angel Lopez. Arquitectura por capas en un diseño controlado por dominios**

<https://ajlopez.wordpress.com/2008/09/12/layered-architecture-in-domain-driven-design/>

[ANTERIOR](#)

[SIGUIENTE](#)

Diseño de un modelo de dominio de microservicio

25/11/2019 • 20 minutes to read • [Edit Online](#)

Defina un modelo de dominio enriquecido para cada microservicio de negocios o contexto delimitado.

El objetivo es crear un modelo de dominio coherente único para cada microservicio de negocio o contexto delimitado (BC). Pero tenga en cuenta que en ocasiones un BC o microservicio de negocio puede estar compuesto por varios servicios físicos que comparten un único modelo de dominio. El modelo de dominio debe capturar las reglas, el comportamiento, el lenguaje de negocios y las restricciones del contexto delimitado o microservicio de negocio que representa.

El modelo de entidad del dominio

Las entidades representan objetos del dominio y se definen principalmente por su identidad, continuidad y persistencia en el tiempo y no solo por los atributos que las componen. Como afirma 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 para un modelo. Por tanto, debe identificarlas y diseñarlas cuidadosamente.

La identidad de una entidad puede abarcar varios microservicios o contextos delimitados.

La misma identidad (es decir, el mismo valor de `Id`, aunque quizás no sea la misma entidad de dominio) se puede modelar en varios contextos delimitados o microservicios. Pero eso no implica que la misma entidad, con los mismos atributos y lógica, se implemente en varios contextos delimitados. En su lugar, las entidades de cada contexto delimitado limitan sus atributos y comportamientos a los requeridos en el dominio de ese contexto delimitado.

Por ejemplo, es posible que la entidad de comprador tenga la mayoría de los atributos de una persona que estén definidos en la entidad de usuario en el microservicio de perfiles o identidades, incluida la identidad. Pero la entidad de comprador en el microservicio de pedidos podría tener menos atributos, porque solo determinados datos del comprador están relacionados con el proceso de pedido. El contexto de cada microservicio o contexto delimitado afecta a su modelo de dominio.

Las entidades de dominio deben implementar el comportamiento además de los atributos de datos.

Una entidad de dominio en DDD debe implementar la lógica del dominio o el comportamiento relacionado con los datos de entidad (el objeto al que se obtiene acceso en memoria). Por ejemplo, como parte de una clase de entidad de pedido debería implementar la lógica de negocios y las operaciones como métodos para tareas como agregar un elemento de pedido, la validación de datos y el cálculo total. Los métodos de la entidad se encargan de las invariables y las reglas de la entidad en lugar de tener esas reglas distribuidas por el nivel de aplicación.

En la figura 7-8 se muestra una entidad de dominio que implementa no solo los atributos de datos, sino también las operaciones o los métodos con lógica de dominio relacionada.

Domain Entity pattern

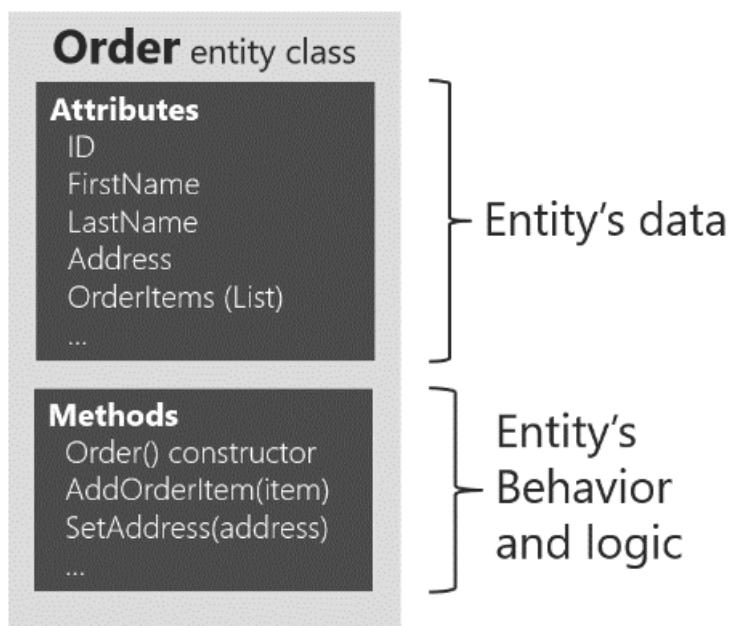


Figura 7-8. Ejemplo de un diseño de entidad de dominio en el que se implementan datos y comportamiento

Una entidad del modelo de dominio implementa comportamientos a través de métodos, es decir, no es un modelo "anémico". Evidentemente, en ocasiones puede tener entidades que no implementen ninguna lógica como parte 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 mayor parte de la lógica se define en la raíz agregada. Si tiene un microservicio complejo con gran cantidad de lógica implementada en las clases de servicio en lugar de en las entidades de dominio, podría encontrarse en el modelo de dominio anémico que se explica en la sección siguiente.

Diferencias entre el modelo de dominio y el 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 en función de los nombres del espacio de dominio, que están conectados con las relaciones enriquecidas y la estructura de los modelos de dominio reales. 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 usan desde un conjunto de objetos de servicio (denominado tradicionalmente *capa de negocio*) que captura toda la lógica de negocios o de dominio. La capa de negocio se encuentra en la parte superior del modelo de datos y usa el modelo de datos al igual que los datos.

El modelo de dominio anémico es simplemente un diseño de estilo de procedimientos. Los objetos de entidad anémicos no son objetos reales, ya que carecen de comportamiento (métodos). Solo contienen propiedades de datos y, por tanto, no se trata de un diseño orientado a objetos. Al colocar todo el comportamiento en objetos de servicio (la capa de negocio), básicamente se crea [código espagueti](#) o [scripts de transacción](#), y, por tanto, se pierden las ventajas que proporciona un modelo de dominio.

Pero si el microservicio o contexto delimitado es muy sencillo (un servicio CRUD), es posible que sea suficiente con el modelo de dominio anémico en forma de objetos de entidad con solo propiedades de datos y que no merezca la pena implementar modelos 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 pedidos implementa patrones DDD, pero el microservicio de catálogo, que es un servicio CRUD simple, no lo hace.

Hay usuarios que afirman que el modelo de dominio anémico es un antipatrón. En realidad, depende de lo que se vaya a implementar. Si el microservicio que se va a crear es bastante sencillo (por ejemplo, un servicio CRUD), seguir el modelo de dominio anémico no es un antipatrón. Pero si es necesario abordar la complejidad del dominio de un microservicio que tiene muchas reglas de negocio cambiantes, es posible que el modelo de dominio anémico sea un antipatrón para ese microservicio o contexto delimitado. En ese caso, es posible que diseñarlo como un modelo enriquecido con entidades que contienen datos y comportamiento además de implementar otros patrones DDD (agregados, objetos de valor, etc.) tenga enormes ventajas para el éxito a largo plazo de este tipo de 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 mencionado Eric Evans, "Muchos objetos no tienen identidad conceptual. Estos objetos describen ciertas características de una cosa".

Una entidad requiere una identidad, pero en un sistema hay muchos objetos que no lo hacen, como el patrón de objeto de valor. Un objeto de valor es un objeto sin identidad conceptual que describe un aspecto de dominio. Se trata de objetos de los que se crea una instancia para representar elementos de diseño que solo interesan temporalmente. Interesa lo *que* son, no *quiénes* son. Los números y las cadenas son algunos ejemplos, pero también pueden ser conceptos de nivel superior como grupos de atributos.

Es posible que algo que sea una entidad en un microservicio no lo sea en otro, porque en el segundo caso, es posible que el contexto delimitado tenga un significado diferente. Por ejemplo, una dirección en una aplicación de comercio electrónico podría no tener ninguna identidad, ya que es posible que solo represente un grupo de atributos del perfil de cliente para una persona o empresa. En este caso, la dirección se debería clasificar como un objeto de valor. Pero en una aplicación para una empresa de energía eléctrica, la dirección del cliente podría ser importante para el dominio de negocio. Por tanto, la dirección debe tener una identidad para poder vincular el sistema de facturación directamente con la dirección. En ese caso, una dirección debería clasificarse como una entidad de dominio.

Una persona con un nombre y apellido normalmente es una entidad debido a que una persona tiene identidad, incluso si el nombre y apellido coinciden con otro conjunto de valores, por ejemplo si también hacen referencia a otra persona.

Los objetos de valor son difíciles de administrar en bases de datos relacionales y ORM como EF, mientras que en las bases de datos orientadas a documentos son más fáciles de implementar y usar.

EF Core 2.0 incluye la característica [Entidades poseídas](#) que facilita administrar los 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 el desarrollo controlado por pruebas**

<https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects>

- **Eric Evans. Diseño orientado al dominio: abordar la complejidad en el corazón del software.**

(Libro; incluye una descripción 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 grupos de entidades de datos diferentes y procesos que pueden controlar un área importante de funcionalidad, como el cumplimiento de pedidos o el inventario. Una unidad de DDD más específica es el agregado, que describe un clúster o grupo de entidades y comportamientos que se pueden tratar como una unidad coherente.

Normalmente, un agregado se define según las transacciones que se necesitan. 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 entre sí, pero no se puede seleccionar simplemente un grupo de objetos y etiquetarlos como agregado. Debe empezar por 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 transaccionalmente coherentes son las que constituyen un agregado. La mejor manera de identificar agregados probablemente sea pensar en las operaciones de transacción.

El modelo de raíz agregada o entidad raíz

Un agregado se compone de al menos una entidad: la raíz agregada, que también se denomina entidad raíz o entidad principal. Además, puede tener varios objetos de valor y entidades secundarias, con todas las entidades y objetos trabajando de forma conjunta para implementar las transacciones y el comportamiento necesarios.

El propósito de una raíz agregada es asegurar la coherencia del agregado; debe ser el único punto de entrada para las actualizaciones del agregado a través de métodos u operaciones en la clase de raíz agregada. Los cambios en las entidades dentro del agregado solo se deben realizar a través de la raíz agregada. Se encarga de proteger la coherencia del agregado, teniendo en cuenta todas las invariables y reglas de coherencia que es posible que tenga que cumplir en el agregado. Si cambia una entidad secundaria o un objeto de valor por separado, la raíz agregada no podrá garantizar que el agregado esté en un estado válido. Sería como una mesa con una pata coja. El propósito principal de la raíz agregada es mantener la coherencia.

En la figura 7-9 se pueden ver agregados de ejemplo como el de Comprador, que contiene una sola entidad (la raíz agregada Comprador). El agregado de pedido contiene varias entidades y un objeto de valor.

Aggregate pattern

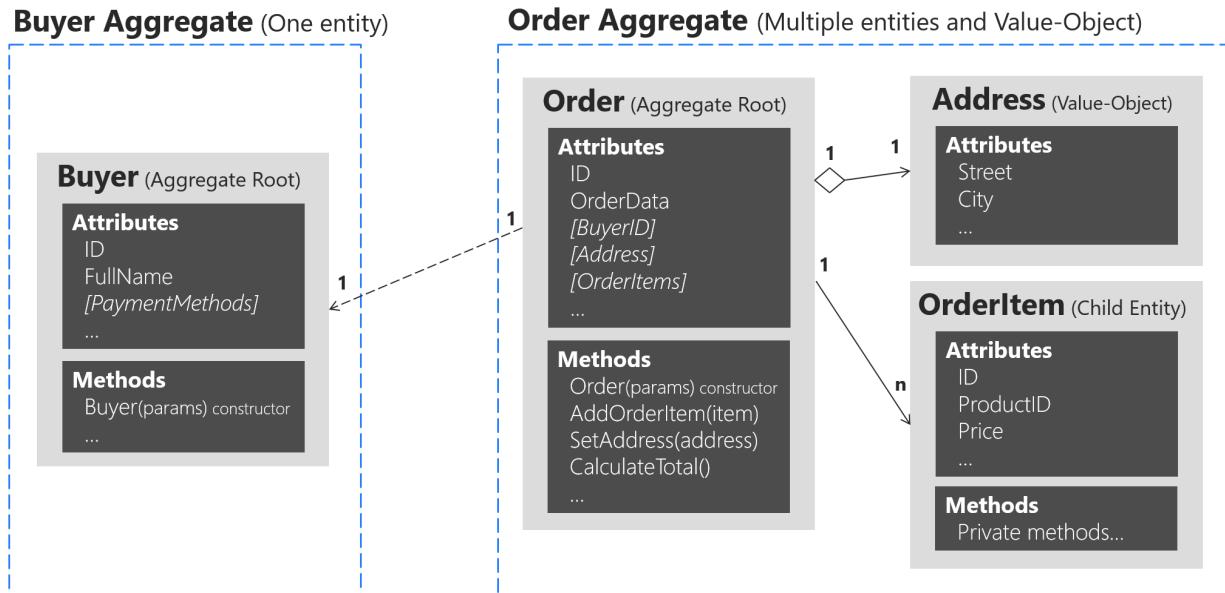


Figura 7-9. Ejemplo de agregados con una o varias entidades

Un modelo de dominio de un DDD se compone de agregados, un agregado puede tener una sola 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 ilustra un caso en el que el comprador tiene una única entidad, como un ejemplo de agregado que solo contiene una raíz agregada.

Con el fin de mantener la separación de agregados y límites claros entre ellos, un procedimiento recomendado en un modelo de dominio de DDD consiste en no permitir la navegación directa entre agregados y tener solo el campo de clave externa (FK), como se implementa en el [modelo de dominio de microservicio Ordering](#) 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:

```
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
}
```

Para identificar y trabajar con agregados se requiere investigación y experiencia. Para obtener más información, vea la lista siguiente de recursos adicionales.

Recursos adicionales

- **Vaughn Vernon. Diseño eficaz de agregados - Parte I: modelado de un único agregado** (de <http://dddcommunity.org/>)
http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf
- **Vaughn Vernon. Diseño eficaz de agregados - Parte II: hacer que los agregados funcionen de forma conjunta** (de <http://dddcommunity.org/>)
http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Vaughn Vernon. Diseño eficaz de agregados - Parte III: obtención de datos mediante la detección**

(de <http://dddcommunity.org/>)

http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_3.pdf

- **Sergey Grybniak. Patrones de diseño tácticos de diseño guiado por el dominio**

<https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part>

- **Chris Richardson. Desarrollo de microservicios transaccionales con agregados**

<https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson>

- **DevIQ. El patrón de agregado**

<https://deviq.com/aggregate-pattern/>

[ANTERIOR](#)

[SIGUIENTE](#)

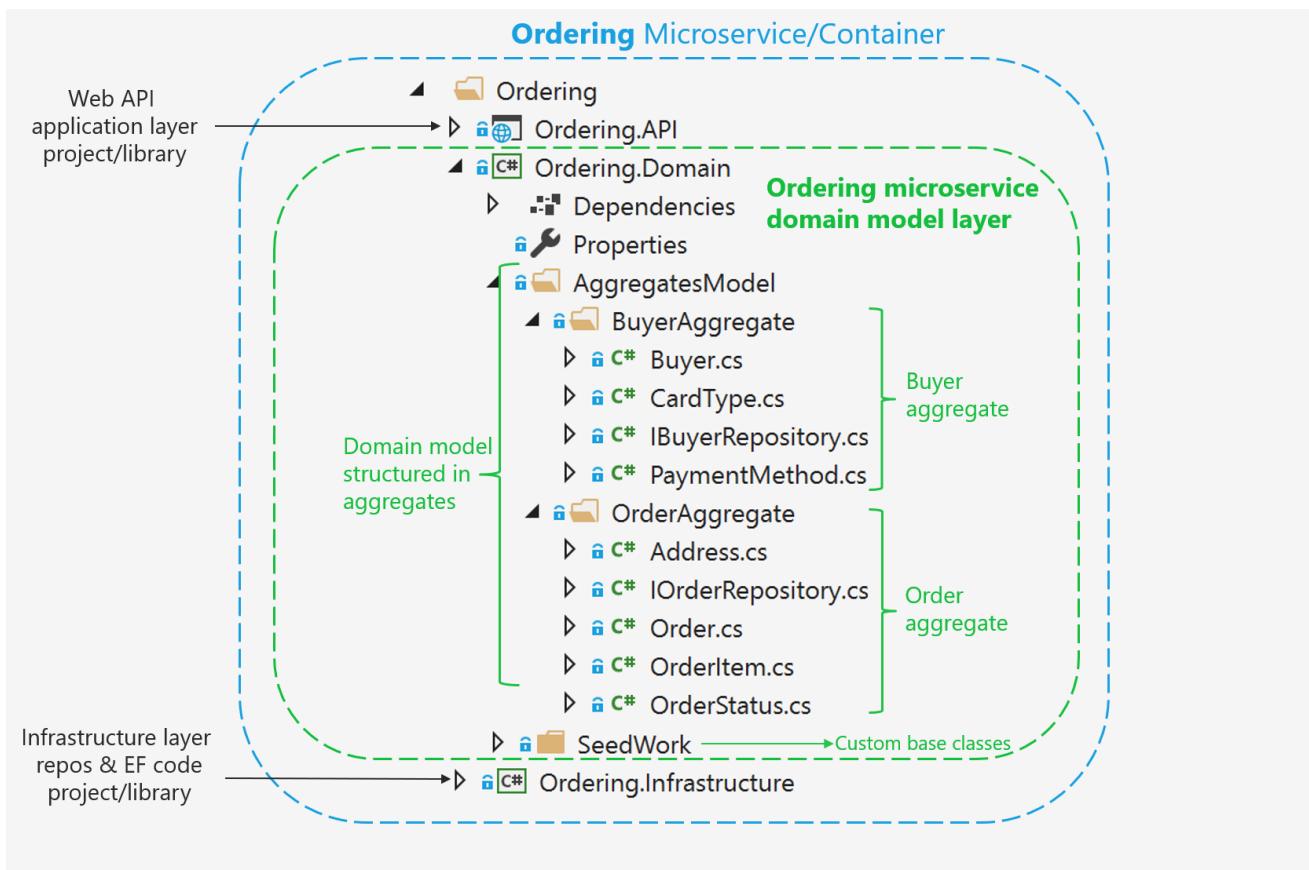
Implementación de un modelo de dominio de microservicio con .NET Core

13/01/2020 • 20 minutes to read • [Edit Online](#)

En la sección anterior se han explicado los principios y patrones de diseño fundamentales para diseñar un modelo de dominio. Ahora es el momento de analizar las posibles formas de implementar el modelo de dominio mediante .NET Core (código C# sin formato) y EF Core. Tenga en cuenta que el modelo de dominio se compone simplemente del código. Tiene solo los requisitos del modelo de EF Core, pero no las dependencias reales en EF. En el modelo de dominio no debe haber dependencias fuertes ni referencias a EF Core ni ningún otro ORM.

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 descubra que otra organización de carpetas comunica con mayor claridad las opciones de diseño elegidas para la aplicación. Como puede ver en la figura 7-10, en el modelo de dominio Ordering hay dos agregados, el agregado Order y el agregado Buyer. Cada agregado es un grupo de entidades de dominio y objetos de valor, aunque también podría tener un agregado compuesto por una sola entidad de dominio (la raíz de agregado o entidad raíz).



La vista Explorador de soluciones para el proyecto Ordering.Domain, en la que se muestra la carpeta AggregatesModel que contiene las carpetas BuyerAggregate y OrderAggregate, cada una con sus clases de entidad, archivos de objeto de valor y así sucesivamente.

Figura 7-10. Estructura del modelo de dominio del microservicio Ordering de eShopOnContainers

Además, la capa de modelo de dominio incluye los contratos de repositorio (interfaces) que son los requisitos de

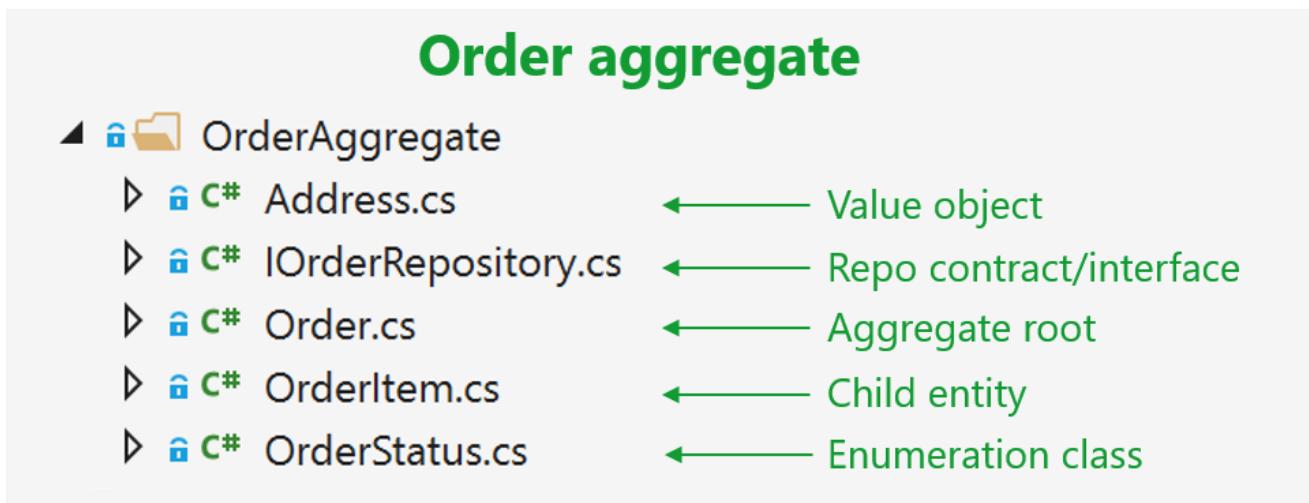
infraestructura del modelo de dominio. Es decir, estas interfaces expresan qué repositorios y métodos debe implementar la capa de infraestructura. Es fundamental que la implementación de los repositorios se coloque fuera de la capa de modelo de dominio, en la biblioteca de capas de infraestructura, para que la capa de modelo de dominio no quede "contaminada" por API o clases de tecnologías de infraestructura, como Entity Framework.

También puede ver una carpeta [SeedWork](#) que contiene clases base personalizadas que se pueden usar como base para las entidades de dominio y los objetos de valor, para no tener 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 aproximarse a la coherencia transaccional. Esos objetos pueden ser instancias de entidades (una de las cuales es la raíz de agregado o entidad raíz) más los objetos de valor adicionales.

La coherencia transaccional significa que se garantiza la coherencia y actualización de un agregado al final de una acción empresarial. Por ejemplo, la composición del agregado Order del modelo de dominio del microservicio Ordering de eShopOnContainers es la que se muestra en la figura 7-11.



Una vista detallada de la carpeta OrderAggregate: Address.cs es un objeto de valor, IOrderRepository es una interfaz de repositorio, Order.cs es una raíz agregada, OrderItem.cs es una entidad secundaria y OrderStatus.cs es una clase de enumeración.

Figura 7-11. Agregado Order en la solución de Visual Studio

Si abre cualquiera de los archivos de una carpeta de agregado, puede ver que está marcado como clase base personalizada o interfaz, como entidad u objeto de valor, tal como se ha implementado en la carpeta [SeedWork](#).

Implementación de entidades de dominio como clases POCO

En .NET, los modelos de dominio se implementan mediante la creación de clases POCO que implementan 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 código común relacionado con entidades. Tenga en cuenta que estas clases base e interfaces las define el usuario en el proyecto de modelo de dominio, por lo que es el código, no el código de infraestructura de un ORM, como EF.

```

// COMPATIBLE WITH ENTITY FRAMEWORK CORE 2.0
// 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 clase POCO. No tiene ninguna dependencia directa con Entity Framework Core ni ningún otro marco de trabajo de infraestructura. Esta implementación es como debería ser en DDD, simplemente código C# que implementa un modelo de dominio.

Además, la clase se decora con una interfaz denominada `IAggregateRoot`. Esta interfaz es una interfaz vacía, que a veces se denomina *interfaz de marcador*, que se usa simplemente para indicar que esta clase de entidad también es una raíz de agregado.

Una interfaz de marcador a veces se considera un anti-patrón; pero también es una manera eficaz de marcar una clase, sobre todo 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` sobre la clase. En cualquier caso, es una cuestión de preferencias.

Tener una raíz de agregado significa que la mayoría del código relacionado con la coherencia y las reglas de negocio de las entidades del agregado deben implementarse 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 ni directa; la clase AggregateRoot debe mantener el control y la coherencia de cualquier operación de actualización en sus entidades secundarias.

Encapsulación de datos en entidades de dominio

Un problema habitual de los modelos de entidad es que exponen propiedades de navegación de colecciones como tipos de lista públicamente accesibles. Esto permite que cualquier desarrollador colaborador manipule el contenido de estos tipos de colecciones, con lo que se pueden omitir importantes reglas de negocio relacionadas con la colección, lo que podría dejar el objeto en un estado no válido. La solución es conceder acceso de solo lectura a las colecciones relacionadas y proporcionar explícitamente métodos que definan formas para que los clientes las manipulen.

En el código anterior, observe que muchos atributos son de solo lectura o privados, y que solo pueden actualizarlos los métodos de clase, por lo que cualquier actualización tiene en cuenta las invariables de dominio de negocio de cuenta y la lógica especificada en los métodos de clase.

Por ejemplo, de acuerdo a los patrones DDD, **no debería hacer lo siguiente** desde ningún método de controlador de comando ni clase de capa de aplicación (de hecho debería ser imposible hacerlo):

```
// 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, las reglas o las validaciones del dominio relacionadas con esa operación con las entidades secundarias se distribuirá a la capa de aplicación (controladores de comandos y controladores de Web API).

Si omite la raíz de agregado, esta no puede garantizar sus invariables, su validez ni su coherencia. Al final tendrá código espagueti o 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 realizarse mediante métodos explícitos con lenguaje ubicuo explícito sobre el cambio que están realizando en la entidad.

Además, las colecciones de la entidad (por ejemplo, OrderItems) 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 fragmento de código siguiente muestra la manera adecuada de programar la tarea de agregar un objeto OrderItem al agregado Order.

```

// 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 la lógica relacionadas con la creación de un objeto OrderItem están bajo el control de la raíz de agregado Order, en el método AddOrderItem, especialmente las validaciones y la lógica relacionadas con otros elementos del agregado. Por ejemplo, podría obtener el mismo artículo como resultado de varias llamadas a AddOrderItem. En ese método, puede examinar los artículos y consolidar los mismos en un único objeto OrderItem con varias unidades. Además, si hay importes de descuento distintos pero el identificador de producto es el mismo, se aplicaría el mayor descuento. Este principio se aplica a cualquier otra lógica de dominio del 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 fin último del patrón de raíz de agregado.

Cuando use Entity Framework Core 1.1 o posterior, una entidad DDD se puede expresar mejor porque permite [asignar a campos](#) además de a 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 de la colección de campos en los métodos públicos y proporcionar acceso de solo lectura mediante el método AsReadOnly.

En DDD se prefiere actualizar la entidad únicamente mediante métodos de la entidad (o el constructor) para controlar cualquier invariable y la coherencia de los datos, para que las propiedades solo se definan con un descriptor de acceso get. Las propiedades se basan en campos privados. A los miembros privados solo se puede acceder desde la clase. Pero hay una excepción: EF Core también debe establecer estos campos (de forma que pueda 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 la tabla de base de datos no es responsabilidad del dominio, sino que forma parte de la capa de infraestructura y persistencia. Se menciona aquí simplemente para que sea consciente de las nuevas capacidades de EF Core 1.1 o posterior relacionadas con la forma de modelar entidades. En la sección de infraestructura y persistencia se explican más detalles 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.

Asignación de campos sin propiedades

La característica de EF Core 1.1 o posterior para asignar columnas a campos también permite no usar propiedades. En su lugar, puede simplemente asignar columnas de una tabla a campos. Un caso de uso común de esto son los campos privados de un estado interno al que no es necesario acceder desde fuera de la entidad.

Por ejemplo, en el ejemplo de código OrderAggregate anterior, hay varios campos privados, como el campo `_paymentMethodId`, sin ninguna propiedad relacionada para un establecedor ni un captador. Ese campo también podría calcularse en la lógica de negocios de Order y usarse desde los métodos de Order, pero debe conservarse además en la base de datos. Así, en EF Core (a partir de la versión 1.1) hay una forma de asignar un campo sin ninguna 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.** **Modeling Aggregates with DDD and Entity Framework (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://docs.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**
<http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>

[ANTERIOR](#)

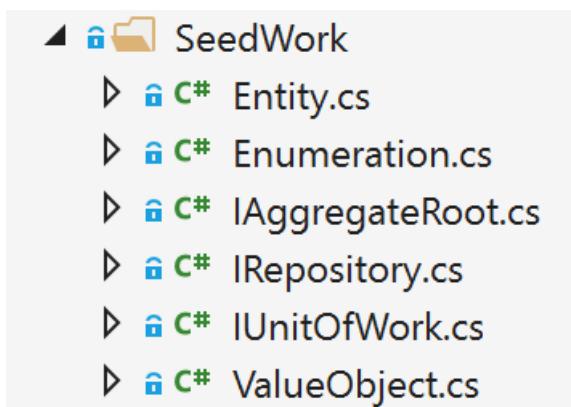
[SIGUIENTE](#)

Seedwork (interfaces y clases base reutilizables para su modelo de dominio)

13/01/2020 • 6 minutes to read • [Edit Online](#)

La carpeta de soluciones contiene una carpeta *SeedWork*. Esta carpeta contiene las clases base personalizadas que puede usar como base de los objetos de valor y las entidades de dominio. Use estas clases base para no tener código redundante en la clase de objeto de cada dominio. La carpeta para estos tipos de clases se denomina *SeedWork* y no nombres parecidos como *Marco*. Se llama *SeedWork* porque la carpeta contiene solo un pequeño subconjunto de clases reutilizables que realmente no se puede considerar un marco. *Seedwork* es un término introducido por [Michael Feathers](#) y popularizado por [Martin Fowler](#), pero esta carpeta también se puede denominar Common, SharedKernel o similar.

En la Figura 7-12 se muestran las clases que forman el seedwork 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 de lo que requiere implementación. Estas interfaces también se usan mediante la inserción de dependencias del nivel de aplicación.



Contenido detallado de la carpeta SeedWork, que contiene interfaces y clases base: Entity.cs, Enumeration.cs, IAggregateRoot.cs, IRepository.cs, IUnitOfWork.cs y ValueObject.cs.

Figura 7-12. Un conjunto de muestra de interfaces y clases base "seedwork" del modelo de dominio

Este es el tipo de reutilización de copiar y pegar que muchos desarrolladores comparten entre proyectos, y no un marco formal. Puede tener seedworks en cualquier nivel o biblioteca. Pero si el conjunto de clases e interfaces se hace lo suficientemente grande, puede crear una sola biblioteca de clases.

La clase base de entidad personalizada

El código siguiente es un ejemplo de clase base Entity en la que puede colocar código que puede ser utilizado de la misma forma por cualquier entidad de dominio, como el id. de entidad, [operadores de igualdad](#), una lista de eventos de dominio por entidad, etc.

```
// 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;
        }
        set
        {
            if (_requestedHashCode != value.GetHashCode())
            {
                _requestedHashCode = value.GetHashCode();
                _domainEvents.Add(Notification.Create("Entity.Id", value));
            }
            _Id = value;
        }
    }
}
```

```

    {
        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://blogs.microsoft.com/ericlippert/2011/02/28/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);
}
}

```

Este código, en el que se utiliza una lista de eventos de dominio por entidad, se describirá en las secciones

siguientes, al hablar de los eventos de dominio.

Contratos de repositorio (interfaces) en el nivel de modelo de dominio

Los contratos de repositorio no son más que interfaces .NET que expresan los requisitos de contrato de los repositorios que se van a utilizar en cada agregado.

Los repositorios en sí, con el código básico de EF Core o cualquier otra dependencia de infraestructura y código (Linq, SQL, etc.), no se deben implementar en el modelo de dominio; los repositorios solo deben implementar las interfaces que defina en el modelo de dominio.

Otro patrón relacionado con esta práctica (que coloca interfaces de repositorio en el nivel de modelo de dominio) es el patrón de interfaz separada. Como [explica](#) Martin Fowler, "utilice una interfaz separada para definir una interfaz en un paquete e implementarla en otro. De esta forma, un cliente que necesite la dependencia en la interfaz puede no tener en cuenta para nada la implementación".

Seguir el patrón de interfaz separada permite que el nivel de aplicación (en este caso, el proyecto API web para el microservicio) tenga una dependencia en los requisitos definidos en el modelo de dominio, pero no una dependencia directa en el nivel de infraestructura/persistencia. Además, puede usar la inserción de dependencias para aislar la implementación, que se implementa en el nivel de infraestructura/persistencia utilizando repositorios.

Por ejemplo, el siguiente ejemplo con la interfaz de `IOrderRepository` define las operaciones que la clase `OrderRepository` tendrá que implementar en el nivel de infraestructura. En la implementación actual de la aplicación, el código solo necesita agregar o actualizar los pedidos en la base de datos, puesto que las consultas se dividen siguiendo el enfoque de CQRS simplificado.

```
// 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. Separated Interface** (Interfaz independiente).
<https://www.martinfowler.com/eaaCatalog/separatedInterface.html>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación de objetos de valor

25/11/2019 • 21 minutes to read • [Edit Online](#)

Como se describe en secciones anteriores sobre las entidades y agregados, la identidad es esencial para las entidades, pero hay muchos objetos y elementos de datos en un sistema que no requieren ninguna identidad ni ningún seguimiento de identidad, como los objetos de valor.

Un objeto de valor puede hacer referencia a otras entidades. Por ejemplo, en una aplicación que genera una ruta que describe cómo ir 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 Ciudad, Carretera, etc.

En la figura 7-13 se muestra el objeto de valor Address en el agregado Order.

Value Object within Aggregate

Order Aggregate (Multiple entities and Value-Object)

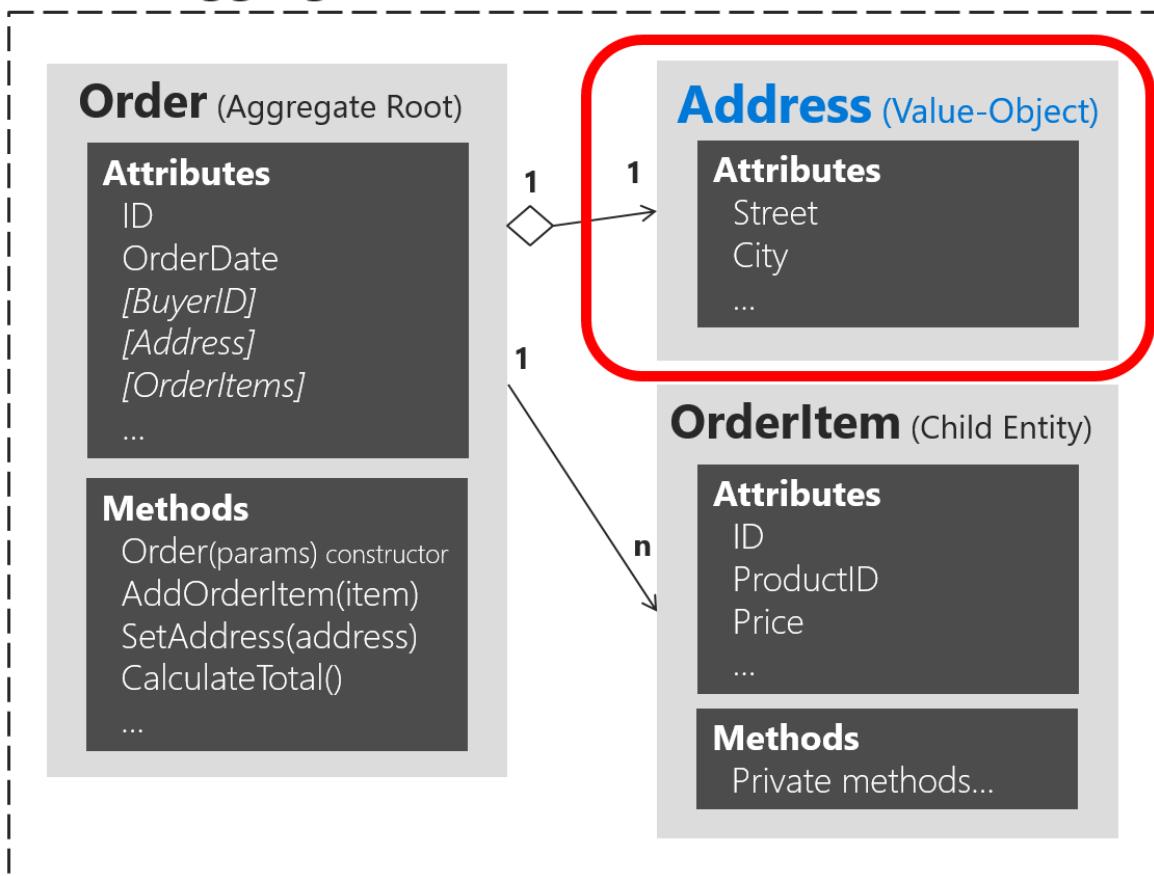


Figura 7-13. Objeto de valor Dirección en el agregado Pedido

Como se muestra en la figura 7-13, una entidad suele constar de varios atributos. Por ejemplo, la entidad `Order` se puede modelar como una entidad con una identidad y puede estar formada internamente por un conjunto de atributos, como `OrderId`, `OrderDate`, `OrderItems`, etc. Pero la dirección, que es un valor complejo formado por el país o región, la calle, la ciudad, etc., y que no tiene ninguna identidad en este dominio, se debe modelar y tratar

como un objeto de valor.

Características importantes de los objetos de valor

Hay dos características principales en los objetos de valor:

- No tienen ninguna identidad.
- Son inmutables.

La primera característica ya se ha mencionado. 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 permitir que cambien durante la vigencia del objeto.

Los objetos de valor le permiten hacer algunos trucos de rendimiento gracias a su naturaleza inmutable. Esto es así sobre todo en los sistemas en los que puede haber miles de instancias de objetos de valor, muchas de las cuales tienen los mismos valores. Su naturaleza inmutable permite que se puedan reutilizar y pueden ser objetos intercambiables, ya que sus valores son los mismos y no tienen ninguna identidad. Este tipo de optimización a veces puede suponer una diferencia entre el software que se ejecuta con lentitud y el software que tiene un buen rendimiento. Como es lógico, todos estos casos dependen el entorno de aplicación y del contexto de implementación.

Implementación de objetos de valor en C#

En cuanto a la implementación, puede tener una clase base de objeto de valor que tenga métodos de utilidad básicos, como la igualdad según la comparación entre todos los atributos (ya que un objeto de valor no se debe basar en la identidad) y otras características esenciales. En el ejemplo siguiente se muestra una clase base de objeto de valor que se usa en el microservicio de ordenación de eShopOnContainers.

```

public abstract class ValueObject
{
    protected static bool EqualOperator(ValueObject left, ValueObject right)
    {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
        {
            return false;
        }
        return ReferenceEquals(left, null) || left.Equals(right);
    }

    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }

    protected abstract IEnumerable<object> GetAtomicValues();

    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
        }

        ValueObject other = (ValueObject)obj;
        Ienumerator<object> thisValues = GetAtomicValues().GetEnumerator();
        Ienumerator<object> otherValues = other.GetAtomicValues().GetEnumerator();
        while (thisValues.MoveNext() && otherValues.MoveNext())
        {
            if (ReferenceEquals(thisValues.Current, null) ^
                ReferenceEquals(otherValues.Current, null))
            {
                return false;
            }

            if (thisValues.Current != null &&
                !thisValues.Current.Equals(otherValues.Current))
            {
                return false;
            }
        }
        return !thisValues.MoveNext() && !otherValues.MoveNext();
    }

    public override int GetHashCode()
    {
        return GetAtomicValues()
            .Select(x => x != null ? x.GetHashCode() : 0)
            .Aggregate((x, y) => x ^ y);
    }
    // Other utility methods
}

```

Puede usar esta clase al implementar el objeto de valor real, al igual que con el objeto de valor Address (Dirección) que se muestra en el ejemplo siguiente:

```

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; }

    private 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> GetAtomicValues()
    {
        // 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;
    }
}

```

Puede ver cómo esta implementación de objeto de valor de Address no tiene ninguna identidad y, por tanto, ningún campo de identificador, ni en la clase Address ni tampoco en la clase ValueObject.

Hasta EF Core 2.0 no fue posible no tener ningún campo de identificador en una clase que se fuera a usar en Entity Framework, lo que ayuda a implementar mejor los objetos de valor sin identificador. Eso es precisamente la explicación de la sección siguiente.

Se podría argumentar que los objetos de valor, al ser inmutables, deben ser de solo lectura (es decir, propiedades get-only), y realmente es cierto. Pero los objetos de valor normalmente se serializan y deserializan para recorrer colas de mensajes. Asimismo, si fueran de solo lectura, el deserializador no podría asignar los valores, por lo que simplemente se dejan como un conjunto privado, lo cual ofrece un nivel de solo lectura suficiente para que resulte práctico.

Cómo conservar los objetos de valor en la base de datos con EF Core 2.0

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 (EF) Core, que suele tener como destino las entidades con identidad?

Contexto y enfoques anteriores con EF Core 1.1

Como contexto, una limitación al usar EF Core 1.0 y 1.1 era que no se podían utilizar [tipos complejos](#) tal y como se definen en EF 6.x en .NET Framework tradicional. Por lo tanto, si se usaba EF Core 1.0 o 1.1, era necesario almacenar el objeto de valor como una entidad de EF con un campo de identificador. Luego, para que se pareciera más a un objeto de valor sin ninguna identidad, se podía ocultar su identificador para dejar claro que la identidad de un objeto de valor no es importante en el modelo de dominio. Ese identificador se podía ocultar usando como [propiedad reemplazada](#). Puesto que esa configuración para ocultar el identificador en el modelo está establecida en el nivel de la infraestructura de EF, resultaría algo transparente para su modelo de dominio.

En la versión inicial de eShopOnContainers (.NET Core 1.1), el identificador oculto necesario para la

infraestructura de EF Core estaba implementado del siguiente modo en el nivel de DbContext, usando la API fluida en el proyecto de la infraestructura. Por lo tanto, el identificador quedaba oculto desde el punto de vista del modelo de dominio, pero seguía presente en la infraestructura.

```
// 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
}
```

Pero la persistencia de ese objeto de valor en la base de datos se efectuaba como una entidad normal en otra tabla.

Con EF Core 2.0 hay nuevos y mejores métodos para conservar los objetos de valor.

Conservar objetos de valor como tipos entidad de propiedad en EF Core 2.0

Aunque haya algunas lagunas entre el patrón de objeto de valor canónico en el DDD y el tipo de entidad de propiedad en EF Core, ahora mismo es la mejor manera de conservar objetos de valor con EF Core 2.0. Puede consultar las limitaciones al final de esta sección.

La función del tipo de entidad de propiedad se agregó a EF Core a partir de la versión 2.0.

Un tipo de entidad de propiedad permite asignar tipos que no tienen su propia identidad definida de forma explícita en el modelo de dominio y que se usan como propiedades (como los objetos de valor) en cualquiera de las entidades. Un tipo de entidad de propiedad comparte el mismo tipo CLR con otro tipo de entidad (es decir, solo es una clase convencional). La entidad que contiene la navegación definitoria es la entidad del propietario. 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, los tipos de propiedad tienen identidad, 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:

- La identidad del propietario
- La propiedad de navegación que los señala
- En el caso de las colecciones de tipos de propiedad, un componente independiente (todavía no se admite en EF Core 2.0, se hará en la versión 2.2).

Por ejemplo, en el modelo de dominio Ordering de eShopOnContainers, como parte de la entidad Order, el objeto de valor Address se implementa como un tipo de entidad de propiedad dentro de la entidad del propietario, 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 al modo en que se usan los tipos complejos en EF6 en el .NET Framework tradicional.

Es importante tener en cuenta que los tipos de propiedad nunca se detectan por convención en EF Core, por lo que se deben declarar explícitamente.

En eShopOnContainers, en OrderingContext.cs, dentro del método OnModelCreating(), se aplican varias configuraciones de infraestructura. Una de ellas está relacionada con la entidad Order.

```
// 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 está definida para la entidad Order:

```
// 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 método `orderConfiguration.OwnsOne(o => o.Address)` especifica que la propiedad `Address` es una entidad de propiedad del tipo `Order`.

De forma predeterminada, las convenciones de EF Core asignan a las columnas de base de datos de las propiedades del tipo de entidad de propiedad el nombre `EntityProperty_OwnedEntityProperty`. Por lo tanto, las propiedades internas de `Address` aparecerán en la tabla `Orders` con los nombres `Address_Street` y `Address_City` (y así sucesivamente para `State`, `Country` y `ZipCode`).

Puede anexar el método fluido `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:

```
orderConfiguration.OwnsOne(p => p.Address)
    .Property(p=>p.Street).HasColumnName("ShippingStreet");

orderConfiguration.OwnsOne(p => p.Address)
    .Property(p=>p.City).HasColumnName("ShippingCity");
```

Se puede encadenar el método `OwnsOne` en una asignación fluida. En el siguiente ejemplo hipotético, `OrderDetails` posee `BillingAddress` y `ShippingAddress`, que son tipos `Address`. Luego, `OrderDetails` es propiedad del tipo `Order`.

```

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; }
}

```

Más datos 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 de propiedad en nuestro modelo de metadatos es una composición del tipo de propietario, la propiedad de navegación y el tipo CLR del tipo de propiedad.
- 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 las entidades de propiedad:

- 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.
- Desde EF Core 2.1, se puede configurar con el atributo [Owned].

Limitaciones de las entidades de propiedad:

- No se puede crear un DbSet<T> de un tipo de propiedad (por diseño).
- No se puede llamar a modelBuilder.Entity<T>() en los tipos de propiedad (actualmente por cuestiones de diseño).
- Todavía no hay colecciones de tipos de propiedad (en EF Core 2.1, pero se admitirán en la versión 2.2).
- 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 de hoja de las mismas jerarquías de herencia como otros tipos de propiedad. EF Core no deducirá 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).
- Pueden hacer referencia a otras entidades (es decir, pueden actuar como el lado dependiente en las relaciones con otros tipos que no son de propiedad).

Recursos adicionales

- **Martin Fowler. El patrón ValueObject**

<https://martinfowler.com/bliki/ValueObject.html>

- **Eric Evans. Diseño orientado al dominio: abordar la complejidad en el corazón del software.**

(Libro; incluye una descripción de los objetos de valor)

<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

- **Vaughn Vernon. Implementing Domain-Driven Design** (Implementación del diseño guiado por el dominio). (Libro; incluye una descripción de los objetos de valor)

<https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/>

- **Propiedades reemplazadas**

<https://docs.microsoft.com/ef/core/modeling/shadow-properties>

- **Complex types and/or value objects** (Tipos u objetos de valor complejos). Descripción en el repositorio de GitHub de EF Core (pestaña Problemas)

<https://github.com/aspnet/EntityFramework/issues/246>

- **ValueObject.cs.** Clase base de objeto de valor en eShopOnContainers.

<https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/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>

ANTERIOR

SIGUIENTE

Uso de las clases de enumeración en lugar de los tipos de enumeración

04/11/2019 • 3 minutes to read • [Edit Online](#)

Las [enumeraciones](#) (o *tipos enum* abreviado) son un contenedor de lenguaje fino alrededor de un tipo entero. Es posible que quiera limitar su uso al momento en que almacena un valor de un conjunto cerrado de valores. La clasificación basada en tamaños (pequeño, mediano, grande) 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 da lugar 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 habilitan todas las características enriquecidas de un lenguaje orientado a objetos.

Sin embargo, esto no es un tema crítico y, en muchos casos, por simplicidad, puede seguir usando [tipos enum](#) normales si lo prefiere. En cualquier caso, el uso de las clases de enumeración está más relacionado con los conceptos de tipo empresarial.

Implementación de una clase base de enumeración

El microservicio de pedidos en eShopOnContainers proporciona una implementación de clase base de enumeración de ejemplo, como se muestra en el ejemplo siguiente:

```

public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }

    public override string ToString() => Name;

    public static IEnumerable<T> GetAll<T>() where T : Enumeration
    {
        var fields = typeof(T).GetFields(BindingFlags.Public |
                                         BindingFlags.Static |
                                         BindingFlags.DeclaredOnly);

        return fields.Select(f => f.GetValue(null)).Cast<T>();
    }

    public override bool Equals(object obj)
    {
        var otherValue = obj as Enumeration;

        if (otherValue == null)
            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 u objeto de valor, como ocurre con la clase `CardType` : `Enumeration` siguiente:

```

public class CardType : Enumeration
{
    public static CardType Amex = new CardType(1, "Amex");
    public static CardType Visa = new CardType(2, "Visa");
    public static CardType MasterCard = new CardType(3, "MasterCard");

    public CardType(int id, string name)
        : base(id, name)
    {
    }
}

```

Recursos adicionales

- **Enum's are evil—update (Las enumeraciones son contraproducentes: actualice)**
<https://www.planetgeek.ch/2009/07/01/enums-are-evil/>
- **Daniel Hardman. How Enums Spread Disease — And How To Cure It (Cómo las enumeraciones**

contagian la enfermedad y cómo curarla)

<https://codecraft.co/2012/10/29/how-enums-spread-disease-and-how-to-cure-it/>

- **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.** Ardalís - Classes to help produce strongly typed smarter enums in .NET. (Ardalís: clases para ayudar a crear enumeraciones fuertemente tipadas de manera más inteligente en .NET)

<https://www.nuget.org/packages/Ardalis.SmartEnum/>

[ANTERIOR](#)

[SIGUIENTE](#)

Diseño de validaciones en el nivel de modelo de dominio

23/10/2019 • 11 minutes to read • [Edit Online](#)

En el diseño guiado por el dominio (DDD), las reglas de validación se pueden considerar invariables. La responsabilidad principal de un agregado es aplicar invariables en todos los cambios de estado para todas las entidades 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 un elemento de pedido siempre debe tener una cantidad que debe constar de un entero positivo, un nombre de artículo y un precio. Por lo tanto, la aplicación de invariables es responsabilidad de las entidades de dominio (en especial de la raíz agregada) y un objeto de entidad no debería poder existir si no es válido. Las reglas invariables se expresan como contratos y, si se infringen, se generan excepciones o notificaciones.

El razonamiento es que se producen muchos errores porque los objetos tienen un estado que no deberían tener nunca. A continuación se muestra una buena explicación de Greg Young, publicada en un [debate en línea](#):

Ahora imaginémonos que tenemos un SendUserCreationEmailService que toma un UserProfile... ¿Cómo podemos justificar en ese servicio que Name no es nulo? ¿Lo volvemos a comprobar? O lo que es más probable: no se molesta en comprobarlo y "espera lo mejor" (espera que alguien se haya molestado en validarla antes de enviársela). Por supuesto, si usamos TDD, una de las primeras pruebas que deberíamos escribir es que si, al enviar un cliente con un nombre nulo, se generaría un error. Pero una vez que comenzamos a escribir estos tipos de pruebas una y otra vez nos damos cuenta de que... "Espera, si no hubiéramos dejado que Name fuera nulo, no tendríamos todas estas pruebas".

Implementación de validaciones en el nivel de modelo de dominio

Las validaciones se suelen implementar en constructores de entidad de dominio o en métodos que pueden actualizar la entidad. Hay varias maneras de implementar las validaciones, como la comprobación de los datos y la generación de 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 mientras se produce.

Validación de condiciones y generación de excepciones

En el siguiente ejemplo de código se muestra el método de validación más sencillo en una entidad de dominio mediante la generación de una excepción. En la tabla de referencias que encontrará al final de esta sección puede ver vínculos a implementaciones más avanzadas según los patrones que hemos analizado anteriormente.

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
```

Un ejemplo mejor demostraría la necesidad de garantizar que el estado interno no varió o que se produjeron todas las mutaciones de un método. Por ejemplo, la siguiente implementación dejaría el objeto en un estado no válido:

```
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, la primera línea de dirección y la ciudad ya se han cambiado, lo cual podría invalidar la dirección.

Se puede aplicar un enfoque similar en el constructor de la entidad, generando una excepción para garantizar que la entidad sea 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 MaxLength o Required, se pueden usar para configurar las propiedades de campo de base de datos de EF Core, como se explica en detalle en la sección [Asignación de tabla](#), pero [ya no funcionan para la validación de entidades en EF Core](#) (tampoco funciona el método `IValidatableObject.Validate`), tal y como lo 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 de modelos, antes de la invocación de acciones del controlador como de costumbre, pero ese modelo está pensado para ser un ViewModel o DTO, y eso ataña a MVC o la API, no al modelo de dominio.

Después de tener la diferencia conceptual clara, todavía puede usar anotaciones de datos y `IValidatableObject` en la clase de entidad para la validación, si las acciones reciben un parámetro de objeto de clase de entidad, lo que no se recomienda. En ese caso, la validación se producirá durante el enlace de modelos, justo antes de invocar la acción y puede comprobar la propiedad `ModelState.IsValid` del controlador para comprobar el resultado, pero nuevamente, ocurre en el controlador, no antes de guardar el objeto de entidad en el `DbContext`, tal como se había llevado a cabo desde EF 4.x.

Todavía puede implementar la validación personalizada en la clase de entidad con las anotaciones de datos y el método `IValidatableObject.Validate` invalidando el método `SaveChanges` de `DbContext`.

Puede ver un ejemplo de implementación de la validación de entidades de `IValidatableObject` en [este comentario en GitHub](#). Ese ejemplo no realiza validaciones basadas en atributos, pero deberían ser fáciles de implementar mediante la reflexión en el mismo reemplazo.

Pero desde la óptica del DDD, el modelo de dominio se ajusta mejor con el uso de excepciones en los métodos de comportamiento de la entidad o con la implementación de los patrones de especificación y notificación para aplicar reglas de validación.

Puede resultar lógico usar anotaciones de datos en el nivel de aplicación en las clases ViewModel (en lugar de hacerlo en las entidades de dominio) que aceptarán la entrada a fin de permitirlas para la validación del modelo en la capa de la interfaz de usuario, pero no se debería hacer en la exclusión de validación dentro del modelo de dominio.

Validación de entidades implementando el patrón de especificación y el patrón de notificación

Por último, un enfoque más elaborado para implementar validaciones en el modelo de dominio consiste en implementar el patrón de especificación junto con el patrón de notificación, como se explica en algunos de los recursos adicionales que se muestran más adelante.

Merece la pena mencionar que también se puede usar solo uno de estos patrones (por ejemplo, validándolo manualmente con instrucciones de control, pero usando el patrón de notificación para apilar y devolver una lista de errores de validación).

Uso de la validación diferida en el dominio

Hay varios métodos para tratar las validaciones diferidas en el dominio. En su libro [Implementing Domain-Driven Design](#) (Implementación del diseño guiado por el dominio), Vaughn Vernon habla de ellos en la sección sobre la validación.

Validación en dos pasos

Plantéese también usar la validación en dos pasos. Use la validación de nivel de campo en los objetos de transferencia de datos (DTO) de comandos y la validación de nivel de dominio dentro de las entidades. Para ello, puede devolver un objeto de resultado en vez de excepciones para que resulte más fácil tratar los errores de validación.

Si usa la validación de campos con anotaciones de datos, por ejemplo, no se duplica la definición de validación. Pero la ejecución puede ser del lado servidor y del lado cliente en el caso de los DTO (comandos y ViewModels, por ejemplo).

Recursos adicionales

- **Rachel Appel. Validación de modelos en ASP.NET Core MVC**
<https://docs.microsoft.com/aspnet/core/mvc/models/validation>
- **Rick Anderson. Agregar validación a una aplicación ASP.NET Core MVC**
<https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation>
- **Martin Fowler. Replacing Throwing Exceptions with Notification in Validations** \ (Reemplazo del inicio de excepciones por notificaciones en validaciones)
<https://martinfowler.com/articles/replaceThrowWithNotification.html>
- **Specification and Notification Patterns** \ (Patrones de especificación y notificación)
<https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns>
- **Lev Gorodinski. Validation in Domain-Driven Design (DDD)** \ (Validación en diseños guiados por dominio)
<http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/>
- **Colin Jack. Domain Model Validation** \ (Validación de modelos de dominio)
<https://colinjack.blogspot.com/2008/03/domain-model-validation.html>
- **Jimmy Bogard. Validation in a DDD world** \ (Validación en un mundo de diseños guiados por dominio)
<https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/>

[ANTERIOR](#)

[SIGUIENTE](#)

Validación del lado cliente (validación de los niveles de presentación)

23/10/2019 • 5 minutes to read • [Edit Online](#)

Incluso cuando el origen de verdad es el modelo del dominio y, en última instancia, debe tener validación en el nivel de modelo de dominio, la validación todavía puede controlarse en el nivel de modelo de dominio (servidor) y la IU (lado cliente).

La validación del lado cliente es una gran ventaja para los usuarios. Les permite ahorrar un tiempo que, de otro modo, pasarían esperando un viaje de ida y vuelta al servidor que podría devolver errores de validación. En términos comerciales, incluso unas pocas fracciones de segundos multiplicadas por cientos de veces al día suman una gran cantidad de tiempo, gastos y frustración. La validación inmediata y sencilla permite a los usuarios trabajar de forma más eficiente y generar una entrada y salida de datos de mayor 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 podrían ser similares, pero servir para un propósito diferente. Si le preocupa DRY (el principio de No repetirse), tenga en cuenta que en este caso la reutilización del código también puede indicar el acoplamiento y en las aplicaciones empresariales es más importante no acoplar el lado servidor al lado cliente que seguir el principio DRY.

Incluso cuando se usa la validación del lado cliente, siempre debe validar sus comandos o DTO de entrada en el código de servidor, ya que las API de servidor son un posible vector de ataque. Normalmente, la mejor opción es hacer ambas cosas porque, si tiene una aplicación cliente, desde la perspectiva de la experiencia del usuario es mejor anticiparse y no permitir que el usuario escriba información no válida.

Por tanto, normalmente se validan los ViewModels en el código del lado cliente. También puede validar los DTO o los 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 esté creando. Será diferente si valida los datos en una aplicación web MVC con la mayor parte del código en .NET, una aplicación web SPA en la que la validación se codifique en JavaScript o TypeScript, o bien una aplicación móvil codificada con Xamarin y C#.

Recursos adicionales

Validación en aplicaciones móviles de Xamarin

- **Validar la entrada de texto y mostrar errores**

https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/

- **Devolución de llamada de validación**

<https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/>

Validación en aplicaciones ASP.NET Core

- **Rick Anderson. Agregar validación a una aplicación ASP.NET Core MVC**

<https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation>

Validación en aplicaciones SPA Web (Angular 2, TypeScript, JavaScript)

- **Ado Kukic. Validación de formularios de Angular 2**

<https://scotch.io/tutorials/angular-2-form-validation>

- **Validación de formularios**

<https://angular.io/guide/form-validation>

- **Validation (Validación).** Documentación de Breeze.

<https://breeze.github.io/doc-js/validation.html>

En resumen, estos son los conceptos más importantes en lo que respecta a la validación:

- Las entidades y los agregados deben aplicar su propia coherencia y ser "siempre válidos". Las raíces agregadas son responsables de la coherencia de varias entidades dentro del mismo agregado.
- Si cree que una entidad debe entrar en un estado no válido, considere el uso de un modelo de objetos diferente: por ejemplo, usando 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 que todos ellos se han creado, considere la posibilidad de usar el patrón Factory.
- En la mayoría de los casos, tener validación redundante en el lado cliente es adecuado, porque la aplicación puede ser proactiva.

[ANTERIOR](#)

[SIGUIENTE](#)

Eventos de dominio: diseño e implementación

09/12/2019 • 42 minutes to read • [Edit Online](#)

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 asíncrona 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 asíncronos. Pero los eventos de integración siempre deben ser asíncronos.

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.

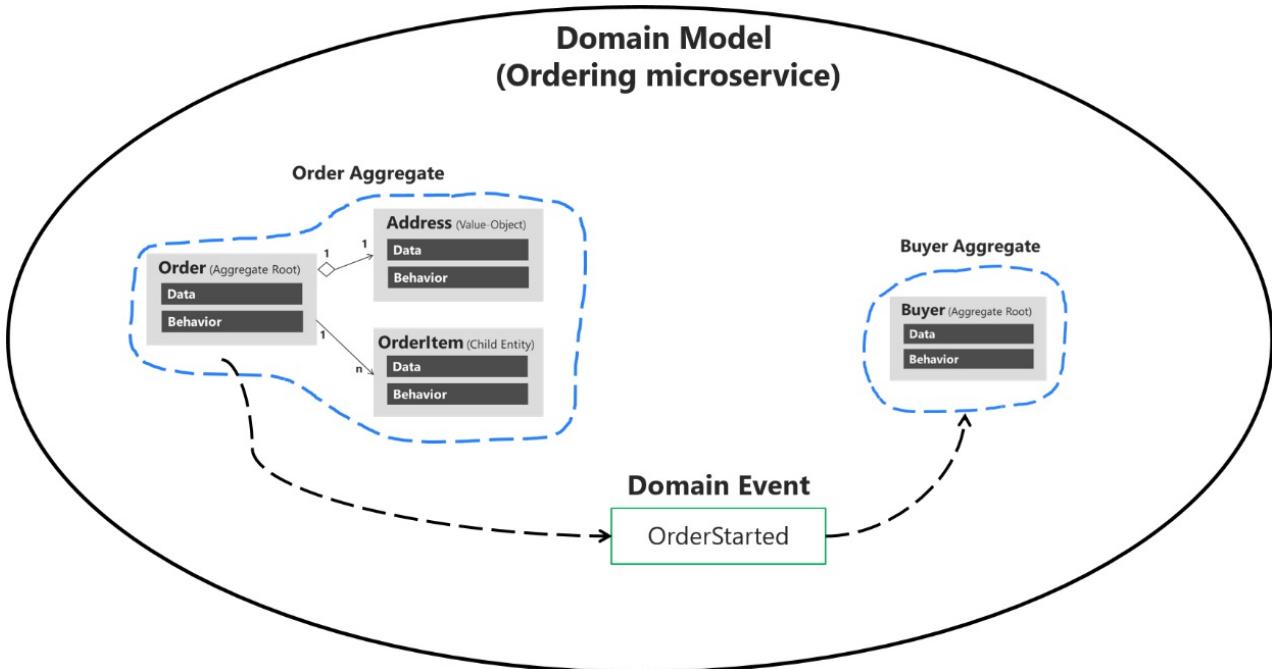


Figura 7-14. Eventos de dominio para exigir la coherencia entre varios agregados dentro del mismo dominio

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 acopla 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.

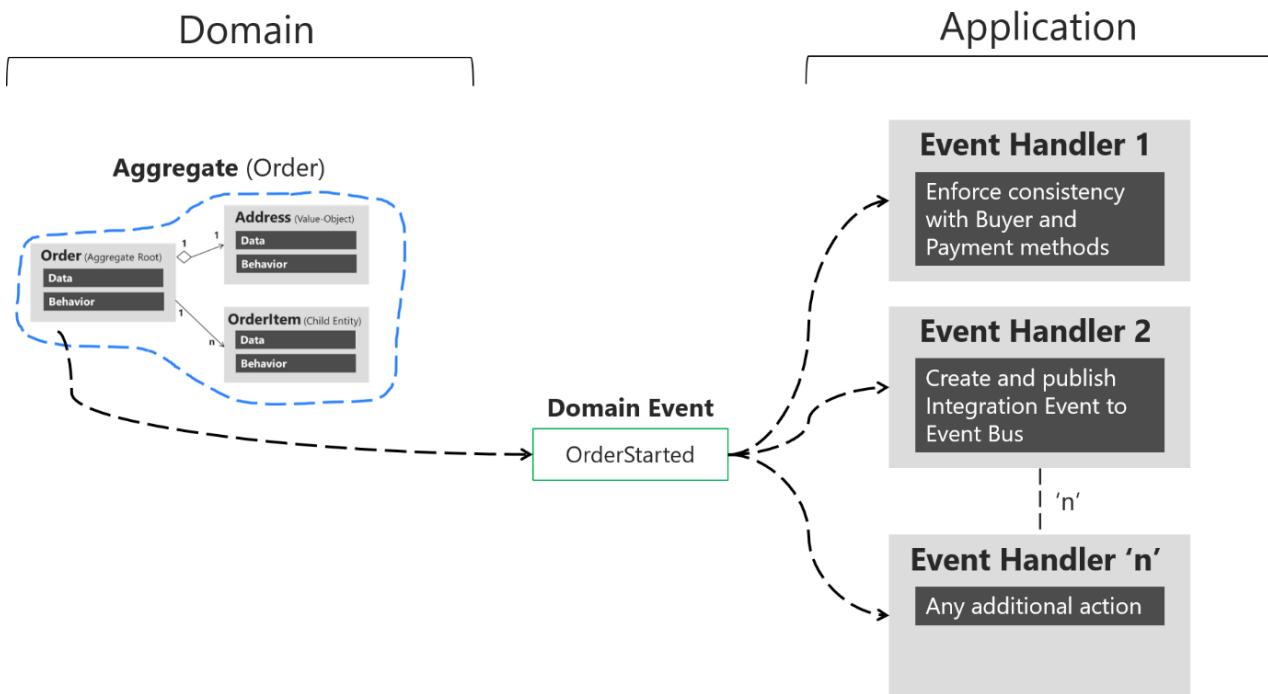


Figura 7-15. Control de varias acciones por dominio

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 el nivel 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 de eventos 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:

```

public class OrderStartedDomainEvent : INotification
{
    public string UserId { 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,
        int cardTypeId, string cardNumber,
        string cardSecurityNumber, string cardHolderName,
        DateTime cardExpiration)
    {
        Order = order;
        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 debe 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 se llama, 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, le interesa centrarse únicamente en lo que sucede en las clases agregadas actuales; no le interesa que repentinamente se le redirija a otros controladores de eventos para los efectos secundarios relacionados con otros agregados o la lógica de la 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 *justo después* 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`:

```
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:

```
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:

```

// 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, como cuando se usa una base de datos NoSQL como Azure CosmosDB, se deben 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 debe ser mayor o no se ha diseñado 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, me interesa 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 la duración de DbContext se define de forma predeterminada como "en ámbito". Por tanto, el objeto DbContext se comparte entre varios objetos de repositorio de los 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 distribuyan, 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 empresariales. 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).

Otra manera de asignar eventos a varios controladores de eventos consiste en usar el registro de tipos en un contenedor de IoC para poder inferir de forma dinámica a dónde enviar los eventos. En otras palabras, debe saber qué controladores de eventos tienen que obtener un evento específico. En la figura 7-16 se muestra un enfoque simplificado para esto.

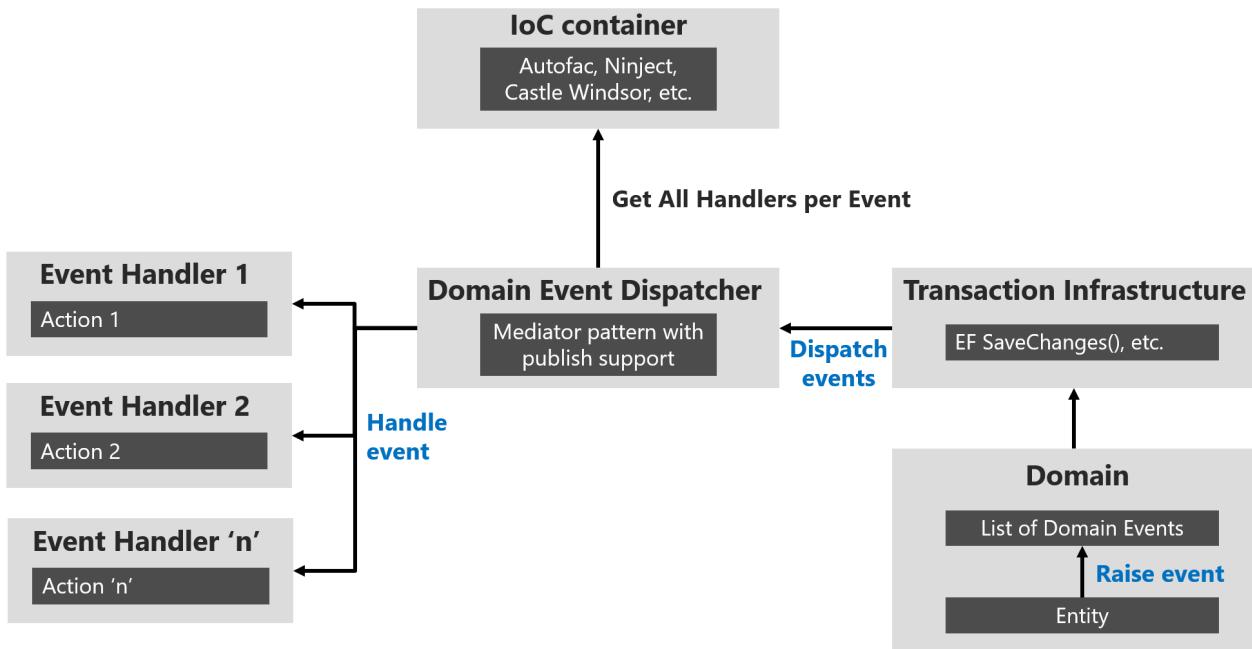


Figura 7-16. Distribuidor de eventos de dominio con IoC

Puede crear usted mismo todos los artefactos para implementar este enfoque. Pero también puede usar bibliotecas disponibles como [MediatR](#), que interiormente usa el contenedor de IoC. Por tanto, puede usar directamente las interfaces predefinidas y los métodos de publicación y distribución del objeto de mediador.

En el código, primero debe registrar los tipos de controlador de eventos en el contenedor de IoC, como se muestra en el ejemplo siguiente del [microservicio de pedidos de eShopOnContainers](#):

```

public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        // Other registrations ...
        // Register the DomainEventHandler classes (they implement IAsyncNotificationHandler<>)
        // in assembly holding the Domain Events
        builder.RegisterAssemblyTypes(typeof(ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler)
            .GetTypeInfo().Assembly)
            .AsClosedTypesOf(typeof(IAsyncNotificationHandler<>));
        // Other registrations ...
    }
}

```

En primer lugar, el código identifica el ensamblado que contiene los controladores de eventos de dominio localizando el ensamblado que contiene cualquiera de los controladores (mediante `typeof(ValidateOrAddBuyerAggregateWhenXxxx)`, pero podría haber elegido cualquier otro controlador de eventos para buscar el ensamblado). Como todos los controladores de eventos implementan la interfaz `IAsyncNotificationHandler`, el código solo busca esos tipos y registra todos los controladores de eventos.

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:

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : IAsyncNotificationHandler<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.

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : INotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILoggerFactory _logger;
    private readonly IBuyerRepository<Buyer> _buyerRepository;
    private readonly IIdentityService _identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
        ILoggerFactory logger,
        IBuyerRepository<Buyer> buyerRepository,
        IIdentityService identityService)
    {
        // ...Parameter validations...
    }

    public async Task Handle(OrderStartedDomainEvent orderStartedEvent)
    {
        var cardTypeId = (orderStartedEvent.CardTypeId != 0) ? orderStartedEvent.CardTypeId : 1;
        var userGuid = _identityService.GetUserIdentity();
        var buyer = await _buyerRepository.FindAsync(userGuid);
        bool buyerOriginallyExisted = (buyer == null) ? false : true;

        if (!buyerOriginallyExisted)
        {
            buyer = new Buyer(userGuid);
        }

        buyer.VerifyOrAddPaymentMethod(cardTypeId,
            $"Payment Method on {DateTime.UtcNow}",
            orderStartedEvent.CardNumber,
            orderStartedEvent.CardSecurityNumber,
            orderStartedEvent.CardHolderName,
            orderStartedEvent.CardExpiration,
            orderStartedEvent.Order.Id);

        var buyerUpdated = buyerOriginallyExisted ? _buyerRepository.Update(buyer)
            : _buyerRepository.Add(buyer);

        await _buyerRepository.UnitOfWork
            .SaveEntitiesAsync();

        // Logging code using buyerUpdated info, etc.
    }
}
```

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://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf#page=25
- **Jan Stenberg. 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** (Vaughn Vernon. Diseño eficaz de agregados - Parte II: hacer que los agregados funcionen de forma conjunta)
https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Jimmy Bogard. Strengthening your domain: Domain Events** (Jimmy Bogard. Reforzar el dominio: eventos de dominio)
<https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/>
- **Tony Truong. Domain Events Pattern Example** (Ejemplo de patrón de eventos de dominio)
<https://www.tonytruong.net/domain-events-pattern-example/>
- **Udi Dahan. How to create fully encapsulated Domain Models** (Cómo crear modelos de dominio totalmente encapsulados)
<http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>
- **Udi Dahan. Domain Events – Take 2** (Eventos de dominio: parte 2)
<http://udidahan.com/2008/08/25/domain-events-take-2/>
- **Udi Dahan. Domain Events – Salvation** (Eventos de dominio: salvación)
<http://udidahan.com/2009/06/14/domain-events-salvation/>
- **Jan Kronquist. Don't publish Domain Events, return them!** (No publique eventos de dominio, devuélvalos)
<https://blog.jayway.com/2013/06/20/dont-publish-domain-events-return-them/>
- **Cesar de la Torre. Domain Events vs. Integration Events in DDD and microservices architectures**

(Eventos de integración en DDD y arquitecturas de microservicios)

<https://devblogs.microsoft.com/cesardelatorre/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/>

[ANTERIOR](#)

[SIGUIENTE](#)

Diseño del nivel de persistencia de infraestructura

25/11/2019 • 18 minutes to read • [Edit Online](#)

Los componentes de persistencia de datos proporcionan acceso a los datos que se hospedan 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 de [unidad de trabajo](#), como los objetos `DbContext` de Entity Framework (EF). `DbContext` de EF implementa ambos, el repositorio y los patrones de unidad de trabajo.

El modelo de repositorio

Los repositorios son clases o componentes que encapsulan la lógica necesaria para tener acceso a orígenes de datos. Centralizan la funcionalidad de acceso a datos comunes, lo que proporciona un mejor mantenimiento y el desacoplamiento de la infraestructura o tecnología que se usa para acceder a bases de datos desde el nivel de 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 permite centrarse en la lógica de persistencia de datos en lugar del establecimiento del acceso a los datos.

El modelo de repositorio es una manera bien documentada de trabajar con un origen de datos. En el libro [Patterns of Enterprise Application Architecture](#) (Patrones de arquitectura de aplicaciones empresariales), Martin Fowler describe un repositorio de esta forma:

Un repositorio realiza las tareas de un intermediario entre los niveles de modelo de dominio y asignación de datos, actuando de forma similar a un conjunto de objetos de dominio en memoria. Los objetos de cliente generan consultas mediante declaraciones y las envían a los repositorios para obtener las respuestas. Conceptualmente, un repositorio encapsula un conjunto de objetos almacenados en la base de datos y las operaciones que se pueden realizar en ellos, proporcionando una manera de que esté más cerca de la capa de persistencia. Además, los repositorios admiten la finalidad de separar, con claridad y en una dirección, la dependencia entre el dominio de trabajo y la asignación de datos.

Definir un repositorio por agregado

Para cada agregado o raíz agregada, se debe crear una clase de repositorio. En un microservicio basado en patrones de diseño controlado por dominios (DDD), el único canal que se debe usar para actualizar la base de datos deben ser los repositorios. Esto se debe a que tienen una relación uno a uno con la raíz agregada, que controla las invariables del agregado y la coherencia transaccional. Es correcto consultar la base de datos a través de otros canales (como con un enfoque CQRS), dado que las consultas no cambian el estado de la base de datos. Pero el área transaccional (es decir, las actualizaciones) siempre se debe controlar mediante los repositorios y las raíces agregadas.

Básicamente, un repositorio permite llenar los datos en memoria que proceden de la base de datos en forma de entidades de dominio. Una vez que las entidades se encuentran en memoria, se pueden cambiar y después volver a conservar en la base de datos a través de transacciones.

Como se indicó anteriormente, si se usa el modelo de arquitectura de CQS/CQRS, las consultas iniciales se realizan por medio de consultas paralelas fuera del modelo de dominio, ejecutadas por instrucciones SQL simples mediante Dapper. Este enfoque es mucho más flexible que los repositorios, ya que se pueden consultar y combinar las tablas que se necesitan, y estas consultas no están limitadas por las 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 al nivel de la aplicación (por ejemplo, un servicio de API web). Cuando se recibe un comando en un

controlador de comandos, se usan repositorios para obtener los datos que se quieren actualizar desde la base de datos. Se actualiza en memoria con los datos que se pasa con los comandos y después se agregan o actualizan los datos (entidades de dominio) en la base de datos a través de una transacción.

Es importante destacar de nuevo que solo se debe definir un repositorio para cada raíz agregada, como se muestra en la figura 7-17. Para lograr el objetivo de la raíz agregada de mantener la coherencia transaccional entre todos los objetos del agregado, nunca se debe crear un repositorio para cada tabla de la base de datos.

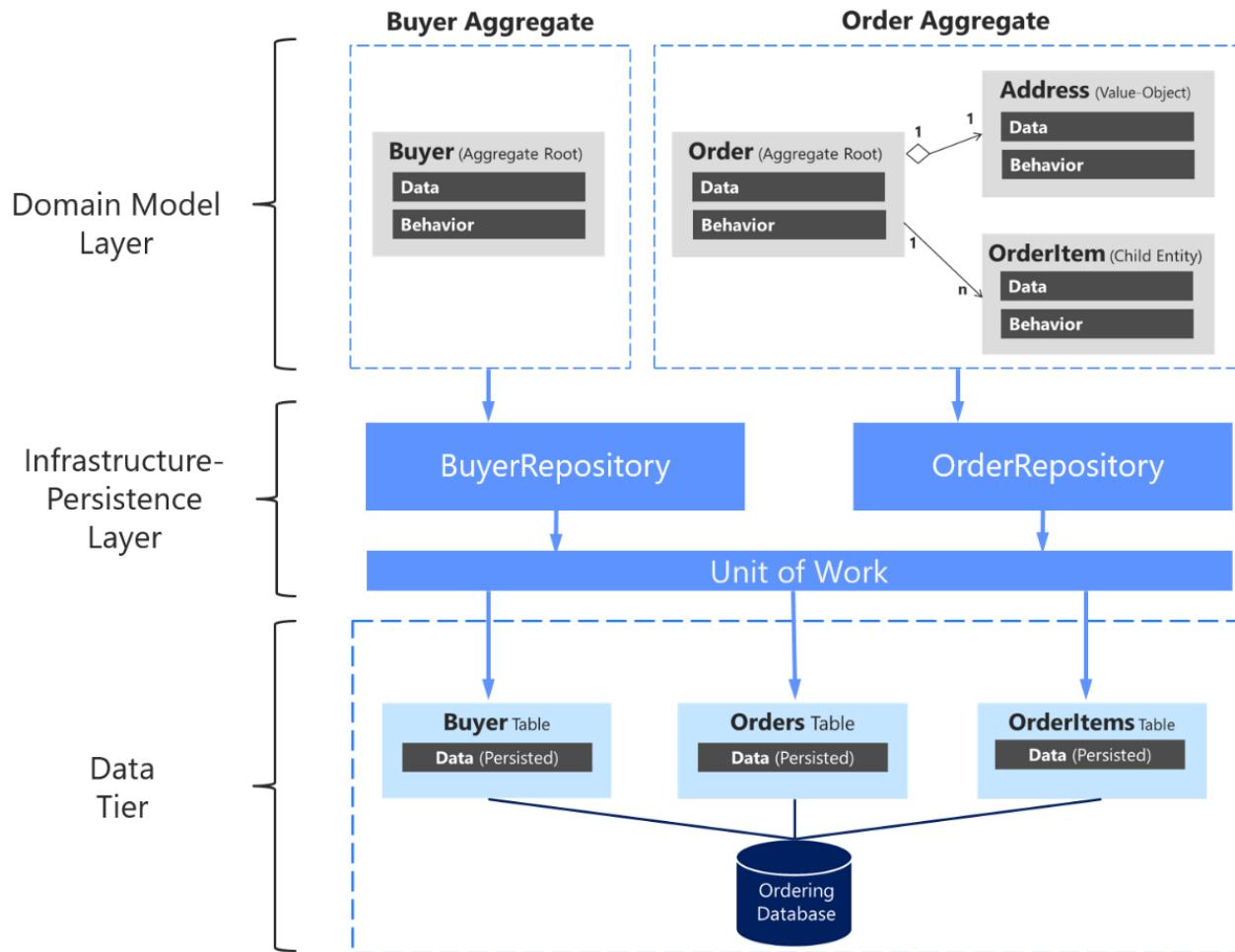


Figura 7-17. La relación entre repositorios, agregados y tablas de base de datos

En el diagrama anterior se muestran las relaciones entre las capas de dominio e infraestructura: el agregado Comprador depende del IBuyerRepository y el agregado Pedido depende de las interfaces de IOrderRepository, estas interfaces se implementan en el nivel de infraestructura por los repositorios correspondientes que dependen de UnitOfWork, también implementada allí, que accede a las tablas del nivel de datos.

Aplicación de una raíz agregada por repositorio

Puede ser útil implementar el diseño de repositorio de tal manera que aplique la regla de que solo las raíces agregadas deban tener repositorios. Puede crear un tipo de repositorio base o genérico que limite el tipo de las entidades con las que funciona para asegurarse de que tengan la interfaz de marcador `IAggregateRoot`.

Por tanto, cada clase de repositorio que se implemente en el nivel de infraestructura implementa su propio contrato o interfaz, como se muestra en el código siguiente:

```
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
        // ...
    }
}
```

Cada interfaz de repositorio específica implementa la interfaz genérica IRepository:

```
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);
    // ...
}
```

Pero una manera mejor de que el código aplique la convención de que cada repositorio esté relacionado con un único agregado consiste en implementar un tipo de repositorio genérico. De este modo, es explícito que se está usando un repositorio para tener como destino un agregado concreto. Eso se puede hacer fácilmente mediante la implementación de una interfaz base IRepository genérica, como se muestra en el código siguiente:

```
public interface IRepository<T> where T : IAggregateRoot
{
    //....
}
```

El modelo de repositorio facilita probar la lógica de la aplicación

El modelo de repositorio permite probar fácilmente la aplicación en pruebas unitarias. Recuerde que en las pruebas unitarias solo se prueba el código, no la infraestructura, por lo que las abstracciones de repositorio facilitan alcanzar ese objetivo.

Como se indicó en una sección anterior, se recomienda definir y colocar las interfaces de repositorio en el nivel de modelo de dominio para que el nivel de aplicación (como el microservicio de API web) no dependa directamente del nivel de infraestructura en el que se han implementado las clases de repositorio reales. Al hacer esto y usar la inserción de dependencias en los controladores de la API web, puede implementar repositorios ficticios que devuelven datos falsos en lugar de datos de la base de datos. Ese enfoque desacoplado permite crear y ejecutar pruebas unitarias que centran la lógica de la aplicación sin necesidad de conectividad a la base de datos.

Se pueden producir errores en las conexiones a las bases de datos y, más importante aún, la ejecución de centenares de pruebas en una base de datos no es recomendable por dos motivos. En primer lugar, puede tardar mucho tiempo debido al gran número de pruebas. En segundo lugar, es posible que los registros de base de datos cambien y afecten a los resultados de las pruebas, por lo que podrían no ser coherentes. Realizar pruebas en la base de datos no es una prueba unitaria sino una prueba de integración. Debería tener muchas pruebas unitarias que se ejecuten con rapidez, pero menos pruebas de integración sobre 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, asume que la clase del repositorio las almacenará correctamente. El aspecto importante aquí es crear pruebas unitarias para el modelo de dominio y su lógica de dominio. Las raíces agregadas son los límites de coherencia principales en DDD.

Los repositorios que se implementa en eShopOnContainers se basan en la implementación de DbContext de EF Core de los patrones repositorio y unidad de trabajo mediante el seguimiento de cambios, por lo que no duplican esta funcionalidad.

La diferencia entre el modelo de repositorio y el patrón de clases de acceso a datos (DAL) heredado

Un objeto de acceso a datos realiza directamente operaciones de acceso y persistencia de datos en el almacenamiento. Un repositorio marca los datos con las operaciones que se quieren realizar en la memoria de un objeto de unidad de trabajo (como sucede en EF al usar la clase [DbContext](#)), pero estas actualizaciones no se realizan de forma inmediata en la base de datos.

Una unidad de trabajo se conoce como una sola transacción que implica varias operaciones de inserción, actualización o eliminación. En otras palabras, significa que para una acción de usuario específica (como el registro en un sitio web) todas las transacciones de inserción, actualización o eliminación se administran en una única operación. Esto es más eficaz que el control de varias transacciones de base de datos de una manera profusa.

Estos operaciones de persistencia múltiples se realizan más adelante en una sola acción cuando el código del nivel de aplicación lo ordena. La decisión sobre cómo aplicar los cambios en memoria al almacenamiento de base de datos real normalmente se basa en el [patrón de unidades de trabajo](#). En EF, el patrón de unidades de trabajo se implementa como el [DbContext](#).

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 tanto, el ORM seleccionado puede optimizar la ejecución en la base de datos mediante la agrupación de varias acciones de actualización en la misma transacción, en lugar de muchas ejecuciones de transacciones pequeñas e independientes.

Los repositorios no deben ser obligatorios

Los repositorios personalizados son útiles por los motivos citados anteriormente, y es el enfoque para el microservicio de pedidos de eShopOnContainers. Pero no es un patrón esencial para implementar en un diseño de DDD o incluso en el desarrollo general de .NET.

Por ejemplo, Jimmy Bogard, al proporcionar información directa para esta guía, afirmó lo siguiente:

Este probablemente será mi comentario más importante. No soy un gran defensor de los repositorios, sobre todo porque ocultan los detalles importantes del mecanismo de persistencia subyacente. Por ese motivo también prefiero MediatR para los comandos. Puedo usar toda la funcionalidad de la capa de persistencia e insertar todo ese comportamiento de dominio en las raíces agregadas. Normalmente no me interesa simular los repositorios: sigo necesitando que esa prueba de integración esté con la acción real. La elección de CQRS significaba que realmente ya no necesitábamos los repositorios.

Los repositorios pueden ser útiles, pero no esenciales para el diseño de DDD, de la misma forma que el patrón de agregado y el modelo de dominio enriquecido lo son. Por tanto, use el modelo de repositorio o no, como considere oportuno. En cualquier caso, se usará el modelo de repositorio siempre que se use EF Core, aunque en este caso el repositorio cubre todo el microservicio o contexto delimitado.

Recursos adicionales

Modelo de repositorio

- **The Repository Pattern** \ (El modelo de repositorio)
<https://deviq.com/repository-pattern/>
- **Edward Hieatt y Rob Mee. Modelo de repositorio.**
<https://martinfowler.com/eaaCatalog/repository.html>
- **The Repository Pattern** \ (El modelo de repositorio)
[https://docs.microsoft.com/previous-versions/msp-n-p/ff649690\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-n-p/ff649690(v=pandp.10))

- **Eric Evans. Diseño orientado al dominio: Tackling Complexity in the Heart of Software.** (Diseño orientado al dominio: abordar la complejidad en el corazón del software) (Libro; incluye un debate sobre el patrón de repositorio)
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

Patrón de unidades de trabajo

- **Martin Fowler. Unit of Work pattern** (Patrón de unidades de trabajo).
<https://martinfowler.com/eaaCatalog/unitOfWork.html>
- **Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application **
(Implementación de los patrones de repositorio y unidad de trabajo en una aplicación ASP.NET MVC)
<https://docs.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>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación del nivel de persistencia de infraestructura con Entity Framework Core

25/11/2019 • 30 minutes to read • [Edit Online](#)

Al utilizar bases de datos relacionales, como SQL Server, Oracle o PostgreSQL, se recomienda implementar el nivel de persistencia basado 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 hace mucho tiempo que forma parte de .NET Framework. Al utilizar .NET Core, también debe usar Entity Framework Core, que se ejecuta en Windows o Linux de la misma manera que .NET Core. EF Core es una reescritura completa de Entity Framework, que se implementa con una superficie mucho menor y con 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 Entity Framework. Se introdujo con .NET Core a mediados de 2016.

Puesto que en la documentación de Microsoft ya hay una introducción a EF Core, aquí nos limitaremos a proporcionar vínculos a dicha información.

Recursos adicionales

- **Entity Framework Core**
<https://docs.microsoft.com/ef/core/>
- **ASP.NET Core MVC con EF Core: serie de tutoriales**
<https://docs.microsoft.com/aspnet/core/data/ef-mvc/>
- **Clase DbContext**
<https://docs.microsoft.com/dotnet/api/microsoft.entityframeworkcore.dbcontext>
- **Comparación de EF Core y EF6**
<https://docs.microsoft.com/ef/efcore-and-ef6/index>

Infraestructura en Entity Framework Core desde una perspectiva DDD

Desde un punto de vista DDD, una capacidad importante de EF es la de utilizar las entidades de dominio POCO, también conocidas en terminología de EF como *entidades Code First* de 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 DDD, debe encapsular las reglas y el comportamiento de dominio dentro de la misma clase de entidad, por lo que puede controlar las invariantes, las validaciones y las 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 cambio, es interesante exponer métodos que controlen cómo y cuándo se pueden actualizar los campos y las colecciones de propiedades, y qué comportamiento y qué acciones se producirán cuando esto ocurra.

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 quiere que se pueda acceder a un campo de entidad desde el exterior, solo puede crear un campo o un atributo en vez 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. En las versiones anteriores de Entity Framework, se requerían propiedades de colección para admitir `ICollection<T>`, lo que significaba que cualquier desarrollador que usara la clase de entidad primaria podía agregar o quitar elementos a través de sus colecciones de propiedades. Esa posibilidad iría en contra de los patrones recomendados en DDD.

Puede usar una colección privada al mismo tiempo que expone un objeto `IReadOnlyCollection<T>` de solo lectura, como se muestra en el ejemplo de código siguiente:

```
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);
    }
}
```

Tenga en cuenta que solo se puede obtener acceso de solo lectura a la propiedad `OrderItems`, mediante `IReadOnlyCollection<OrderItem>`. Este tipo es de solo lectura, por lo que está protegido frente a las actualizaciones externas normales.

EF Core proporciona una manera de asignar el modelo de dominio a la base de datos física sin que "contamine" el modelo de dominio. Se trata de código POCO puro de .NET, puesto que la acción de asignación se implementa en el nivel 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 la clase `OrderEntityTypeConfiguration`, la llamada a `SetPropertyAccessMode` indica a EF Core que debe acceder a la propiedad `OrderItems` a través de su campo.

```

// 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
    }
}

```

Al usar campos en lugar de propiedades, se conserva la entidad `orderItem` como si tuviera una propiedad `List<OrderItem>`. Pero expone un descriptor de acceso único, el método `AddOrderItem`, para agregar nuevos elementos al pedido. Como resultado, el comportamiento y los datos permanecen unidos y son coherentes a lo largo de cualquier código de aplicación que utilice el modelo de dominio.

Implementación de los repositorios personalizados con Entity Framework Core

En el nivel de implementación, un repositorio no es más que una clase con código de persistencia de datos coordinada por una unidad de trabajo (DbContext en EF Core) al realizar actualizaciones, como se muestra en la clase siguiente:

```

// using statements...
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;
        }
    }
}

```

Tenga en cuenta que la interfaz `IBuyerRepository` proviene del nivel de modelo de dominio como un contrato. Pero 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 puede establecerse explícitamente con `services.AddDbContext<>`).

Métodos que se pueden 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 de forma que queden contenidos por su agregado relacionado. Recuerde que hay una relación de uno a uno entre un agregado y su repositorio relacionado. Tenga en cuenta que un objeto entidad de raíz agregada podría tener entidades secundarias insertadas en su gráfico de EF. Por ejemplo, un comprador puede tener varias formas de pago como entidades secundarias relacionadas.

Como el enfoque para el microservicio de ordenación en `eShopOnContainers` también se basa en CQS/CQRS, la mayoría de consultas no se implementa en repositorios personalizados. Los desarrolladores pueden crear libremente las consultas y combinaciones que necesiten para el nivel de presentación sin las restricciones impuestas por agregados, repositorios personalizados por agregado y DDD en general. La mayoría de repositorios personalizados sugeridos por esta guía tiene varios métodos de actualización o transacción, pero solo se actualizan los métodos de consulta necesarios para obtener los datos. Por ejemplo, el repositorio `BuyerRepository` implementa un método `FindAsync`, porque la aplicación necesita saber si existe un comprador determinado antes de crear un nuevo comprador relacionado con el pedido.

Pero los métodos de consulta reales para obtener los datos que se van a enviar al nivel de presentación o a las

aplicaciones cliente se implementan, como se ha mencionado, en las consultas CQRS basadas en consultas flexibles mediante Dapper.

Uso de un repositorio personalizado frente al uso de DbContext EF directamente

La clase DbContext de Entity Framework se basa en los patrones de unidad de trabajo y repositorio, y puede utilizarse directamente desde el código, así como desde un controlador MVC de ASP.NET Core. Esta es la forma de crear el código más sencillo, como en el microservicio de catálogo CRUD en eShopOnContainers. En los casos en los que quiera disponer del código más sencillo posible, puede utilizar directamente la clase DbContext, igual que muchos desarrolladores.

Pero implementar repositorios personalizados ofrece varias ventajas al implementar aplicaciones o microservicios más complejos. Los patrones de unidad de trabajo y repositorio están diseñados para encapsular el nivel de persistencia de infraestructura de tal modo que se separe de los niveles de aplicación y de modelo de dominio. Implementar estos patrones puede facilitar el uso de repositorios ficticios que simulen 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) y usar repositorios que faciliten la simulación de los repositorios.

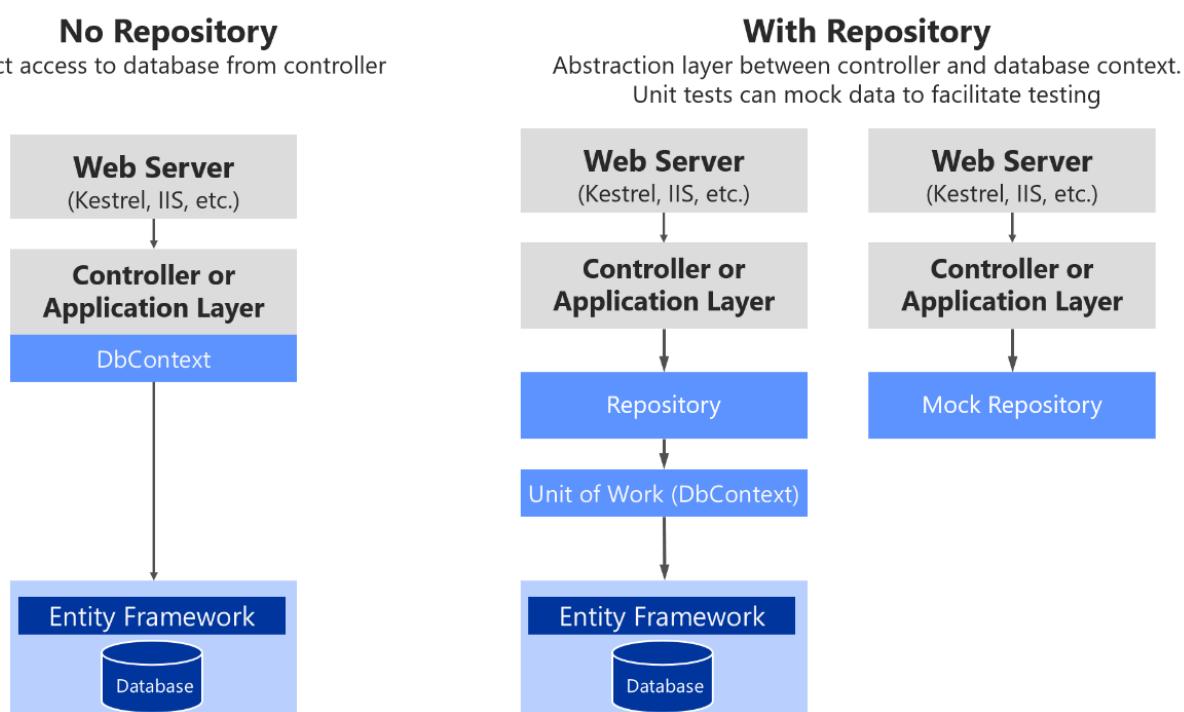


Figura 7-18. Uso de repositorios personalizados frente a DbContext sin formato

En la figura 7-18 se muestra que el uso de un repositorio personalizado agrega una capa de abstracción que permite simular el repositorio para facilitar las pruebas. Hay varias alternativas al plantear una simulación. Puede limitarse a simular repositorios o puede 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 centremos en el nivel de aplicación, verá cómo funciona la inserción de dependencias en ASP.NET Core y cómo se implementa al utilizar repositorios.

En resumen, los repositorios personalizados le permiten probar el código más fácilmente con pruebas unitarias que no se ven afectadas por el estado de la capa de datos. Si ejecuta pruebas que también tienen acceso a la base de datos real a través de Entity Framework, no se trata de pruebas unitarias sino de pruebas de integración, que son mucho más lentas.

Si estaba usando DbContext directamente, tendría que simularlo o ejecutar pruebas unitarias mediante el uso de SQL Server en la memoria 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

puede probar los controladores MVC.

Duración de DbContext de EF y de la instancia IUnitOfWork en el contenedor de IoC

El objeto `DbContext` (expuesto como un objeto `IUnitOfWork`) debería compartirse entre varios repositorios dentro del mismo ámbito de solicitud HTTP. Por ejemplo, esto sucede cuando la operación que se está ejecutando debe tratar con varios agregados o simplemente porque está usando 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, hay que establecer la duración del servicio de la instancia del objeto `DbContext` en `ServiceLifetime.Scoped`. Se trata de la duración predeterminada al registrar `DbContext` con `services.AddDbContext` en el contenedor de IoC desde el método `ConfigureServices` del archivo `Startup.cs` en el proyecto de ASP.NET Core Web API. Esto se ilustra en el código siguiente:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc(options =>
    {
        options.Filters.Add(typeof(HttpGlobalExceptionFilter));
    }).AddControllersAsServices();

    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 se debe configurar como `ServiceLifetime.Transient` o `ServiceLifetime.Singleton`.

Duración de la instancia de repositorio en su contenedor IoC

De forma similar, la duración del repositorio normalmente se establece como determinada (`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.

```
// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
    .As<IOrderRepository>()
    .InstancePerLifetimeScope();
```

Tenga en cuenta que utilizar la duración de singleton para el repositorio puede causar problemas de simultaneidad graves al establecer `DbContext` en una duración determinada (`InstancePerLifetimeScope`) (duraciones predeterminadas para `DbContext`).

Recursos adicionales

- **Implementación de los patrones de repositorio y unidad de trabajo en una aplicación ASP.NET MVC**

<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 las duraciones de servicio del contenedor IoC de ASP-NET Core con ámbitos de instancia de contenedor Autofac IoC**

<https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/>

Asignación de tabla

La asignación de tabla identifica los datos de tabla que se van a consultar en la base de datos y que se guardarán en ella. Anteriormente, vimos cómo las entidades de domino (por ejemplo, un dominio de producto o de pedido) se podían usar para generar un esquema de base de datos relacionado. EF está diseñado basándose en el concepto de *convenciones*. Las convenciones generan preguntas como "¿Cuál será el nombre de la 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 el identificador.

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 valor `DbSet< TEntity >` a la entidad determinada, se utiliza el nombre de clase.

Anotaciones de datos frente a API fluida

Hay muchas convenciones de EF Core adicionales, la mayoría de las cuales se puede cambiar mediante anotaciones de datos o la API fluida, que se implementan con el método `OnModelCreating`.

Las anotaciones de datos se utilizan en las mismas clases del modelo de entidad, lo que supone un método más intrusivo desde el punto de vista de DDD. Esto es así porque el modelo se contamina con anotaciones de datos relacionadas con la base de datos de la infraestructura. Por otro lado, la API fluida es una forma práctica de cambiar la mayoría de convenciones y asignaciones en el nivel de infraestructura de la persistencia de datos, por lo que el modelo de entidad estará limpio y desacoplado de la infraestructura de persistencia.

API fluida y el método `OnModelCreating`

Como se ha indicado, puede usar el método `OnModelCreating` en la clase `DbContext` con el fin de cambiar las convenciones y las asignaciones.

El microservicio de ordenación en `eShopOnContainers` implementa configuraciones y asignaciones explícitas, cuando es necesario, tal y como se muestra en el código siguiente.

```

// 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)
            .ForSqlServerUseSequenceHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

        //Address Value Object persisted as owned entity type supported since EF Core 2.0
        orderConfiguration.OwnsOne(o => o.Address);

        orderConfiguration.Property<DateTime>("OrderDate").IsRequired();
        orderConfiguration.Property<int?>("BuyerId").IsRequired(false);
        orderConfiguration.Property<int>("OrderStatusId").IsRequired();
        orderConfiguration.Property<int?>("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 fluida dentro del mismo método OnModelCreating, pero se aconseja crear particiones en el código y tener varias clases de configuración, una por cada entidad, tal como se muestra en el ejemplo. Especialmente para modelos grandes, es aconsejable tener clases de configuración independientes para configurar diferentes tipos de entidad.

En el código de ejemplo se muestran algunas asignaciones y declaraciones explícitas. 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 de ejemplo anterior es que utiliza 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. A modo de resumen, el algoritmo Hi-Lo asigna identificadores únicos a filas de la tabla, pero no depende del almacenaje inmediato de la fila en la base de datos. Esto le permite empezar a usar los identificadores de forma inmediata, como sucede con los identificadores de la base de datos secuencial normal.

El algoritmo Hi-Lo describe un mecanismo para obtener un lote de identificadores únicos de una secuencia de una base de datos relacionada. Estos identificadores son seguros porque la base de datos garantiza que son únicos, por lo que no se producirán colisiones entre los usuarios. Este algoritmo es interesante por los siguientes motivos:

- No interrumpe el patrón de la unidad de trabajo.
- Obtiene la secuencia de id. por lotes, para minimizar los recorridos de ida y vuelta a la base de datos.
- Genera un identificador que pueden leer los humanos, a diferencia de las técnicas que utilizan los identificadores GUID.

EF Core es compatible con [HiLo](#) con el método `ForSqlServerUseSequenceHiLo`, tal como se muestra en el ejemplo anterior.

Asignación de campos en lugar de propiedades

Con esta característica, disponible desde la versión 1.1 de EF Core, puede asignar directamente columnas a los campos. Es posible no utilizar propiedades en la clase de entidad y simplemente asignar columnas de una tabla a los 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 únicos o también con colecciones, como si se tratara de un campo `List<>`. Este punto se mencionó anteriormente cuando analizamos el modelado de las clases de modelo de dominio, pero aquí puede ver cómo se realiza esta asignación con la configuración `PropertyAccessMode.Field` resaltada en el código anterior.

Uso de propiedades reemplazadas en EF Core y ocultas en el nivel de infraestructura

Las propiedades reemplazadas en EF Core son propiedades que no existen en su 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 consultas

Como se mencionó anteriormente en la sección de diseño, el patrón de especificación de consultas es un modelo de diseño controlado por dominios diseñado como el lugar donde se puede incluir la definición de una consulta con lógica opcional de ordenación y paginación.

El patrón de especificación de consultas define una consulta en un objeto. Por ejemplo, para encapsular una consulta paginada que busque algunos productos, se puede crear una especificación `PagedProduct` que tome los parámetros de entrada necesarios (`pageNumber`, `pageSize`, `filter`, etc.). Después, dentro de cualquier método del repositorio (normalmente una sobrecarga de `List()`) aceptaría una `IQuerySpecification` y ejecutaría la consulta esperada según esa especificación.

Un ejemplo de una interfaz Specification genérica es el siguiente código de [eShopOnWeb](#).

```
// 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; }
}
```

La siguiente es la implementación de una clase base de especificación genérica.

```
// 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
    // e.g. Basket.Items.Product
    protected virtual void AddInclude(string includeString)
    {
        IncludeStrings.Add(includeString);
    }
}
```

La siguiente especificación carga una entidad de cesta única a partir del id. o del id. de comprador al que pertenece la cesta y realiza una [carga diligente](#) de la colección de artículos de la cesta.

```
// 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 a continuación cómo un repositorio de EF genérico puede usar una especificación de este tipo para filtrar y cargar de forma diligente los datos relacionados con un determinado tipo de entidad T.

```

// 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 filtro, 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, se puede usar perfectamente dentro del repositorio para crear un conjunto de resultados. Puede ver cómo se usa este enfoque en el método List anterior, en que se utilizan expresiones IQueryable intermedias para generar la lista de consultas de inclusión antes de ejecutar la consulta con los criterios de especificación de la última línea.

Recursos adicionales

- **Asignación de tabla**
<https://docs.microsoft.com/ef/core/modeling/relational/tables>
- **Uso de Hi-Lo para generar claves con Entity Framework Core**
<https://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/>
- **Campos de respaldo**
<https://docs.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 reemplazadas**
<https://docs.microsoft.com/ef/core/modeling/shadow-properties>
- **Patrón de especificación**
<https://deviq.com/specification-pattern/>

[ANTERIOR](#)

[SIGUIENTE](#)

Uso de bases de datos NoSQL como una infraestructura de persistencia

25/11/2019 • 24 minutes to read • [Edit Online](#)

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, quizás 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.

```
{
  "id": "2017001",
  "orderDate": "2/25/2017",
  "buyerId": "1234567",
  "address": [
    {
      "street": "100 One Microsoft Way",
      "city": "Redmond",
      "state": "WA",
      "zip": "98052",
      "country": "U.S."
    }
  ],
  "orderItems": [
    {"id": 20170011, "productId": "123456", "productName": ".NET T-Shirt",
     "unitPrice": 25, "units": 2, "discount": 0},
    {"id": 20170012, "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. Es multimodelo y admite modelos de datos de documentos, clave-valor, gráfico y columnas.

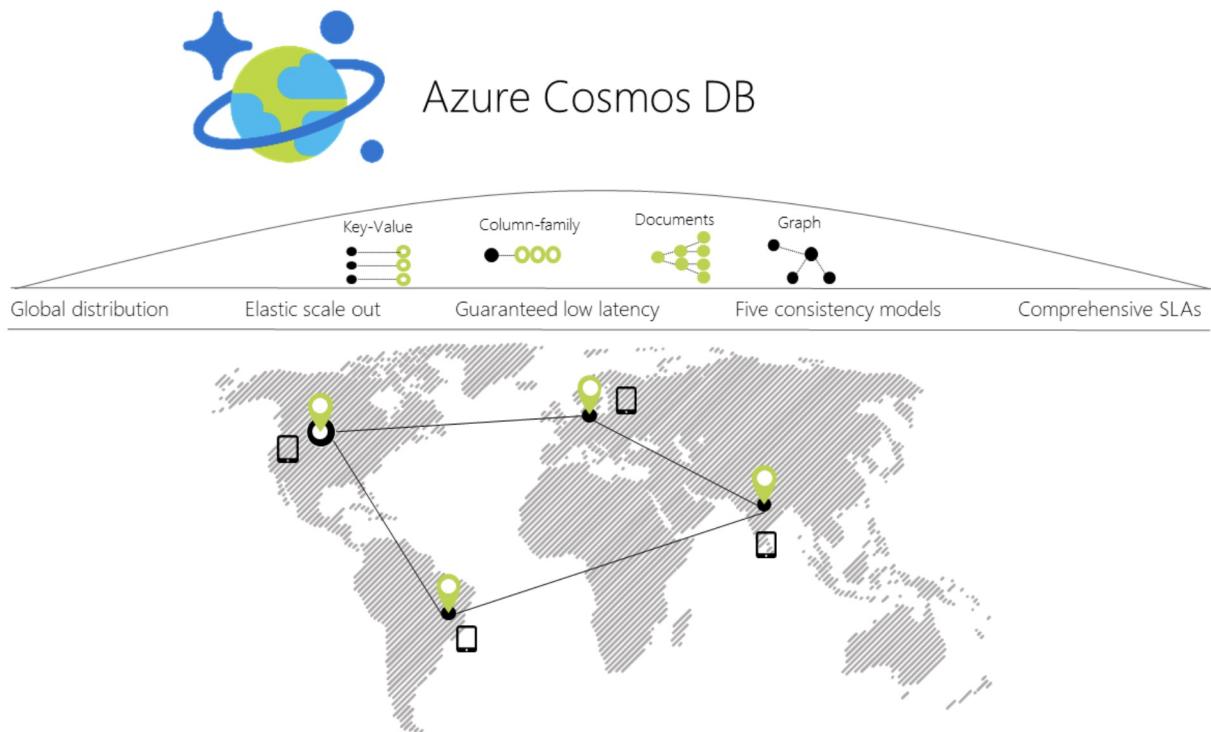


Figura 7-19. Distribución global de Azure Cosmos DB

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# 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 = "2017001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "P018009186470"
}

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 = 20170011,
    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. A pesar de que hay un [emulador de Azure Cosmos DB](#) local capaz de ejecutarse en una máquina de desarrollo local (por ejemplo, un equipo PC), hasta finales de 2017, solo es compatible con Windows, pero no con Linux.

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 hándicap 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](#).

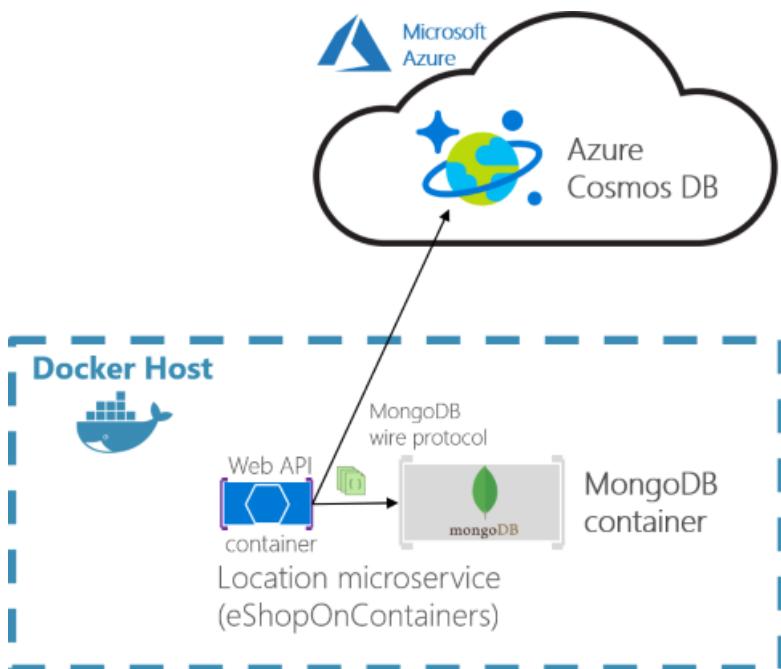


Figura 7-21. eShopOnContainers con contenedores de MongoDB para desarrollo o entorno o Azure Cosmos DB para producción

La base de datos Azure Cosmos DB de producción se ejecuta en la nube de Azure como un servicio escalable y de PaaS.

Los contenedores de .NET Core personalizados pueden ejecutarse en un host Docker de desarrollo local (es decir, con Docker para Windows en un equipo 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 Core, 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, estamos usando 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 se pretende utilizar la API de MongoDB para tener acceso a Azure Cosmos DB en aplicaciones de Azure para producción, se deben analizar y comparar las diferencias entre las capacidades y el rendimiento al usar la API de MongoDB para acceder a las bases de datos de Azure Cosmos DB y usar la API de Azure Cosmos DB nativa. 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 renuncia 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 Core

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.

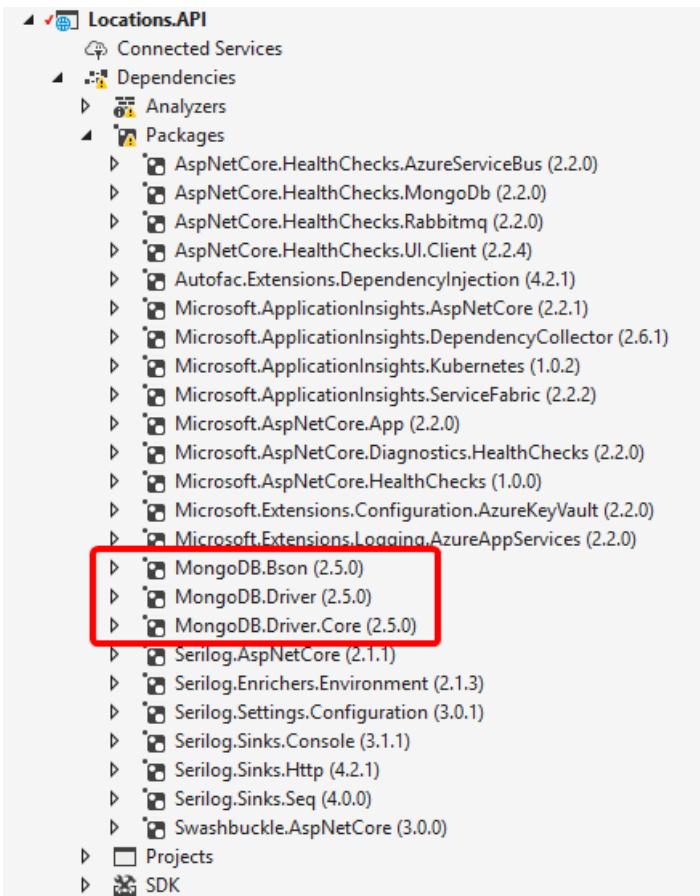


Figura 7-22. Referencias de paquetes NuGet de la API MongoDB en un proyecto de .NET Core

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.

```

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.

```

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.

```

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.

```

# docker-compose.override.yml
version: '3.4'
services:
    # Other services
    locations.api:
        environment:
            # Other settings
            - ConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosql.data}

```

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://nosql.data` 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:

```

# .env file, in eShopOnContainers root folder
# Other Docker environment variables

ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost
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>

```

Debe quitar la marca de comentario de la línea ESHOP_AZURE_COSMOSDB y actualizarla 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 la marca de comentario se ha quitado del archivo .env, entonces el contenedor usa una cadena de conexión de MongoDB predeterminada que apunta al contenedor de MongoDB local implementado en eShopOnContainers que se denomina nosql.data y se definió en el archivo docker-compose, tal y como se muestra en el siguiente código .yml.

```

# docker-compose.yml
version: '3.4'
services:
  # ...Other services...
  nosql.data:
    image: mongo

```

Recursos adicionales

- **Modelado de datos del documento para bases de datos NoSQL**
<https://docs.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 de MongoDB**
<https://docs.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://docs.microsoft.com/azure/cosmos-db/create-mongodb-dotnet>
- **Uso del Emulador de Azure Cosmos DB para desarrollo y pruebas de forma local**
<https://docs.microsoft.com/azure/cosmos-db/local-emulator>
- **Conectar una aplicación de MongoDB a Azure Cosmos DB**
<https://docs.microsoft.com/azure/cosmos-db/connect-mongodb-account>
- **Imagen de Docker del Emulador de Cosmos DB (contenedor Windows)**
<https://hub.docker.com/r/microsoft/azure-cosmosdb-emulator/>
- **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://docs.microsoft.com/azure/cosmos-db/mongodb-mongochef>

[ANTERIOR](#)

[SIGUIENTE](#)

Diseño del nivel de aplicación de microservicios y la API web

23/10/2019 • 3 minutes to read • [Edit Online](#)

Uso de principios SOLID e inserción de dependencias

Los principios SOLID son técnicas fundamentales para utilizar en cualquier aplicación moderna y crítica, como para el desarrollo de un microservicio con patrones DDD. En inglés, SOLID representa un acrónimo que agrupa cinco principios fundamentales:

- Principio de responsabilidad única
- Principio de abierto y cerrado
- Principio de sustitución de Liskov
- Principio de segregación de interfaces
- Principio de inversión de dependencias

SOLID hace referencia a la forma de diseñar los niveles internos de una aplicación o de un microservicio, así como a separar las dependencias entre ellas. No está relacionado con el dominio, sino con el diseño técnico de la aplicación. El principio final, el de inversión de dependencias, le permite desacoplar el nivel de infraestructura del resto de niveles, lo que permite una mejor implementación desacoplada de los niveles de DDD.

La inserción de dependencias (DI) es una forma de implementar el principio de inversión de dependencias. Es una técnica para lograr el acoplamiento flexible entre los objetos y sus dependencias. En lugar de crear directamente instancias de colaboradores o de usar referencias estáticas (es decir, usar new...), los objetos que una clase necesita para llevar a cabo sus acciones se proporcionan a la clase (o se "insertan" en ella). A menudo, las clases declaran sus dependencias a través de su constructor, lo que les permite seguir el principio de dependencias explícitas. Normalmente, la inserción de dependencias se basa en determinados contenedores de Inversión de control (IoC). ASP.NET Core proporciona un sencillo contenedor de IoC integrado. Aun así, usted puede usar el contenedor de IoC que prefiera, como Autofac o Ninject.

Siguiendo los principios SOLID, las clases tenderán naturalmente a ser pequeñas, a estar factorizadas correctamente y a poder probarse fácilmente. Pero, ¿cómo puede saber si se van a insertar demasiadas dependencias en sus clases? Si usa la inversión de dependencias a través del constructor, le resultará fácil saberlo con solo mirar el número de parámetros de su 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.

Necesitaríamos otra guía para tratar SOLID con detalle. Para esta guía solo necesita tener unos conocimientos mínimos de estos temas.

Recursos adicionales

- **SOLID: SOLID: principios fundamentales de OOP**
<https://deviq.com/solid/>
- **Contenedores de Inversión de control y el patrón de inserción de dependencias**
<https://martinfowler.com/articles/injection.html>
- **Steve Smith. New es como pegamento**
<https://ardalis.com/new-is-glue>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación del nivel de aplicación de microservicios mediante la API web

25/11/2019 • 57 minutes to read • [Edit Online](#)

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.

:::image type="complex" source="./media/microservice-application-layer-implementation-web-api/ordering-api-microservice.png" alt="Captura de pantalla del Explorador de soluciones de Ordering.API mostrando las subcarpetas Application: Behaviors, Commands, DomainEventHandlers, IntegrationEvents, Models, Queries y Validations.":::
Vista del Explorador de soluciones del microservicio Ordering.API que muestra las subcarpetas de la carpeta Application: Behaviors, Commands, DomainEventHandlers, IntegrationEvents, Models, Queries y Validations.
:::image-end:::

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 método `ConfigureServices` de la clase `Startup` de la aplicación. Las dependencias se implementan en los servicios que un tipo necesita y que se registran en el contenedor IoC.

Normalmente, le interesaría insertar dependencias que implementen objetos de infraestructura. Una dependencia muy 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 Core inserta los objetos de repositorio necesarios a través del constructor. La clase es un controlador de comandos, que se explica en la sección siguiente.

```

public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                      IOrderRepository orderRepository,
                                      IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
            throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
            throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
            throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Create the Order AggregateRoot
        // 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, 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);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}

```

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 proporcionado por ASP.NET Core, registre los tipos que quiera insertar en el método ConfigureServices del archivo Startup.cs, como se muestra en el código siguiente:

```
// Registration of types into ASP.NET Core built-in container
public void ConfigureServices(IServiceCollection services)
{
    // Register out-of-the-box framework services.
    services.AddDbContext<CatalogContext>(c =>
    {
        c.UseSqlServer(Configuration["ConnectionString"]);
    },
    ServiceLifetime.Scoped
);
    services.AddMvc();
    // Register custom application dependencies.
    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 Core, 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.

```

public class ApplicationModule : Autofac.Module
{
    public string QueriesConnectionString { get; }

    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }

    protected override void Load(ContainerBuilder builder)
    {
        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://docs.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.

High level “Writes-side” in CQRS

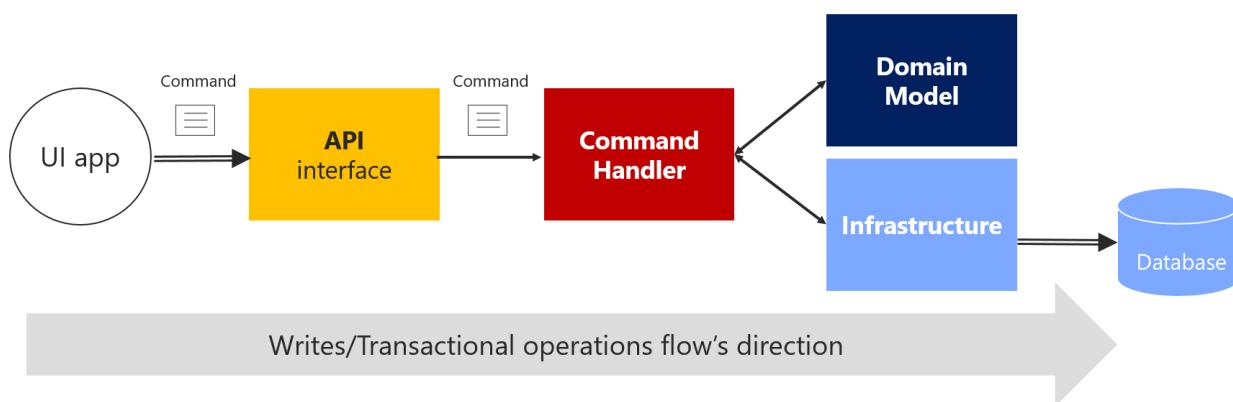


Figura 7-24. Vista general de los comandos o el "lado transaccional" en un patrón CQRS

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 e 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.

```
// DDD and CQRS patterns comment
// Note that we recommend that you implement immutable commands
// In this case, immutability is achieved by having all the setters as private
// plus being able to update the data just once, when creating the object
// through the 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://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/how-to-implement-a-
lightweight-class-with-auto-implemented-properties
[DataContract]
public class CreateOrderCommand
    : IAsyncRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;
    [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 city,
    string street,
    string state, string country, string zipcode,
    string cardNumber, string cardHolderName, DateTime cardExpiration,
    string cardSecurityNumber, int cardTypeId) : this()
{
    _orderItems = MapToOrderItems(basketItems);
    City = city;
    Street = street;
    State = state;
    Country = country;
    ZipCode = zipcode;
    CardNumber = cardNumber;
    CardHolderName = cardHolderName;
    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 pretende o espera que los comandos pasen a través de un proceso de serialización o deserialización, las propiedades deben tener un establecedor privado y el atributo `[DataMember]` (o `[JsonProperty]`), en caso contrario, el deserializador no podrá reconstruir el objeto en el destino con los valores necesarios.

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:

```

[DataContract]
public class UpdateOrderStatusCommand
    : IAsyncRequest<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.

La 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 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. Pero 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. Reviselos 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, se pretende resaltar el método `Handle` y las operaciones con los objetos de modelo de dominio y agregados.

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                      IOrderRepository orderRepository,
                                      IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
                          throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
                          throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
                          throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Create the Order AggregateRoot
        // 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, 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);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}
```

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 correcta y una vez finalizada la transacción, generar eventos de integración. (Es posible que clases de infraestructura como repositorios también los generen).

- **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/commands-and-events>
- **What does a command handler do? (¿De qué se encarga un controlador de comandos?)**
<https://cqrs.nu/Faq/command-handlers>
- **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.

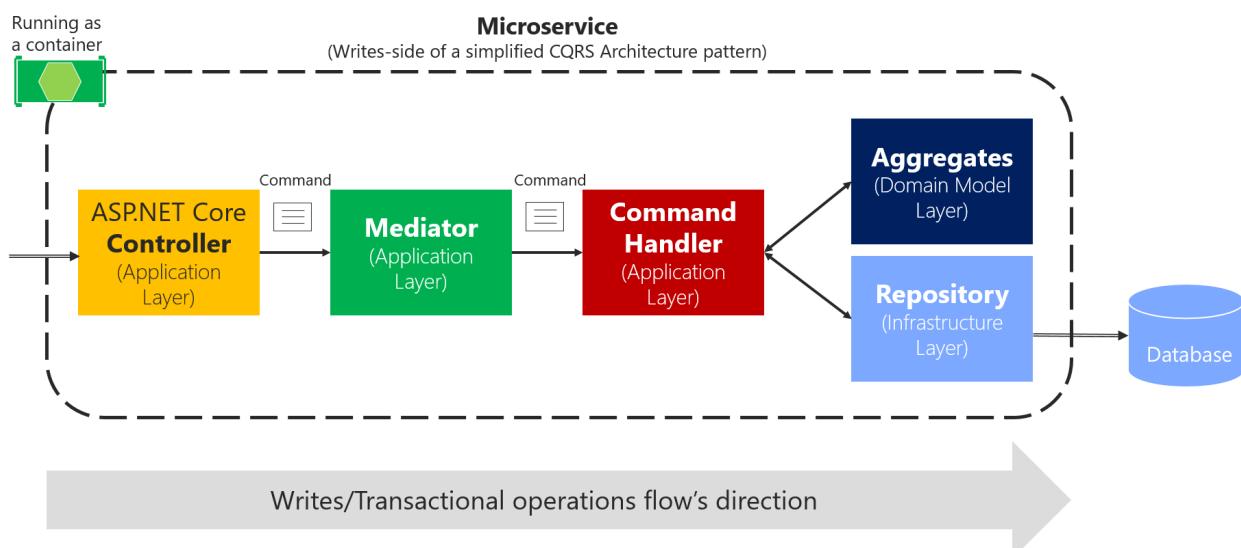


Figura 7-25. Uso del patrón de mediador en proceso en un único microservicio CQRS

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, en el microservicio de pedidos de eShopOnContainers, se implementaron dos comportamientos de ejemplo, las clases [LogBehavior](#) y [ValidatorBehavior](#). En la siguiente sección se explica la implementación de los comportamientos mostrando cómo eShopOnContainers usa los [comportamientos de MediatR 3](#).

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.

Writes-side of a CQRS Architecture pattern using messaging

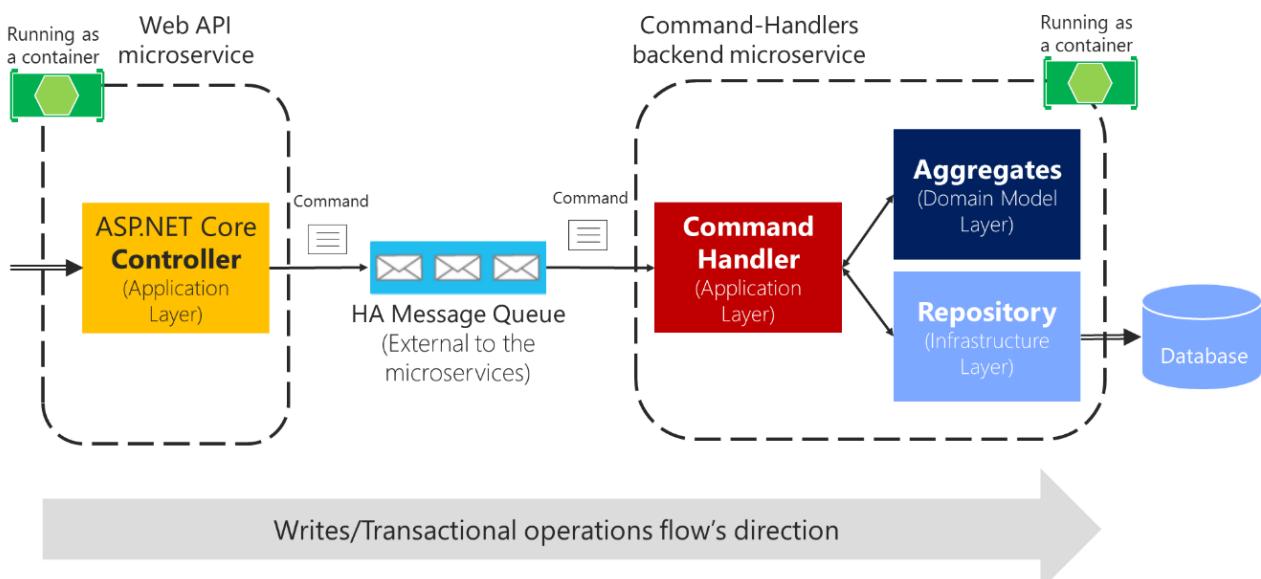


Figura 7-26. Uso de colas de mensajes (comunicación fuera de proceso y entre procesos) con comandos CQRS

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 será 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 gran cantidad de código en la que los usuarios usan 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 de la aplicación 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 realice una acción [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 ingestión 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 Core, 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 bastante 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:

```
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:

```
[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.

```

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);

```

Pero este caso también es un poco 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 del que se realiza el seguimiento con 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.

```

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.

```

// IdentifiedCommandHandler.cs
public class IdentifiedCommandHandler<T, R> :
    IAsyncRequestHandler<IdentifiedCommand<T, R>, R>
    where T : IRequest<R>
{
    private readonly IMediator _mediator;
    private readonly IRequestManager _requestManager;

    public IdentifiedCommandHandler(IMediator mediator,
                                    IRequestManager requestManager)
    {
        _mediator = mediator;
        _requestManager = requestManager;
    }

    protected virtual R CreateResultForDuplicateRequest()
    {
        return default(R);
    }

    public async Task<R> Handle(IdentifiedCommand<T, R> message)
    {
        var alreadyExists = await _requestManager.ExistAsync(message.Id);
        if (alreadyExists)
        {
            return CreateResultForDuplicateRequest();
        }
        else
        {
            await _requestManager.CreateRequestForCommandAsync<T>(message.Id);

            // Send the embedded business command to mediator
            // so it runs its related CommandHandler
            var result = await _mediator.Send(message.Command);

            return result;
        }
    }
}

```

Dado que IdentifiedCommand actúa como la envoltura 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 en la base de datos Ordering, como se muestra en el código siguiente.

```

// CreateOrderCommandHandler.cs
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                      IOrderRepository orderRepository,
                                      IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
            throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
            throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
            throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Add/Update the Buyer AggregateRoot
        var address = new Address(message.Street, message.City, message.State,
                                  message.Country, message.ZipCode);
        var order = new Order(message.UserId, 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);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}

```

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.

```

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 IAsyncRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(
            typeof(CreateOrderCommand).GetTypeInfo().Assembly).
            AsClosedTypesOf(typeof(IAsyncRequestHandler<,>));

        // Other types registration
        //...
    }
}

```

Aquí es donde "ocurre la magia" con MediatR.

Como cada controlador de comandos implementa la interfaz genérica `IAsyncRequestHandler<T>`, al registrar los ensamblados, el código registra con `RegisteredAssemblyTypes` todos los tipos marcados como `IAsyncRequestHandler` mientras relaciona los `CommandHandlers` con sus `Commands`, gracias a la relación indicada en la clase `CommandHandler`, como en el siguiente ejemplo:

```

public class CreateOrderCommandHandler
    : IAsyncRequestHandler<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.

```

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 IAsyncRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(
            typeof(CreateOrderCommand).GetTypeInfo().Assembly).
            AsClosedTypesOf(typeof(IAsyncRequestHandler<,>));

        // 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.

```

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:

```

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;
    }
}

```

El comportamiento aquí 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:

```

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

23/10/2019 • 2 minutes to read • [Edit Online](#)

Sus aplicaciones basadas en microservicios y en la nube deben estar preparadas para los errores parciales que seguramente se acabarán produciendo en algún momento. Debe diseñar su aplicación de modo que sea resistente a estos errores parciales.

La resistencia es la capacidad de recuperarse de errores y seguir funcionando. No se trata de evitar los errores, sino de aceptar el hecho de que se producirán errores y responder a ellos para evitar el tiempo de inactividad o la pérdida de datos. El objetivo de la resistencia consiste en que la aplicación vuelva a un estado totalmente operativo después de un error.

Es todo un desafío diseñar e implementar una aplicación basada en microservicios. Pero también necesita mantener la aplicación en ejecución en un entorno en el que con seguridad se producirá algún tipo de error. Por lo tanto, la aplicación debe ser resistente. Debe estar diseñada para hacer frente a errores parciales, como las interrupciones de red o el bloqueo de nodos o máquinas virtuales en la nube. Incluso los microservicios (contenedores) que se mueven a otro nodo dentro de un clúster pueden causar breves errores intermitentes dentro de la aplicación.

Los numerosos componentes individuales de la aplicación también deberían incorporar características de seguimiento de estado. Mediante las directrices descritas en este capítulo, podrá crear una aplicación que funcione sin problemas aunque se produzcan tiempos de inactividad transitorios o las interrupciones típicas de las implementaciones complejas y basadas en la nube.

[ANTERIOR](#)

[SIGUIENTE](#)

Controlar errores parciales

25/11/2019 • 5 minutes to read • [Edit Online](#)

En sistemas distribuidos como las aplicaciones basadas en microservicios, hay un riesgo siempre presente de error parcial. Por ejemplo, se puede producir un error en un único contenedor o microservicio o este podría no estar disponible para responder durante un breve período, o se podría bloquear una única máquina virtual o un servidor. Puesto que los clientes y los servicios son procesos independientes, es posible que un servicio no pueda responder de forma oportuna a una solicitud del cliente. Es posible que el servicio esté sobrecargado y responda muy lentamente a las solicitudes, o bien que simplemente no sea accesible durante un breve período debido a problemas de red.

Por ejemplo, considere la página de detalles Order de la aplicación de ejemplo eShopOnContainers. Si el microservicio Ordering no responde cuando el usuario intenta enviar un pedido, una implementación incorrecta del proceso del cliente (la aplicación web MVC), por ejemplo, si el código de cliente usara RPC sincrónicas sin tiempo de espera, bloquearía los subprocessos indefinidamente en espera de una respuesta. Además de la mala experiencia del usuario, cada espera sin respuesta usa o bloquea un subprocesso, y los subprocessos son muy valiosos en aplicaciones altamente escalables. Si hay muchos subprocessos bloqueados, al final el tiempo de ejecución de la aplicación puede quedarse sin subprocessos. En ese caso, la aplicación puede no responder globalmente en lugar de solo parcialmente, como se muestra en la figura 8-1.

Partial failures

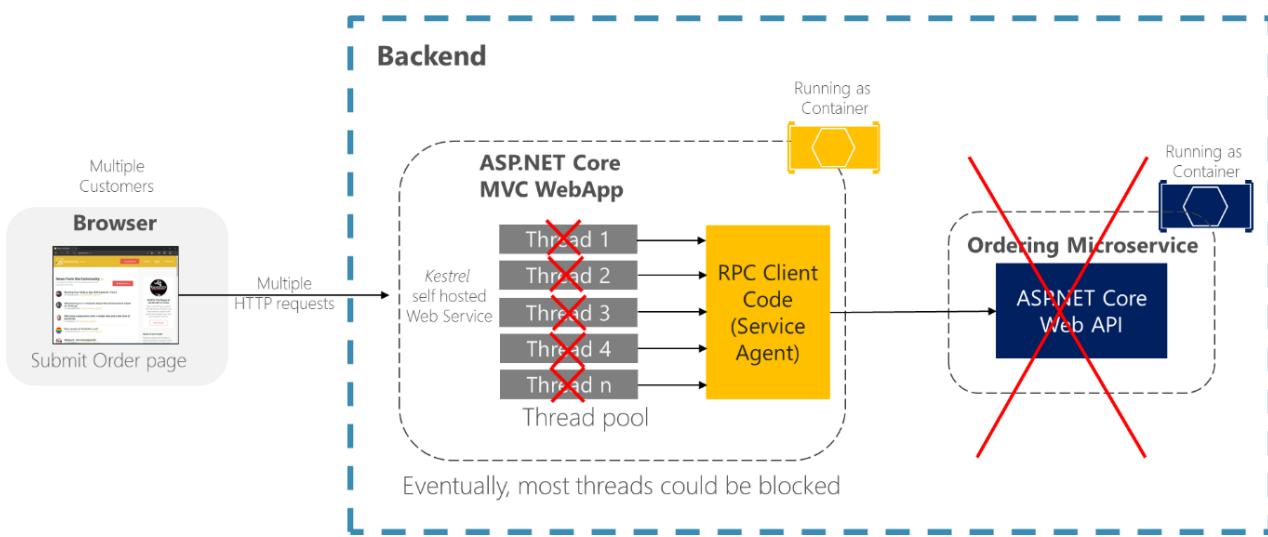


Figura 8-1. Errores parciales debido a dependencias que afectan a la disponibilidad del subprocesso de servicio

En una aplicación basada en microservicios de gran tamaño, cualquier error parcial se puede amplificar, especialmente si la mayor parte de la interacción de los microservicios internos se basa en llamadas HTTP sincrónicas (lo que se considera un anti-patrón). 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 proporción 1:4) a decenas de microservicios internos como dependencias sincrónicas. Esta situación se muestra en la figura 8-2, especialmente la dependencia #3, que inicia una cadena y llama a la dependencia 4, que llama a la dependencia 5.

Multiple distributed dependencies

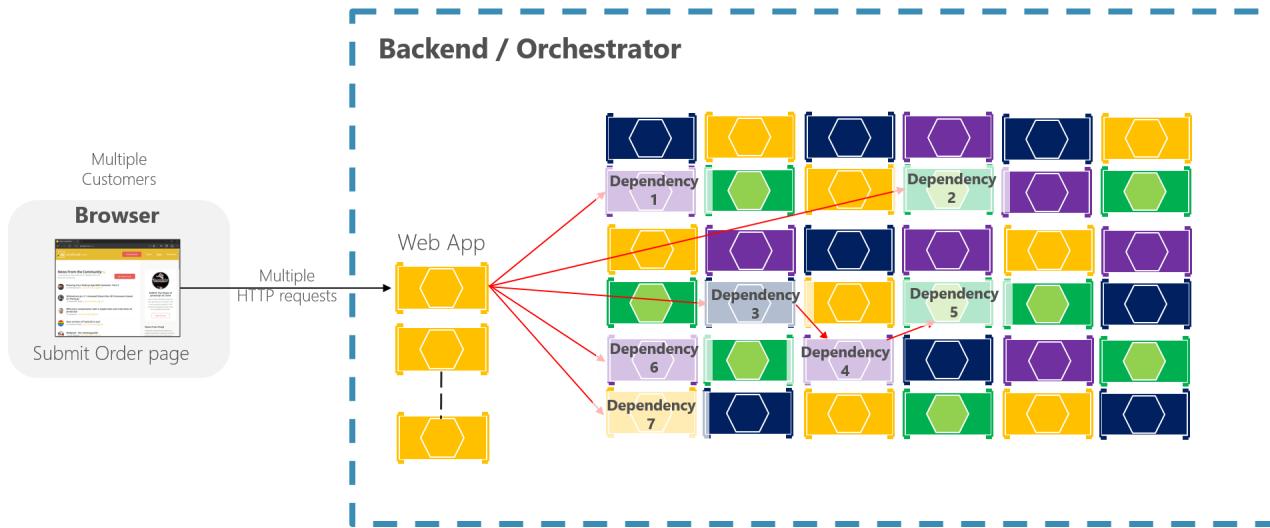


Figura 8-2. Impacto de tener un diseño incorrecto que incluye cadenas largas de solicitudes HTTP

Los errores intermitentes están garantizados en un sistema distribuido basado en la nube, aunque cada dependencia tenga una disponibilidad excelente. Es un hecho que se debe 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 con un 99,99 % de disponibilidad cada una darían lugar a varias horas de tiempo de inactividad al mes debido a este efecto dominó. Cuando se produce un error en una dependencia de un microservicio al controlar un gran volumen de solicitudes, ese error puede saturar rápidamente todos los subprocessos de solicitudes disponibles en cada servicio y bloquear toda la aplicación.

Partial Failure Amplified in Microservices

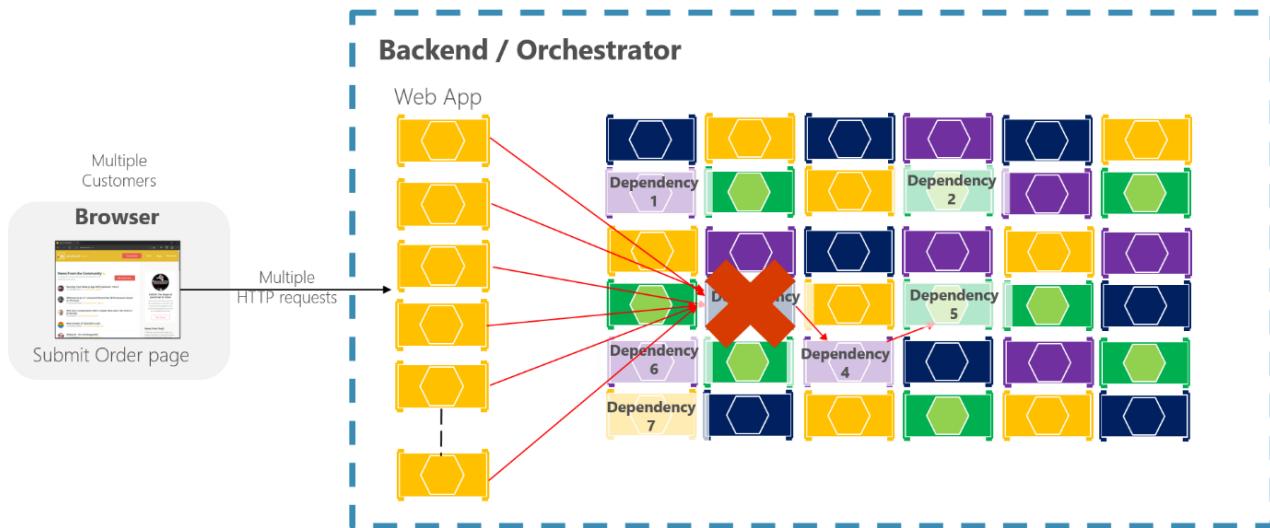


Figura 8-3. Error parcial amplificado por los microservicios con cadenas largas de llamadas HTTP sincrónicas

Para minimizar este problema, en la sección [La integración asincrónica del microservicio obliga a su autonomía](#), esta guía recomienda usar la comunicación asincrónica entre los microservicios internos.

Además, es fundamental que diseñe las aplicaciones cliente y los microservicios para controlar los errores parciales, es decir, que compile microservicios y aplicaciones cliente resistentes.

Estrategias para controlar errores parciales

23/10/2019 • 5 minutes to read • [Edit Online](#)

Las estrategias para tratar los errores parciales son las siguientes.

Usar la comunicación asincrónica (por ejemplo, la comunicación basada en mensajes) a través de microservicios internos. Es muy aconsejable no crear cadenas largas de llamadas HTTP sincrónicas a través de los microservicios internos, porque ese diseño incorrecto podría convertirse en la principal causa de interrupciones incorrectas. Por el contrario, excepto en el caso de comunicaciones front-end entre las aplicaciones cliente y el primer nivel de microservicios o puertas de enlace de API específicas, se recomienda usar solamente una comunicación asíncrona (basada en mensajes) cuando haya pasado el ciclo inicial de solicitud/respuesta en los microservicios internos. La coherencia definitiva y las arquitecturas orientadas a eventos le ayudarán a minimizar el efecto dominó. Estos enfoques exigen un nivel más alto de autonomía de microservicio y, por lo tanto, evitan el problema que se describe a continuación.

Usar reintentos con retroceso exponencial. Esta técnica ayuda a evitar fallos cortos e intermitentes mediante la realización de un número determinado de intentos de llamada en caso de que el servicio no esté disponible solo durante un breve período de tiempo. Esto puede ocurrir debido a problemas de red intermitentes o cuando un contenedor o microservicio se mueve a otro nodo del 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. Utilizar tiempos de expiración garantiza que los recursos nunca se bloqueen indefinidamente.

Usar el patrón de interruptor. En este enfoque, el proceso de cliente supervisa el 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 producen errores en un gran número de solicitudes, esto sugiere que el servicio no está disponible y que enviar solicitudes no sirve de nada.) Tras un período de tiempo de expiración, el cliente debe volver a intentarlo y, si las nuevas solicitudes se realizan correctamente, desactivar el interruptor.

Proporcionar reservas. En este enfoque, el proceso del cliente realiza una lógica de reserva cuando falla una solicitud, como devolver los datos almacenados en caché o un valor predeterminado. Este enfoque es adecuado para las consultas, pero es más complejo para las actualizaciones o los comandos.

Limitar el número de solicitudes en cola. Los clientes también deben imponer un límite máximo en la cantidad de solicitudes pendientes que un microservicio de cliente puede enviar a un servicio determinado. Si se alcanza el límite, probablemente no tenga sentido realizar más solicitudes y dichos intentos deben generar 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 mucho más rápida que la que tendría un interruptor, puesto que el interruptor espera a que se produzcan los errores. 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

- [Resiliency patterns \(Patrones de resistencia\)](#)

<https://docs.microsoft.com/azure/architecture/patterns/category/resiliency>

- **Adding Resilience and Optimizing Performance (Agregar resistencia y optimizar el rendimiento)**
[https://docs.microsoft.com/previous-versions/msp-n-p/jj591574\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-n-p/jj591574(v=pandp.10))
- **Bulkhead (Mampara).** Repositorio de GitHub. Implementación con la directiva Polly.
<https://github.com/App-vNext/Polly/wiki/Bulkhead>
- **Designing resilient applications for Azure (Diseñar aplicaciones resistentes de Azure)**
<https://docs.microsoft.com/azure/architecture/resiliency/>
- **Control de errores transitorios**
<https://docs.microsoft.com/azure/architecture/best-practices/transient-faults>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementar reintentos con retroceso exponencial

23/10/2019 • 2 minutes to read • [Edit Online](#)

Los *reintentos con retroceso exponencial* son una técnica que reintenta una operación, con un tiempo de espera que aumenta exponencialmente, hasta que se alcanza un número máximo de reintentos (el [retroceso exponencial](#)). Esta técnica se basa en el hecho de que los recursos en la nube pueden no estar disponibles de forma intermitente durante más de unos segundos por cualquier motivo. Por ejemplo, un orquestador puede mover un contenedor a otro nodo de un clúster para el equilibrio de carga. Durante ese tiempo se podrían producir errores en algunas solicitudes. Otro ejemplo podría ser una base de datos como SQL Azure, que puede moverse a otro servidor para el equilibrio de carga, lo que haría que la base de datos no estuviera disponible durante unos segundos.

Existen muchos enfoques para implementar la lógica de reintentos con retroceso exponencial.

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación de conexiones SQL resistentes de Entity Framework Core

04/11/2019 • 5 minutes to read • [Edit Online](#)

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.

```
// Startup.cs from any ASP.NET Core Web API
public class Startup
{
    // Other code ...
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddDbContext<CatalogContext>(options =>
        {
            options.UseSqlServer(Configuration["ConnectionString"],
                sqlServerOptionsAction: sqlOptions =>
            {
                sqlOptions.EnableRetryOnFailure(
                    maxRetryCount: 10,
                    maxRetryDelay: TimeSpan.FromSeconds(30),
                    errorNumbersToAdd: null);
            });
        });
    }
    //...
}
```

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.

```
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 `_integrationEventLogService`. 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:

```

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://docs.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.

```

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://docs.microsoft.com/ef/core/miscellaneous/connection-resiliency
        var strategy = _context.Database.CreateExecutionStrategy();
        await strategy.ExecuteAsync(async () =>
    {
        using (var transaction = _context.Database.BeginTransaction())
        {
            await action();
            transaction.Commit();
        }
    });
}
}

```

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://docs.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](#)

Exploración de reintentos de llamada HTTP personalizados con retroceso exponencial

23/10/2019 • 3 minutes to read • [Edit Online](#)

Para crear microservicios resistentes, debe controlar los posibles escenarios de error HTTP. Una manera de controlar esos errores, aunque no se recomienda, consiste en crear una implementación de reintentos propia con retroceso exponencial.

Nota importante: En esta sección se muestra cómo se puede crear código personalizado propio para implementar los reintentos de llamada HTTP. Pero no se recomienda hacerlo por su cuenta, sino usar mecanismos más eficaces y confiables, aunque más sencillos, como `HttpClientFactory` con Polly, disponible desde .NET Core 2.1. Esos enfoques recomendados se explican en las secciones siguientes.

Como exploración inicial, podría implementar su propio código con una clase de utilidad para retroceso exponencial como [RetryWithExponentialBackoff.cs](#), junto con código similar al siguiente.

```
public sealed class RetryWithExponentialBackoff
{
    private readonly int maxRetries, delayMilliseconds, maxDelayMilliseconds;

    public RetryWithExponentialBackoff(int maxRetries = 50,
        int delayMilliseconds = 200,
        int maxDelayMilliseconds = 2000)
    {
        this.maxRetries = maxRetries;
        this.delayMilliseconds = delayMilliseconds;
        this.maxDelayMilliseconds = maxDelayMilliseconds;
    }

    public async Task RunAsync(Func<Task> func)
    {
        ExponentialBackoff backoff = new ExponentialBackoff(this.maxRetries,
            this.delayMilliseconds,
            this.maxDelayMilliseconds);

        retry:
        try
        {
            await func();
        }
        catch (Exception ex) when (ex is TimeoutException ||
            ex is System.Net.Http.HttpRequestException)
        {
            Debug.WriteLine("Exception raised is: " +
                ex.GetType().ToString() +
                " -Message: " + ex.Message +
                " -- Inner Message: " +
                ex.InnerException.Message);
            await backoff.Delay();
            goto retry;
        }
    }
}

public struct ExponentialBackoff
{
    private readonly int m_maxRetries, m_delayMilliseconds, m_maxDelayMilliseconds;
    private int m_retries, m_pow;

    public ExponentialBackoff(int maxRetries, int delayMilliseconds,
```

```

        int maxDelayMilliseconds)
{
    m_maxRetries = maxRetries;
    m_delayMilliseconds = delayMilliseconds;
    m_maxDelayMilliseconds = maxDelayMilliseconds;
    m_retries = 0;
    m_pow = 1;
}

public Task Delay()
{
    if (m_retries == m_maxRetries)
    {
        throw new TimeoutException("Max retry attempts exceeded.");
    }
    ++m_retries;
    if (m_retries < 31)
    {
        m_pow = m_pow << 1; // m_pow = Pow(2, m_retries - 1)
    }
    int delay = Math.Min(m_delayMilliseconds * (m_pow - 1) / 2,
        m_maxDelayMilliseconds);
    return Task.Delay(delay);
}
}

```

Usar este código en una aplicación de cliente C# (otro microservicio de cliente API Web, una aplicación ASP.NET MVC o incluso una aplicación Xamarin C#) es sencillo. En el ejemplo siguiente se muestra cómo hacerlo, mediante la clase `HttpClient`.

```

public async Task<Catalog> GetCatalogItems(int page,int take, int? brand, int? type)
{
    _apiClient = new HttpClient();
    var itemsQs = $"items?pageIndex={page}&pageSize={take}";
    var filterQs = "";
    var catalogUrl = $"{_remoteServiceBaseUrl}items{filterQs}?pageIndex={page}&pageSize={take}";
    var dataString = "";
    //
    // Using HttpClient with Retry and Exponential Backoff
    //
    var retry = new RetryWithExponentialBackoff();
    await retry.RunAsync(async () =>
    {
        // work with HttpClient call
        dataString = await _apiClient.GetStringAsync(catalogUrl);
    });
    return JsonConvert.DeserializeObject<Catalog>(dataString);
}

```

Recuerde que este código solo es adecuado como prueba de concepto. En las secciones siguientes se explica cómo usar enfoques más sofisticados, aunque más sencillos, con `HttpClientFactory`. `HttpClientFactory` está disponible desde .NET Core 2.1, con bibliotecas de resistencia de eficacia probada, como Polly.

[ANTERIOR](#)

[SIGUIENTE](#)

Uso de HttpClientFactory para implementar solicitudes HTTP resistentes

25/11/2019 • 13 minutes to read • [Edit Online](#)

`HttpClientFactory` es 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 Core

La clase `HttpClient` original y bien conocida se puede usar fácilmente pero, en algunos casos, muchos desarrolladores no la usan de manera correcta.

Como primer problema, aunque esta clase es descartable, usarla con la instrucción `using` no es la mejor opción porque incluso cuando se descarta el objeto `HttpClient`, el socket subyacente no se libera de forma inmediata y puede causar un problema grave denominado "agotamiento de socket". Para obtener más información sobre este problema, vea la entrada de blog [You're using HttpClient wrong and it is destabilizing your software](#) (Está usando `HttpClient` mal y eso desestabiliza el software).

Por tanto, `HttpClient` está diseñado para que se cree una instancia una vez y se reutilice durante la vida 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 enfoques posibles para solucionar 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](#).

Pero hay un segundo problema con `HttpClient` que puede aparecer cuando se usa como objeto singleton o estático. En este caso, un `HttpClient` singleton o estático no respeta los cambios de DNS, tal como se explica en este [problema](#) en el repositorio dotnet/corefx de GitHub.

Para resolver esos problemas mencionados y facilitar la administración de las instancias de `HttpClient`, .NET Core 2.1 ofrece un nuevo `HttpClientFactory` que también se puede usar para implementar llamadas HTTP resistentes si se le integra Polly.

[Polly](#) es una biblioteca de control de errores transitorios que ayuda a los desarrolladores a agregar resistencia a sus aplicaciones mediante el uso de directivas predefinidas de manera fluida y segura para subprocessos.

Qué es HttpClientFactory

`HttpClientFactory` está diseñado para:

- Proporcionar una ubicación central para denominar y configurar instancias lógicas de `HttpClient`. Por ejemplo, puede configurar un cliente (Agente de servicio) preconfigurado para acceder a un microservicio concreto.
- Codifique el concepto de software intermedio de salida a través de controladores de delegación en `HttpClient` e implemente software intermedio basado en Polly para aprovechar las directivas de resistencia de Polly.
- `HttpClient` ya posee el concepto de controladores de delegación, que se pueden vincular entre sí para las solicitudes HTTP salientes. Los clientes HTTP se registran 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 `HttpClientMessageHandlers` para evitar los problemas mencionados y los que se puedan producir al administrar las duraciones de `HttpClient` usted mismo.

NOTE

`HttpClientFactory` está estrechamente ligado a la implementación de la inserción de dependencias (DI) en el paquete de NuGet `Microsoft.Extensions.DependencyInjection`. Para más información sobre el uso de otros contenedores de inserción de dependencias, consulte esta [conversación de GitHub](#).

Varias formas de usar `HttpClientFactory`

Hay varias formas de usar `HttpClientFactory` en la aplicación:

- Usar `HttpClientFactory` directamente.
- Usar clientes con nombre.
- Usar clientes con tipo.
- Usar clientes generados.

En pro de la brevedad, esta guía muestra la manera más estructurada para usar `HttpClientFactory`, que consiste en usar clientes con tipo (el patrón de agente de servicio). Sin embargo, todas las opciones están documentadas y actualmente se muestra en este [artículo que trata sobre el uso de `HttpClientFactory`](#).

Cómo usar clientes con tipo con `HttpClientFactory`

Así pues, ¿qué es un "Cliente con tipo"? Se trata sencillamente de un cliente `HttpClient` configurado por `DefaultHttpClientFactory` tras la inserción.

En el diagrama siguiente se muestra cómo se usan los clientes con tipo con `HttpClientFactory`:

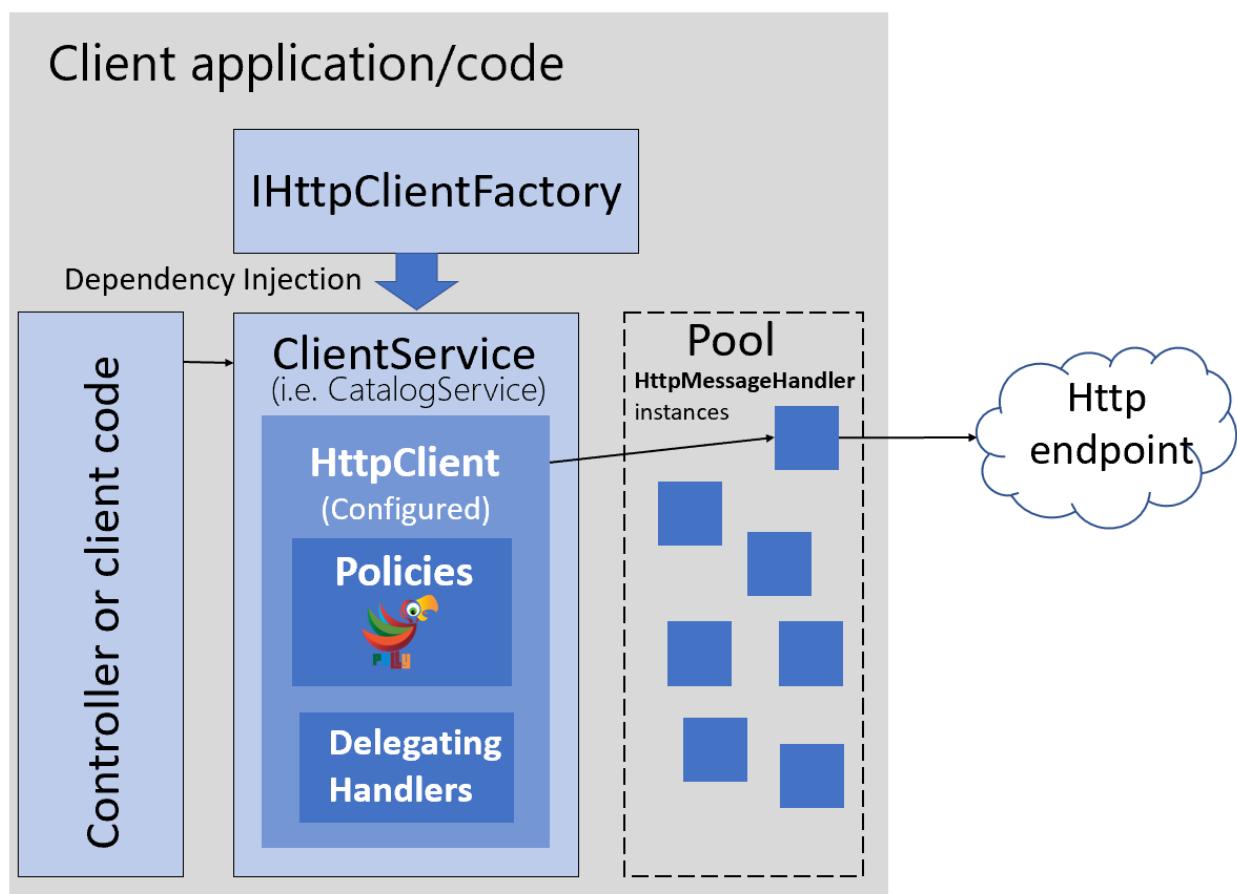


Figura 8-4. Uso de `HttpClientFactory` con clases de cliente con tipo.

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 `HttpMessageHandler` de un grupo que administra. 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` a la aplicación mediante la instalación del paquete de NuGet `Microsoft.Extensions.Http`, que incluye el método de extensión `AddHttpClient()` para `IServiceCollection`. Este método de extensión registra el `DefaultHttpClientFactory` que se va a usar como singleton para la interfaz `IHttpClientFactory`. Define una configuración transitoria para `HttpMessageHandlerBuilder`. Este controlador de mensajes (el objeto `HttpMessageHandler`), tomado de un grupo, lo usa el `HttpClient` devuelto desde la fábrica.

En el código siguiente, puede ver cómo se puede `AddHttpClient()` utilizar para registrar clientes con tipo (agentes de servicio) que necesitan usar `HttpClient`.

```
// Startup.cs
//Add http client services at ConfigureServices(IServiceCollection services)
services.AddHttpClient<ICatalogService, CatalogService>();
services.AddHttpClient<IBasketService, BasketService>();
services.AddHttpClient<IOrderingService, OrderingService>();
```

Al registrar los servicios de cliente como se muestra en el código anterior, `DefaultClientFactory` crea un `HttpClient` estándar para cada servicio.

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 el código siguiente:

```
services.AddHttpClient<ICatalogService, CatalogService>(client =>
{
    client.BaseAddress = new Uri(Configuration["BaseUrl"]);
})
.AddPolicyHandler(GetRetryPolicy())
.AddPolicyHandler(GetCircuitBreakerPolicy());
```

Solo para el ejemplo, puede ver una de las directivas anteriores en el código siguiente:

```
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](#).

Duraciones de HttpClient

Cada vez que se obtiene un objeto `HttpClient` de `IHttpClientFactory`, se devuelve una nueva instancia. Pero cada cliente `HttpClient` usa un controlador `HttpMessageHandler` que `IHttpClientFactory` agrupa y vuelve a usar para reducir el consumo de recursos, siempre y cuando la vigencia de `HttpMessageHandler` no haya expirado.

La agrupación de controladores es conveniente porque cada controlador suele administrar sus propias conexiones HTTP subyacentes. Crear más controladores de lo necesario puede provocar retrasos en la conexión. Además, algunos controladores dejan las conexiones abiertas de forma indefinida, lo que puede ser un obstáculo a la hora de reaccionar ante 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:

```
//Set 5 min as the lifetime for the HttpMessageHandler objects in the pool used for the Catalog Typed Client
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 `Infinite TimeSpan` para deshabilitar la expiración del controlador.

Implementar las clases de cliente con tipo que usan el HttpClient insertado y configurado

Como paso anterior, debe tener las clases de cliente con tipo definidas, como las del código de ejemplo (por ejemplo, "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 remoto de HTTP.

Por ejemplo:

```
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 cliente con tipo (`CatalogService` en el ejemplo) se activa mediante la inserción de dependencias, lo que significa que puede aceptar cualquier servicio registrado en su constructor, además de `HttpClient`.

Un cliente con tipo es, de hecho, un objeto transitorio, lo que significa que se crea una instancia cada vez que se necesita una y que recibirá una instancia de `HttpClient` nueva cada vez se construya. Pero los objetos `HttpMessageHandler` del grupo son los que reutilizan varias solicitudes HTTP.

Usar las clases de cliente con tipo

Por último, una vez que las clases con tipo se implementan y se registran con `AddHttpClient()`, se pueden usar en cualquier lugar donde haya servicios insertados mediante la inserción de dependencias, por ejemplo, en cualquier código de Razor Pages o cualquier controlador de una aplicación web MVC, como en el código siguiente de eShopOnContainers:

```

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 el momento, el código que se ha mostrado solo realiza solicitudes HTTP convencionales, pero la "magia" aparecerá en las secciones siguientes donde, con tan solo agregar directivas y controladores de delegación a los clientes con tipo registrados, todas las solicitudes HTTP que `HttpClient` va a realizar se comportarán teniendo en cuenta las directivas de resistencia como los reintentos con retroceso exponencial, los interruptores o cualquier otro controlador de delegación personalizado para implementar características de seguridad adicionales, como el uso de tokens de autenticación, o bien cualquier otra característica personalizada.

Recursos adicionales

- **Uso de `HttpClientFactory` en .NET Core**
<https://docs.microsoft.com/aspnet/core/fundamentals/http-requests>
- **Código fuente de `HttpClientFactory` en el repositorio de GitHub** `aspnet/Extensions`
<https://github.com/aspnet/Extensions/tree/master/src/HttpClientFactory>
- **Polly (.NET resilience and transient-fault-handling library) (Polly [Biblioteca de control de errores transitorios y resistencia de .NET])**
<http://www.thepollyproject.org/>
- **Uso de `HttpClientFactory` sin inserción de dependencias (problema de GitHub)**
<https://github.com/aspnet/Extensions/issues/1345>

[ANTERIOR](#)

[SIGUIENTE](#)

Implementación de reintentos de llamada HTTP con retroceso exponencial con HttpClientFactory y las directivas de Polly

04/11/2019 • 5 minutes to read • [Edit Online](#)

El enfoque recomendado para los reintentos con retroceso exponencial consiste en aprovechar las ventajas de las bibliotecas de .NET más avanzadas como la biblioteca de código abierto [Polly](#).

Polly es una biblioteca de .NET que proporciona capacidades de resistencia y control de errores transitorios. Puede implementar esas funcionalidades mediante la aplicación de directivas de Polly como las de reinicio, interruptor, aislamiento compartimentado, tiempo de espera y reserva. Polly tiene como destino .NET Framework 4.x y .NET Standard 1.0, 1.1 y 2.0 (que admite .NET Core).

Sin embargo, escribir su propio código personalizado para usar la biblioteca de Polly's con HttpClient puede ser bastante complejo. En la versión original de eShopOnContainers, había un [bloque de creación ResilientHttpClient](#) basado en Polly. Pero con el lanzamiento de [HttpClientFactory](#), la implementación de la comunicación HTTP resistente con Polly se ha convertido en un proceso mucho más sencillo, por lo que ese bloque de creación ha quedado en desuso en eShopOnContainers.

En los pasos siguientes se muestra cómo usar reintentos HTTP con Polly integrados en HttpClientFactory, que se explica en la sección anterior.

Hacer referencia a los paquetes de ASP.NET Core 2.2

`HttpClientFactory` está disponible desde .NET Core 2.1. Sin embargo, le recomendamos que use los últimos paquetes de ASP.NET Core 2.2 de NuGet en su proyecto. Normalmente se necesita el metapquete `AspNetCore` y el paquete de extensión `Microsoft.Extensions.Http.Polly`.

Configurar un cliente con la directiva de reintentos de Polly, en Startup

Como se mostró en las secciones anteriores, tendrá que definir una configuración HttpClient cliente con nombre o tipo en el método `Startup.ConfigureServices(...)` estándar, pero ahora agregará código incremental en el que se especifica la directiva para los reintentos HTTP con retroceso exponencial, como se muestra a continuación:

```
//ConfigureServices() - Startup.cs
services.AddHttpClient<IBasketService, BasketService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5)) //Set lifetime to five minutes
    .AddPolicyHandler(GetRetryPolicy());
```

El método **AddPolicyHandler()** es el que agrega las directivas a los objetos `HttpClient` que se van a usar. En este caso, se agrega una directiva de Polly para reintentos HTTP con retroceso exponencial.

Para tener un enfoque más modular, la directiva de reintentos HTTP se puede definir en un método independiente dentro del archivo `Startup.cs`, como se muestra en el código siguiente:

```

static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
            retryAttempt)));
}

```

Con Polly, se puede definir una directiva de reintentos con el número de reintentos, la configuración de retroceso exponencial y las acciones necesarias cuando se produce una excepción de HTTP, como registrar el error. En este caso, la directiva está configurada para intentar seis veces con un reintento exponencial, a partir de 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 caso de interrupciones parciales, una buena solución es agregar una estrategia de vibración a la directiva o algoritmo de reintento. Esto puede mejorar el rendimiento general del sistema de un extremo a otro añadiendo aleatoriedad al retroceso exponencial. De esta forma, cuando surgen problemas, los picos se reparten. El principio que rige esto se muestra en el ejemplo siguiente:

```

Random jitterer = new Random();
var retryWithJitterPolicy = HttpPolicyExtensions
    .HandleTransientHttpError()
    .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
    .WaitAndRetryAsync(6, // exponential back-off plus some jitter
        retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
            + TimeSpan.FromMilliseconds(jitterer.Next(0, 100)))
    );

```

Polly proporciona algoritmos de vibración listos para la producción a través del sitio web del proyecto.

Recursos adicionales

- **Patrón de reintento**

<https://docs.microsoft.com/azure/architecture/patterns/retry>

- **Polly y HttpClientFactory**

<https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory>

- **Polly (.NET resilience and transient-fault-handling library) (Polly [Biblioteca de control de errores transitorios y resistencia 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>

Implementación del patrón de interruptor

25/11/2019 • 15 minutes to read • [Edit Online](#)

Tal y como se indicó anteriormente, debe controlar los errores que pueden comportar un tiempo variable de recuperación, como puede suceder al intentar conectarse a un recurso o servicio remoto. 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 causados por errores transitorios, como tiempos de espera y conexiones de red lentas, o si los recursos responden de forma lenta 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".

Pero también puede haber situaciones en que los errores se deban a eventos imprevistos que pueden tardar mucho más tiempo en corregirse. La gravedad de estos errores puede ir desde una pérdida parcial de conectividad hasta el fallo total del servicio. En estas situaciones, no tiene sentido que una aplicación reintente continuamente una operación que es probable que no se lleve a cabo correctamente.

Lo que debe hacer la aplicación es codificarse para aceptar que la operación ha fallado 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. Cuando se produce un error en un microservicio o se ejecuta lentamente, es posible que varios clientes reintenten solicitudes con error de forma repetida. Eso genera un riesgo peligroso de que el tráfico destinado al servicio con errores aumente de manera exponencial.

Por tanto, se necesita algún tipo de barrera de defensa para que se detengan las solicitudes excesivas cuando ya no tiene sentido seguir intentándolo. 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 que una aplicación reintente una operación con la expectativa de que finalmente se realice correctamente. El patrón de interruptor impide que una aplicación realice una operación que es probable que falle. Una aplicación puede combinar estos dos patrones. Pero la lógica de reintento debe ser sensible a las excepciones devueltas por el interruptor, y debe dejar de intentar repetir la operación si el interruptor indica que un error no es transitorio.

Implementar el patrón de interruptor con HttpClientFactory 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 HttpClientFactory.

Agregar una directiva de interruptor a la canalización de software intermedio saliente de HttpClientFactory es tan sencillo como agregar un único fragmento de código incremental a lo que ya tiene cuando se usa HttpClientFactory.

En este caso, lo único que se agrega al código que se usa para los reintentos de llamada HTTP es el código en el que se agrega la directiva de interruptor a la lista de directivas que se van a usar, como se muestra en el código incremental siguiente, parte del método ConfigureServices().

```
//ConfigureServices() - Startup.cs
services.AddHttpClient<IBasketService, BasketService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5)) //Sample. Default lifetime is 2 minutes
    .AddHttpMessageHandler<HttpClientAuthorizationDelegatingHandler>()
    .AddPolicyHandler(GetRetryPolicy())
    .AddPolicyHandler(GetCircuitBreakerPolicy());
```

El método `AddPolicyHandler()` es el que agrega las directivas a los objetos `HttpClient` que se van a usar. En este caso, se agrega una directiva de Polly para un interruptor.

Para tener un enfoque más modular, la directiva de interruptor se define en un método independiente denominado `GetCircuitBreakerPolicy()`, como se muestra en el código siguiente:

```
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 relevantes 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 del uso, al utilizar `HttpClient` no hay necesidad de agregar nada nuevo aquí porque el código es el mismo que cuando se usa `HttpClient` con `HttpClientFactory`, como se mostró en las secciones anteriores.

Prueba de reintentos HTTP e interruptores en eShopOnContainers

Cada vez que inicie la solución eShopOnContainers en un host Docker, debe iniciar varios contenedores. Algunos de los contenedores tardan más en iniciarse e inicializarse, como el contenedor de SQL Server. Esto sucede especialmente la primera vez que implementa la aplicación eShopOnContainers en Docker, porque las imágenes y la base de datos se tienen que configurar. El hecho de que algunos contenedores se inicien más lentamente que otros puede provocar que el resto de servicios lancen inicialmente excepciones HTTP, aunque configure las dependencias entre contenedores en el nivel de Docker Compose, como se ha explicado en las secciones anteriores. Las dependencias de Docker Compose entre contenedores solo se dan en el nivel de proceso. El proceso de punto de entrada del contenedor se puede iniciar, pero podría ser que SQL Server no estuviera listo para las consultas. El resultado puede ser una cascada de errores y la aplicación puede obtener una excepción al intentar utilizar dicho contenedor.

Este tipo de error también puede darse en el inicio, cuando la aplicación se está implementando en la nube. En ese caso, podría ser que los orquestadores movieran los contenedores de un nodo o máquina virtual a otro (iniciando

así nuevas instancias) al repartir equitativamente los contenedores entre los nodos de clúster.

La forma en que estos problemas se solucionan al iniciar todos los contenedores en "eShopOnContainers" es mediante el patrón de reintento mostrado anteriormente.

Prueba del interruptor en eShopOnContainers

Hay varias formas de interrumpir y abrir el circuito, y probarlo con eShopOnContainers.

Una opción es reducir el número permitido de reintentos a 1 en la directiva del interruptor y volver a implementar la solución completa 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 consiste en usar middleware personalizado que se implemente en el microservicio **Basket**. Al habilitar este middleware, 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, la solicitud devuelve el código de estado 500. Si el middleware está deshabilitado, no se emite ninguna 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, cuando la aplicación se está ejecutando, puede habilitar el middleware realizando una solicitud con el siguiente URI en cualquier explorador. Tenga en cuenta que el microservicio de ordenación utiliza el puerto 5103.

`http://localhost:5103/failing?enable`

Después, puede comprobar el estado mediante el URI `http://localhost:5103/failing`, como se muestra en la Figura 8-5.

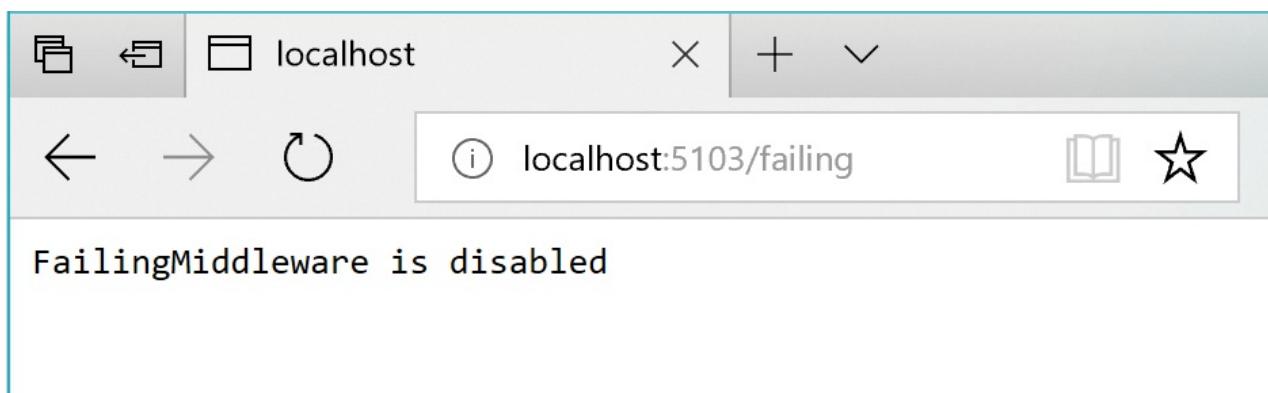


Figura 8-5. Comprobación del estado del middleware ASP.NET "Error": en este caso, deshabilitado

En este punto, el microservicio de la cesta responde con el código de estado 500 siempre que su llamada lo invoque.

Cuando se esté ejecutando 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 un mensaje descriptivo al usuario en que se le indica que espere.

```

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)";
    }
}

```

Aquí tiene un resumen. La directiva de reintentos intenta realizar la solicitud HTTP varias veces y obtiene 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 descriptivo, como el que se muestra en la Figura 8-6.

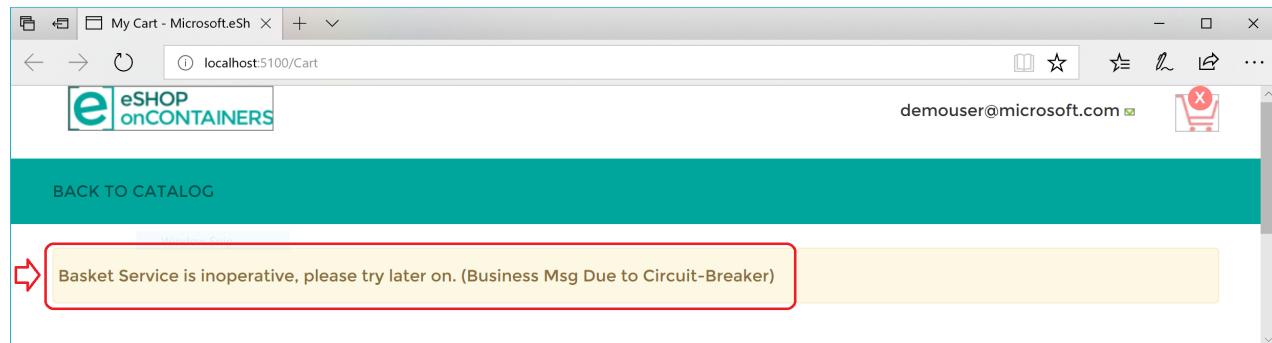


Figura 8-6. Interruptor que devuelve un error en la interfaz de usuario

Puede implementar otra lógica que indique cuándo se debe 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.

Por último, otra posibilidad para `CircuitBreakerPolicy` consiste en usar `Isolate` (que fuerza y mantiene la apertura del circuito) y `Reset` (que lo cierra de nuevo). Estas características se pueden utilizar para crear un punto de conexión HTTP de utilidad que invoque Aislamiento y Restablecer directamente en la directiva. Este tipo de punto de conexión HTTP, protegido adecuadamente, también se puede usar en el entorno de producción para aislar temporalmente un sistema de nivel inferior, como cuando quiere actualizarlo. También puede activar el circuito manualmente para proteger un sistema de nivel inferior que le parezca que está fallando.

Recursos adicionales

- **Circuit Breaker pattern (Patrón de interruptor)**

<https://docs.microsoft.com/azure/architecture/patterns/circuit-breaker>

[ANTERIOR](#)

[SIGUIENTE](#)

Seguimiento de estado

18/12/2019 • 19 minutes to read • [Edit Online](#)

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. Esta 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. 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 usar la característica HealthChecks en una aplicación de la ASP.NET Core 2.2 Web API de ejemplo. La implementación de esta característica en un microservicio a gran escala como eShopOnContainers se explica en la sección posterior. Para empezar, debe definir qué constituye un estado correcto en cada microservicio. En la aplicación de ejemplo, el estado de los microservicios es correcto si se puede acceder a la API del microservicio a través de HTTP y si su base de datos de SQL Server relacionada también está disponible.

En .NET Core 2.2, con las API integradas, puede configurar los servicios, añadir una comprobación de estado para el microservicio y su base de datos de SQL Server dependiente de esta forma:

```
// Startup.cs from .NET Core 2.2 Web API sample
//
public void ConfigureServices(IServiceCollection services)
{
    //...
    // Registers required services for health checks
    services.AddHealthChecks()
        // Add a health check for a SQL database
        .AddCheck("MyDatabase", new
    SqlConnectionHealthCheck(Configuration["ConnectionStrings:DefaultConnection"]));
}
```

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.

```

// Sample SQL Connection Health Check
public class SqlConnectionHealthCheck : IHealthCheck
{
    private static readonly 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();
    }
}

```

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 y Service Fabric 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, cree un middleware que responda a la dirección URL "/hc":

```

// Startup.cs from .NET Core 2.2 Web API sample
//
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...
    app.UseHealthChecks("/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()`. Para todos los servicios dependientes, debe agregarse una implementación `IHealthCheck` que defina su estado de mantenimiento correspondiente.

El proyecto de código abierto [AspNetCore.Diagnostics.HealthChecks](#) resuelve este problema proporcionando implementaciones de comprobación de estado personalizadas para cada uno de estos servicios empresariales basados en .NET Core 2.2. 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:

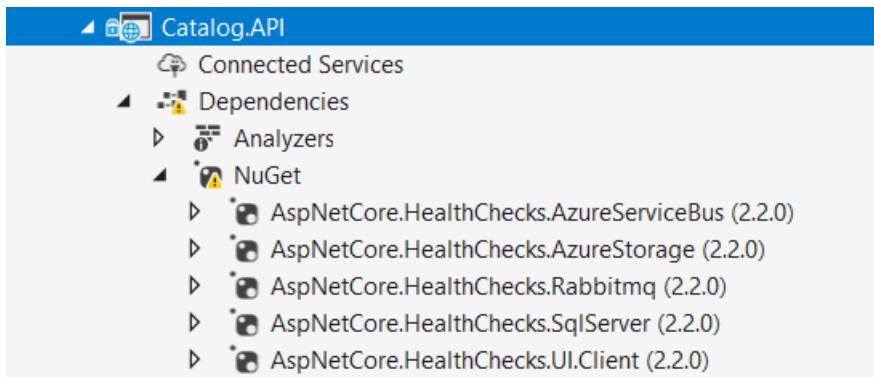


Figura 8-7. Comprobaciones de estado personalizadas implementadas en Catalog.API mediante AspNetCore.Diagnostics.HealthChecks

En el siguiente código, las implementaciones de comprobación de estado se agregan para cada servicio dependiente y, a continuación, se configura el middleware:

```

// Startup.cs 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, agregamos el middleware HealthCheck que se va a escuchar al punto de conexión "/hc":

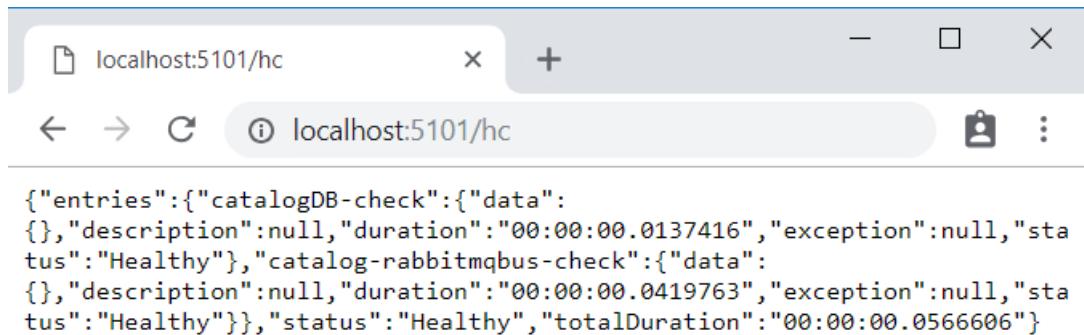
```

// 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 `localhost`, como se muestra en la figura 8-8.



A screenshot of a web browser window. The address bar shows 'localhost:5101/hc'. The main content area displays a JSON object representing the health check results:

```
{"entries": {"catalogDB-check": {"data": {}, "description": null, "duration": "00:00:00.0137416", "exception": null, "status": "Healthy"}, "catalog-rabbitmqbus-check": {"data": {}, "description": null, "duration": "00:00:00.0419763", "exception": null, "status": "Healthy"}}, "status": "Healthy", "totalDuration": "00:00:00.0566606"}
```

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.

Name	Health	On state from	Last execution
Ordering HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:47
Locations HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:38
Marketing HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:39
Identity HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:40
Catalog HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:42

Name	Health	Description	Duration
catalogDB-check	✓ Healthy		00:00:00.0174990
catalog-rabbitmqbus-check	✓ Healthy		00:00:00.0939750

Name	Health	Description	Duration
Basket HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:44
Ordering HTTP Background Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:46
Ordering HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:47
Web Marketing API GW HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:53
Web Shopping API GW HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:48:59
Mobile Marketing API GW HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:49:05
Mobile Shopping API GW HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:49:10
Mobile Shopping Aggregator HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:49:16
Web Shopping Aggregator GW HTTP Check	✓	Healthy 15 minutes ago	11/12/2018, 11:49:20
WebSPA HTTP Check	✓	Healthy 14 minutes ago	11/12/2018, 11:49:30
WebMVC HTTP Check	✓	Healthy 14 minutes ago	11/12/2018, 11:49:38
Payments HTTP Check	✓	Healthy 14 minutes ago	11/12/2018, 11:49:40
Ordering SignalRHub HTTP Check	✓	Healthy 14 minutes ago	11/12/2018, 11:49:41

Xabari Team @ 2018

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. La 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 vigilancia.

Archivo de configuración de ejemplo para la interfaz de usuario de comprobación de estado:

```
// Configuration
{
  "HealthChecks-UI": {
    "HealthChecks": [
      {
        "Name": "Ordering HTTP Check",
        "Uri": "http://localhost:5102/hc"
      },
      {
        "Name": "Ordering HTTP Background Check",
        "Uri": "http://localhost:5111/hc"
      },
      //...
    ]
  }
}
```

Archivo Startup.cs que agrega HealthChecksUI:

```
// Startup.cs from WebStatus(Watch Dog) service
//
public void ConfigureServices(IServiceCollection services)
{
    ...
    // Registers required services for health checks
    services.AddHealthChecksUI();
}
//...
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...
    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://docs.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 .NET

12/11/2019 • 21 minutes to read • [Edit Online](#)

Hay tantos aspectos sobre la seguridad en los microservicios y las aplicaciones web que fácilmente se podrían dedicar al tema varios libros como este por lo que, en esta sección, nos centraremos en la autenticación, la autorización y los secretos de aplicación.

Implementación de la autenticación en microservicios y aplicaciones web .NET

A menudo es necesario que los recursos y las API publicados por un servicio se limiten a determinados usuarios o clientes de confianza. El primer paso para tomar este tipo de decisiones de confianza en el nivel de API es la autenticación. La autenticación es el proceso de comprobar de forma fiable la identidad de un usuario.

En escenarios de microservicios, la autenticación suele controlarse de manera centralizada. Si usa una puerta de enlace de API, esa puerta es un buen lugar para realizar la autenticación, como se muestra en la figura 9-1. Si emplea este método, asegúrese de que no es posible ponerse en contacto directamente con los microservicios individuales (sin la puerta de enlace de API), a menos que haya aplicado seguridad adicional para autenticar si los mensajes provienen de la puerta de enlace.

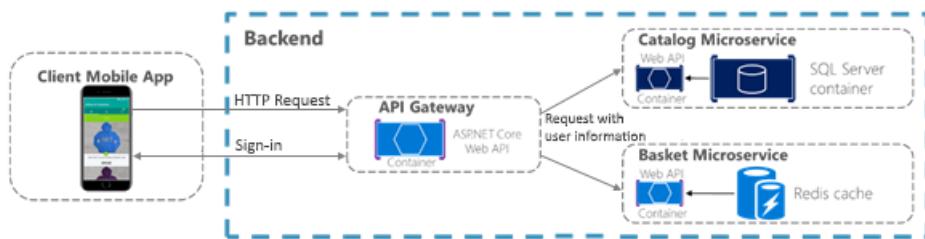


Figura 9-1. Autenticación centralizada con una puerta de enlace de API.

Cuando la puerta de enlace de API centraliza la autenticación, agrega información de usuario al reenviar las solicitudes a los microservicios. Si se puede tener acceso directamente a los servicios, entonces para la autenticación de los usuarios se puede usar un servicio de autenticación como Azure Active Directory o un microservicio de autenticación dedicado que actúe como un servicio de token de seguridad (STS). Las decisiones de confianza se comparten entre los servicios con tokens de seguridad o cookies. (Si es necesario, estos tokens se pueden compartir entre aplicaciones de ASP.NET Core mediante el [uso compartido de cookies](#)). Este patrón se ilustra en la figura 9-2.

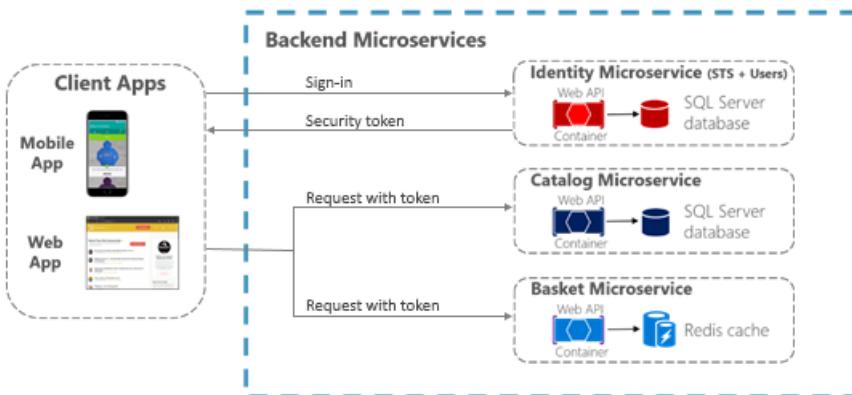


Figura 9-2. Autenticación realizada por un microservicio de identidad; la confianza se comparte mediante un token de autorización.

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, que se comparte entre los microservicios.

Autenticación con ASP.NET Core Identity

El principal mecanismo de ASP.NET Core para identificar a los usuarios de una aplicación es el sistema de pertenencia [ASP.NET Core Identity](#). ASP.NET Core Identity almacena la información del usuario (incluida la información de inicio de sesión, los roles y las 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 incluido en el paquete `Microsoft.AspNetCore.Identity.EntityFrameworkCore`. Pero se pueden usar almacenes personalizados u otros paquetes de terceros para almacenar información de identidad en Table Storage de Azure, Cosmos DB u otras ubicaciones.

El código siguiente procede de la plantilla de proyecto de aplicación web de ASP.NET Core con la autenticación de cuentas de usuario individuales seleccionada. Muestra cómo configurar ASP.NET Core Identity mediante `EntityFrameworkCore` en el método `Startup.ConfigureServices`.

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddIdentity< ApplicationUser, IdentityRole >()
    .AddEntityFrameworkStores< ApplicationDbContext >()
    .AddDefaultTokenProviders();
```

Una vez configurado ASP.NET Core Identity, para habilitarlo llame a `app.UseIdentity` en el método `Startup.Configure` del servicio.

El uso de ASP.NET Core Identity permite varios escenarios:

- Crear información de usuario con el tipo `UserManager` (`userManager.CreateAsync`).
- Autenticar a los usuarios con el tipo `SignInManager`. Puede usar `signInManager.SignInAsync` para iniciar sesión directamente, o bien `signInManager.PasswordSignInAsync` para confirmar que la contraseña del usuario es correcta y, después, iniciar su sesión.
- Identificar a un usuario en función de la información almacenada en una cookie (que se lee mediante el software intermedio de ASP.NET Core Identity), de modo que las solicitudes posteriores desde un explorador incluyan la identidad y las notificaciones del usuario que ha iniciado sesión.

ASP.NET Core Identity también es compatible con la [autenticación en dos fases](#).

ASP.NET Core Identity es una solución recomendada para los escenarios de autenticación que usan un almacén de datos de usuario local y que conservan la identidad entre las solicitudes mediante el uso de cookies (como es habitual en las aplicaciones web MVC).

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 [OAuth 2.0](#). Esto significa que los usuarios pueden iniciar sesión mediante los procesos de autenticación existentes de proveedores como Microsoft, Google, Facebook o Twitter, y asociar esas identidades con una identidad de ASP.NET Core en la aplicación.

Para usar la autenticación externa, incluya el software intermedio de autenticación adecuado en la canalización de procesamiento de solicitudes HTTP de la aplicación. Este software intermedio es responsable de controlar las solicitudes para devolver las rutas URI desde el proveedor de autenticación, capturar información de identidad y hacer que esté disponible mediante el método `SignInManager.GetExternalLoginInfo`.

En la tabla siguiente se muestran proveedores de autenticación externos populares y sus paquetes NuGet asociados:

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, el middleware se registra con una llamada a un método de registro similar a `app.Use{ExternalProvider}Authentication` en `Startup.Configure`. Estos métodos de registro toman un objeto de opciones que contiene un identificador de aplicación e información secreta (una contraseña, por ejemplo), según requiera el proveedor. Los proveedores de autenticación externos requieren que la aplicación se registre (como se explica en la [documentación de ASP.NET Core](#)) para que puedan informar al usuario sobre qué aplicación solicita acceso a su identidad.

Una vez que se haya registrado el middleware en `Startup.Configure`, podrá pedirles a los usuarios que inicien sesión desde cualquier acción de controlador. Para ello, cree un objeto `AuthenticationProperties` que incluya el nombre del proveedor de autenticación y una dirección URL de redireccionamiento. Después, devuelva una respuesta Challenge que pase el objeto `AuthenticationProperties`. El código siguiente muestra un ejemplo de esto.

```
var properties = _signInManager.ConfigureExternalAuthenticationProperties(provider,
    redirectUrl);
return Challenge(properties, provider);
```

El parámetro `redirectUrl` incluye la dirección URL a la que el proveedor externo debe redirigir una vez que se ha autenticado el usuario. La dirección URL debe representar una acción que iniciará la sesión del usuario en función de información de identidad externa, como en el siguiente ejemplo simplificado:

```

// Sign in the user with this external login provider if the user
// already has a login.
var result = await _signInManager.ExternalLoginSignInAsync(info.LoginProvider, info.ProviderKey, isPersistent:
false);

if (result.Succeeded)
{
    return RedirectToLocal(returnUrl);
}
else
{
    ApplicationUser newUser = new ApplicationUser
    {
        // The user object can be constructed with claims from the
        // external authentication provider, combined with information
        // supplied by the user after they have authenticated with
        // the external provider.
        UserName = info.Principal.FindFirstValue(ClaimTypes.Name),
        Email = info.Principal.FindFirstValue(ClaimTypes.Email)
    };
    var identityResult = await _userManager.CreateAsync(newUser);
    if (identityResult.Succeeded)
    {
        identityResult = await _userManager.AddLoginAsync(newUser, info);
        if (identityResult.Succeeded)
        {
            await _signInManager.SignInAsync(newUser, isPersistent: false);
        }
    }
    return RedirectToLocal(returnUrl);
}
}

```

Si elige la opción de autenticación **Cuenta de usuario individual** al crear el proyecto de aplicación web ASP.NET Core en Visual Studio, todo el código necesario para iniciar sesión con un proveedor externo ya está en el proyecto, como se muestra en la figura 9-3.

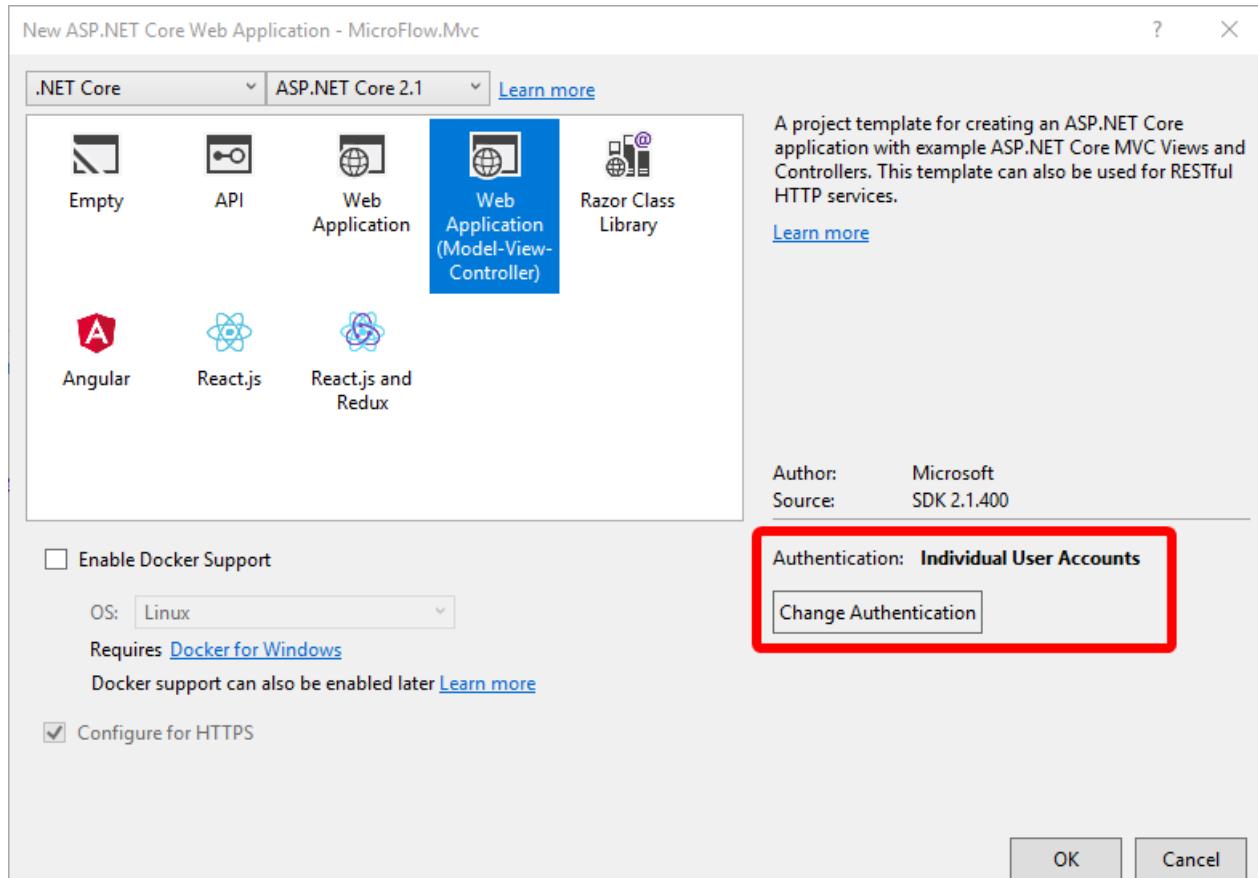


Figura 9-3. Proceso de selección de una opción para usar la autenticación externa al crear un proyecto de aplicación web.

Además de los proveedores de autenticación externa mencionados anteriormente, hay disponibles paquetes de terceros que proporcionan software intermedio para el uso de muchos otros proveedores de autenticación externos. Para obtener una lista, vea el repositorio [AspNet.Security.OAuth.Providers](#) en GitHub.

También puede crear middleware de autenticación externo propio para resolver alguna necesidad especial.

Autenticación con tokens de portador

La autenticación con ASP.NET Core Identity (o con Identity y proveedores de autenticación externos) funciona bien en muchos escenarios de aplicación web en los que es adecuado almacenar información de usuario en una cookie. En cambio, en otros escenarios las cookies no son una manera natural de conservar y transmitir datos.

Por ejemplo, en una Web API de ASP.NET Core que expone puntos de conexión RESTful a los que podrían tener acceso aplicaciones de una sola página (SPA), clientes nativos o incluso otras Web API, normalmente le interesa usar la autenticación mediante token de portador. 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. Con objeto de habilitar la autenticación mediante token, ASP.NET Core admite varias opciones para el uso de [OAuth 2.0](#) y [OpenID Connect](#).

Autenticación con un proveedor de identidad OpenID Connect u OAuth 2.0

Si la información de usuario se almacena en Azure Active Directory u otra solución de identidad compatible con OpenID Connect u OAuth 2.0, puede usar el paquete **Microsoft.AspNetCore.Authentication.OpenIdConnect** para autenticarse con el flujo de trabajo de OpenID Connect. Por ejemplo, para autenticarse en el microservicio Identity.Api de eShopOnContainers, una aplicación web ASP.NET Core puede usar el middleware de ese paquete como se muestra en el siguiente ejemplo simplificado de `Startup.cs`:

```

// Startup.cs

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    //...
    // Configure the pipeline to use authentication
    app.UseAuthentication();
    //...
    app.UseMvc();
}

public void ConfigureServices(IServiceCollection services)
{
    var identityUrl = Configuration.GetValue<string>("IdentityUrl");
    var callBackUrl = Configuration.GetValue<string>("CallBackUrl");

    // Add Authentication services

    services.AddAuthentication(options =>
    {
        options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme = OpenIdConnectDefaults.AuthenticationScheme;
    })
    .AddCookie()
    .AddOpenIdConnect(options =>
    {
        options.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        options.Authority = identityUrl;
        options.SignedOutRedirectUri = callBackUrl;
        options.ClientSecret = "secret";
        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");
    });
}

```

Tenga en cuenta que, cuando se usa este flujo de trabajo, el software intermedio de ASP.NET Core Identity no es necesario, porque el servicio de identidad controla el almacenamiento y la autenticación de la información del usuario.

Emisión de tokens de seguridad desde un servicio de ASP.NET Core

Si prefiere emitir tokens de seguridad para los usuarios locales de ASP.NET Core Identity en lugar de usar un proveedor de identidades externo, puede aprovechar algunas buenas bibliotecas de terceros.

[IdentityServer4](#) y [OpenIddict](#) son proveedores de OpenID Connect que se integran fácilmente con ASP.NET Core Identity y le permiten emitir tokens de seguridad desde un servicio de ASP.NET Core. En la [documentación de IdentityServer4](#) encontrará instrucciones detalladas para usar la biblioteca, pero los pasos básicos para emitir tokens con IdentityServer4 son los que se indican a continuación.

1. Llame a `app.UseIdentityServer` en el método `Startup.Configure` para agregar IdentityServer4 a la canalización de procesamiento de solicitudes HTTP de la aplicación. Esto permite a la biblioteca atender las solicitudes a los puntos de conexión de OpenID Connect y OAuth2 como `/connect/token`.
2. Configure IdentityServer4 en `Startup.ConfigureServices` mediante una llamada a `services.AddIdentityServer`.

3. Para configurar el servidor de identidades, establezca los datos siguientes:

- Las [credenciales](#) que se van a usar para la firma.
- Los [recursos de identidad y de API](#) a los que los usuarios podrían solicitar acceso:
 - Los recursos de API representan funciones o datos protegidos a los que los usuarios pueden tener acceso 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 (notificaciones) que se entregan a un cliente para identificar a un usuario. Las notificaciones pueden incluir el nombre de usuario, la dirección de correo electrónico, etc.
- Los [clientes](#) que se conectarán para solicitar tokens.
- El mecanismo de almacenamiento de la información de usuario, como [ASP.NET Core Identity](#) u otra alternativa.

Al especificar los clientes y los recursos que se van a usar en IdentityServer4, puede pasar una colección [IEnumerable<T>](#) del tipo adecuado a los métodos que toman almacenes de recursos o clientes en memoria. En escenarios más complejos, puede proporcionar tipos de proveedor de recursos o cliente mediante la inserción de dependencias.

En el ejemplo siguiente se muestra el aspecto que podría tener una configuración para que IdentityServer4 use clientes y recursos en memoria proporcionados por un tipo IClientStore personalizado:

```
// Add IdentityServer services
services.AddSingleton<IClientStore, CustomClientStore>();
services.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<ApplicationUser>();
```

Consumo de tokens de seguridad

La autenticación con un punto de conexión de OpenID Connect o mediante la emisión de tokens de seguridad propios se aplica a diversos escenarios. Pero ¿qué sucede si un servicio solo necesita limitar el acceso a los usuarios que tienen tokens de seguridad válidos proporcionados por otro servicio?

Para este escenario, el middleware de autenticación que controla los tokens JWT está disponible en el paquete **Microsoft.AspNetCore.Authentication.JwtBearer**. JWT es el acrónimo de "[JSON Web Token](#)" y es un formato común de token de seguridad (definido en RFC 7519) para la comunicación de notificaciones de seguridad. Un ejemplo simplificado de cómo usar el middleware para consumir esos tokens podría ser similar a este fragmento de código, tomado del microservicio Ordering.Api de eShopOnContainers.

```

// Startup.cs

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    //...
    // Configure the pipeline to use authentication
    app.UseAuthentication();
    //...
    app.UseMvc();
}

public void ConfigureServices(IServiceCollection services)
{
    var identityUrl = Configuration.GetValue<string>("IdentityUrl");

    // Add Authentication services

    services.AddAuthentication(options =>
    {
        options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

    }).AddJwtBearer(options =>
    {
        options.Authority = identityUrl;
        options.RequireHttpsMetadata = false;
        options.Audience = "orders";
    });
}

```

Los parámetros de este uso son los siguientes:

- `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 de emisión de tokens. El software intermedio de autenticación del portador de JWT usa este URI para obtener la clave pública que puede usarse para validar la firma del token. El middleware también confirma que el parámetro `iss` del token coincide con este URI.

Otro parámetro, `RequireHttpsMetadata`, resulta útil para la realización de pruebas; establezcalo en `false` para poder realizarlas en los entornos en los que no tiene certificados. En implementaciones reales, los tokens de portador de JWT siempre se deben pasar a través de HTTPS exclusivamente.

Con este software intermedio, los tokens JWT se extraen automáticamente de los encabezados de autorización. Después, se deserializan, se validan (mediante los valores de los parámetros `Audience` y `Authority`) y se almacenan como información del usuario a la que se hará referencia más adelante a través de acciones de MVC o filtros de autorización.

El software intermedio de autenticación del portador de 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 objeto `TokenValidationParameters` en el objeto `JwtBearerOptions`.

Recursos adicionales

- **Uso compartido de cookies entre aplicaciones**

<https://docs.microsoft.com/aspnet/core/security/cookie-sharing>

- **Introducción a Identity**

<https://docs.microsoft.com/aspnet/core/security/authentication/identity>

- **Rick Anderson. Autenticación en dos fases con SMS**
<https://docs.microsoft.com/aspnet/core/security/authentication/2fa>
- **Habilitación de la autenticación con Facebook, Google y otros proveedores externos**
<https://docs.microsoft.com/aspnet/core/security/authentication/social/>
- **Michell Anicas. Una introducción a OAuth 2**
<https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>
- **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/>

[ANTERIOR](#)

[SIGUIENTE](#)

Acerca de la autorización en microservicios y aplicaciones web de .NET

23/10/2019 • 8 minutes to read • [Edit Online](#)

Después de la autenticación, las API web de ASP.NET Core deben autorizar el acceso. Este proceso permite que un servicio haga que las API estén disponibles para algunos usuarios autenticados, pero no para todos. La [autorización](#) se puede llevar a cabo según los roles de los usuarios o según una directiva personalizada, que podría incluir la inspección de notificaciones u otro tipo de heurística.

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 de controlador si todas las acciones del controlador requieren autorización), como se muestra en el ejemplo siguiente:

```
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 esa acción. Para restringir aún más una API para que esté disponible solo para usuarios específicos, el atributo se puede expandir para especificar los roles o las directivas necesarios que los usuarios deben cumplir.

Implementar la autorización basada en roles

ASP.NET Core Identity tiene un concepto de roles integrado. Además de los usuarios, ASP.NET Core Identity almacena información sobre los distintos roles que la aplicación usa y realiza un seguimiento de los usuarios que están asignados a cada rol. Estas asignaciones se pueden cambiar mediante programación con el tipo `RoleManager` que actualiza roles en almacenamiento persistente y el tipo `UserManager` que puede conceder o revocar roles de los usuarios.

Si está realizando la autenticación con tokens de portador JWT, el middleware de autenticación de portador JWT de ASP.NET Core llenará los roles de un usuario según las notificaciones de rol que se encuentren en el token. Para limitar el acceso a una acción o un controlador de MVC a usuarios con roles específicos, puede incluir un parámetro `Roles` en la anotación `Authorize` (atributo), tal como se muestra en el fragmento de código siguiente:

```
[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 Administrator o PowerUser pueden tener acceso a las API del controlador ControlPanel (por ejemplo, para ejecutar la acción SetTime). La API ShutDown se restringe aún más para permitir el acceso únicamente a los usuarios con el rol Administrator.

Para pedir a un usuario que tenga varios roles, se usan varios atributos Authorize, como se muestra en este ejemplo:

```
[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

También se pueden escribir reglas de autorización personalizada con [directivas de autorización](#). En esta sección se proporciona información general. Para más información, consulte [ASP.NET Authorization Workshop](#) (Taller de autorización de ASP.NET).

Las directivas de autorización personalizadas se registran en el método `Startup.ConfigureServices` mediante el método `service.AddAuthorization`. Este método toma a un delegado que configura un argumento `AuthorizationOptions`.

```
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 pueden asociarse con distintos tipos de requisitos. Una vez registradas las directivas, se pueden aplicar a una acción o a un controlador pasando el nombre de la directiva como el argumento de la directiva 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 `AddPolicy` es simplemente una manera alternativa de autorizar por rol. Si `[Authorize(Policy="AdministratorsOnly")]` se aplica a una API, solo los usuarios del rol Administrator podrán tener acceso a ella.

La segunda llamada `AddPolicy` muestra una manera sencilla de pedir que una notificación concreta esté presente para el usuario. El método `RequireClaim` también toma opcionalmente valores esperados para la notificación. Si se especifican valores, el requisito se cumple solo si el usuario tiene tanto una notificación del tipo correcto como 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 es en el tercer método `AddPolicy`, ya que usa un requisito de autorización personalizada. Mediante el uso de los requisitos de autorización personalizada, puede tener un gran control sobre cómo se realiza la autorización. Para que funcione, debe implementar estos tipos:

- Un tipo Requirements que deriva de `IAuthorizationRequirement` y que contiene campos en que se especifican los detalles del requisito. En el ejemplo, se trata de un campo de edad para el tipo `MinimumAgeRequirement` de ejemplo.
- Un controlador que implementa `AuthorizationHandler<TRequirement>`, donde T es el tipo de `IAuthorizationRequirement` que el controlador puede satisfacer. El controlador debe implementar el método `HandleRequirementAsync`, que comprueba si un contexto especificado que contiene información sobre el usuario satisface 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 personalizada y un controlador para comprobar la edad de un usuario (según una notificación `DateOfBirth`) está disponible en la [documentación de autorización](#) de ASP.NET Core.

Recursos adicionales

- **ASP.NET Core Authentication (Autenticación en ASP.NET Core)**
<https://docs.microsoft.com/aspnet/core/security/authentication/identity>
- **ASP.NET Core Authorization (Autorización en ASP.NET Core)**
<https://docs.microsoft.com/aspnet/core/security/authorization/introduction>
- **Autorización basada en roles**
<https://docs.microsoft.com/aspnet/core/security/authorization/roles>
- **Custom Policy-Based Authorization (Autorización personalizada basada en directivas)**
<https://docs.microsoft.com/aspnet/core/security/authorization/policies>

Almacenar secretos de aplicación de forma segura durante el desarrollo

23/10/2019 • 5 minutes to read • [Edit Online](#)

Para conectar con los recursos protegidos y otros servicios, las aplicaciones de ASP.NET Core normalmente necesitan usar cadenas de conexión, contraseñas u otras credenciales que contienen información confidencial. Estos fragmentos de información confidenciales se denominan *secretos*. Es un procedimiento recomendado no incluir secretos en el código fuente y, ciertamente, no almacenar secretos en el control de código fuente. En su lugar, debe usar el modelo de configuración de ASP.NET Core para leer los secretos desde ubicaciones más seguras.

Debe separar los secretos usados para acceder a los recursos de desarrollo y almacenamiento provisional de los usados para acceder a los recursos de producción, ya que distintas personas deben tener acceso a los diferentes conjuntos de secretos. Para almacenar secretos usados durante el desarrollo, los enfoques comunes son almacenar secretos en variables de entorno o usar la herramienta ASP.NET Core Secret Manager. Para un almacenamiento más seguro en entornos de producción, los microservicios pueden almacenar secretos en un Azure Key Vault.

Almacenamiento de secretos en variables de entorno

Una manera de mantener secretos fuera del código fuente es que los desarrolladores establezcan secretos basados en cadena como [variables de entorno](#) en sus máquinas de desarrollo. Cuando use variables de entorno para almacenar secretos con nombres jerárquicos, como las anidadas en las secciones de configuración, debe asignar un nombre a las variables para incluir la jerarquía completa de sus secciones, delimitada por signos de dos puntos (:).

Por ejemplo, establecer una variable de entorno `Logging:LogLevel:Default` to `Debug` sería equivalente a un valor de configuración del archivo JSON siguiente:

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Debug"  
    }  
  }  
}
```

Para acceder a estos valores de variables de entorno, la aplicación solo debe llamar a `AddEnvironmentVariables` en su `ConfigurationBuilder` al construir un objeto `IConfigurationRoot`.

Tenga en cuenta que las variables de entorno suelen almacenarse como texto sin formato, por lo que si se pone en peligro la máquina o el proceso con las variables de entorno, se verán los valores de las variables de entorno.

Almacenamiento de secretos mediante ASP.NET Core Secret Manager

La herramienta [Secret Manager](#) de ASP.NET Core proporciona otro método para mantener secretos fuera del código fuente. Para usar la herramienta Secret Manager, instale el paquete

Microsoft.Extensions.Configuration.SecretManager en su archivo del proyecto. Una vez que esa dependencia está presente y se ha restaurado, se puede usar el comando `dotnet user-secrets` para establecer el valor de los secretos desde la línea de comandos. Estos secretos se almacenarán en un archivo JSON en el

directorio del perfil del usuario (los detalles varían según el sistema operativo), lejos del código fuente.

La propiedad `UserSecretsId` del proyecto que está usando los secretos organiza los secretos que establece la herramienta Secret Manager. Por tanto, debe asegurarse de establecer la propiedad `UserSecretsId` en el archivo del proyecto, como se muestra en el siguiente fragmento. El valor predeterminado es un GUID asignado por Visual Studio, pero la cadena real no es importante mientras sea única en su equipo.

```
<PropertyGroup>
  <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

Para usar los secretos almacenados con Secret Manager en una aplicación, debe llamar a `AddUserSecrets<T>` en la instancia de `ConfigurationBuilder` para incluir los secretos de la aplicación en su configuración. El parámetro genérico `T` debe ser un tipo del ensamblado que se aplicó a `UserSecretId`. Normalmente, usar `AddUserSecrets<Startup>` está bien.

`AddUserSecrets<Startup>()` se incluye en las opciones predeterminadas del entorno de desarrollo al usar el método `CreateDefaultBuilder` en *Program.cs*.

[ANTERIOR](#)

[SIGUIENTE](#)

Usar Azure Key Vault para proteger secretos en tiempo de producción

23/10/2019 • 3 minutes to read • [Edit Online](#)

Los secretos almacenados como variables de entorno o almacenados por la herramienta Administrador de secretos todavía se almacenan localmente y se descifran en el equipo. Una opción más segura para almacenar secretos es [Azure Key Vault](#), que proporciona una ubicación central y segura para almacenar claves y secretos.

El paquete **Microsoft.Extensions.Configuration.AzureKeyVault** permite que una aplicación de ASP.NET Core lea información de configuración de Azure Key Vault. Siga estos pasos para empezar a usar secretos de Azure Key Vault:

1. Registre la aplicación como una aplicación de Azure AD. (el acceso a los almacenes de claves se administra mediante Azure AD). Puede hacerlo a través del portal de administración de Azure.\

Como alternativa, si quiere que la aplicación se autentique mediante un certificado en lugar de hacerlo con una contraseña o un secreto de cliente, puede usar el cmdlet de PowerShell [New-AzADApplication](#). El certificado que registre en Azure Key Vault solo necesita su clave pública La aplicación usará la clave privada.

2. Conceda a la aplicación registrada acceso al almacén de claves creando una entidad de servicio. Puede hacerlo usando los siguientes comandos de 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 llamando al método de extensión [AzureKeyVaultConfigurationExtensions.AddAzureKeyVault](#) cuando cree una instancia de [IConfigurationRoot](#). 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.

También puede usar una sobrecarga de `AddAzureKeyVault` que toma un certificado en lugar del secreto de cliente solo incluyendo una referencia al paquete [Microsoft.IdentityModel.Clients.ActiveDirectory](#).

IMPORTANT

Le recomendamos que registre Azure Key Vault como el último proveedor de configuración para que pueda invalidar los valores de configuración de proveedores anteriores.

Recursos adicionales

- **Using Azure Key Vault to protect application secrets (Usar Azure Key Vault para proteger secretos de aplicaciones)**
<https://docs.microsoft.com/azure/guidance/guidance-multitenant-identity-keyvault>
- **Safe storage of app secrets during development (Almacenamiento seguro de secretos de aplicación durante el desarrollo)**
<https://docs.microsoft.com/aspnet/core/security/app-secrets>

- **Configuring data protection (Configuración de la protección de datos)**
<https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/overview>
- **Administración y duración de las claves de protección de datos en ASP.NET Core**
<https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/default-settings>
- Repositorio de GitHub **Microsoft.Extensions.Configuration.KeyPerFile**.
<https://github.com/aspnet/Configuration/tree/master/src/Config.KeyPerFile>

[ANTERIOR](#)

[SIGUIENTE](#)

Puntos clave

23/10/2019 • 8 minutes to read • [Edit Online](#)

A modo de resumen y puntos clave, estas son las conclusiones más importantes de esta guía.

Ventajas del uso de contenedores. Las soluciones basadas en contenedor permiten un gran ahorro, ya que ayudan a reducir los problemas de implementación derivados de las dependencias erróneas en los entornos de producción. Los contenedores mejoran significativamente las operaciones de DevOps y producción.

El uso de los contenedores será generalizado. Los contenedores basados en Docker se están convirtiendo en el estándar de facto del sector, ya que son compatibles con los proveedores más importantes de los ecosistemas de Windows y Linux, como Microsoft, Amazon AWS, Google e IBM. El uso de Docker probablemente se ampliará a centros de datos en la nube y locales.

Los contenedores como una unidad de implementación. Un contenedor de Docker se está convirtiendo en la unidad de implementación estándar para cualquier aplicación o servicio basados en servidor.

Microservicios. La arquitectura de microservicios se está convirtiendo en el método preferido para aplicaciones críticas distribuidas y grandes o complejas basadas en múltiples subsistemas independientes en forma de servicios autónomos. En una arquitectura basada en microservicios, la aplicación se basa en una colección de servicios que se desarrollan, prueban, versionan, implementan y escalan por separado. Cada servicio puede incluir cualquier base de datos autónoma relacionada.

Diseño guiado por el dominio y SOA. Los patrones de arquitectura de microservicios se derivan de la arquitectura orientada a servicios (SOA) y del diseño guiado por el dominio (DDD). Al diseñar y desarrollar microservicios para entornos con necesidades y reglas empresariales en evolución, es importante tener en cuenta los enfoques y los patrones DDD.

Retos de los microservicios. Los microservicios ofrecen muchas capacidades eficaces, como la implementación independiente, los límites de subsistema seguros y la diversidad de tecnología. Sin embargo, también suponen muchos retos nuevos relacionados con el desarrollo de aplicaciones distribuidas, como los modelos de datos fragmentados e independientes, la comunicación resistente entre microservicios, la coherencia final y la complejidad operativa que comporta agregar la 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 algunos escenarios específicos son adecuados para las aplicaciones basadas en microservicio. Estas incluyen aplicaciones grandes y complejas con varios subsistemas en evolución. En estos casos, merece la pena invertir en una arquitectura de software más compleja, ya que la agilidad y el mantenimiento de la aplicación serán mejores a largo plazo.

Contenedores para cualquier aplicación. Los contenedores son prácticos para los microservicios, pero también pueden resultar útiles para las aplicaciones monolíticas basadas en el .NET Framework tradicional, al usar contenedores de Windows. Las ventajas de usar Docker, como solucionar muchos problemas relacionados con el paso de la implementación a la producción y proporcionar entornos de desarrollo y prueba vanguardistas, se aplican a muchos tipos diferentes de aplicaciones.

CLI frente a IDE. Con herramientas de Microsoft, puede desarrollar aplicaciones .NET en contenedores con su método preferido. Puede desarrollar con una CLI y un entorno basado en editor mediante la CLI de Docker y Visual Studio Code. O bien, puede usar un enfoque centrado en 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 en los sistemas distribuidos en general, siempre hay el riesgo de error parcial. Puesto que los clientes y los servicios son procesos independientes

(contenedores), es posible que un servicio no pueda responder de forma oportuna a la solicitud de un cliente. Por ejemplo, podría ser que un servicio estuviera inactivo a causa de un error parcial o por mantenimiento, que estuviera sobrecargado y respondiera lentamente a las solicitudes o bien que simplemente fuera inaccesible durante un breve tiempo debido a problemas de red. Por tanto, una aplicación basada en la nube debe estar preparada para dichos errores y disponer de una estrategia para responder a los mismos. Estas estrategias pueden incluir aplicar directivas de reintento (volver a enviar mensajes o solicitudes) e implementar patrones de interruptor para evitar una carga exponencial de solicitudes repetidas. Básicamente, las aplicaciones basadas en la nube deben tener mecanismos resistentes, ya sean personalizados o basados en la infraestructura de nube, como los de alto nivel de orquestadores o buses de servicio.

Seguridad. Nuestro mundo moderno de contenedores y microservicios puede exponer nuevas vulnerabilidades. Existen varias formas de implementar la seguridad de la aplicación básica, basada en la autenticación y la autorización. Sin embargo, la seguridad del contenedor debe tener en cuenta otros componentes clave que resultan en aplicaciones intrínsecamente más seguras. Un elemento fundamental de crear aplicaciones más seguras es tener una forma segura de comunicarse con otros sistemas y aplicaciones, algo que a menudo requiere credenciales, tokens, contraseñas y demás, que normalmente se denominan secretos de la 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 los consuma. Esos secretos deben almacenarse y guardarse de forma segura, como al usar Azure Key Vault.

Orquestadores. Los orquestadores basados en contenedores, como Azure Kubernetes Service y Azure Service Fabric, representan una parte fundamental de todo microservicio significativo y aplicación basada en contenedores. Estas aplicaciones llevan consigo gran complejidad, necesidades de escalabilidad y evolucionan constantemente. En esta guía se han presentado orquestadores y su rol en las soluciones basadas en microservicio y contenedor. Si sus necesidades de aplicación lo están dirigiendo hacia aplicaciones en contenedores complejas, le resultará útil para buscar recursos adicionales que le permitan obtener más información sobre los orquestadores.

[ANTERIOR](#)