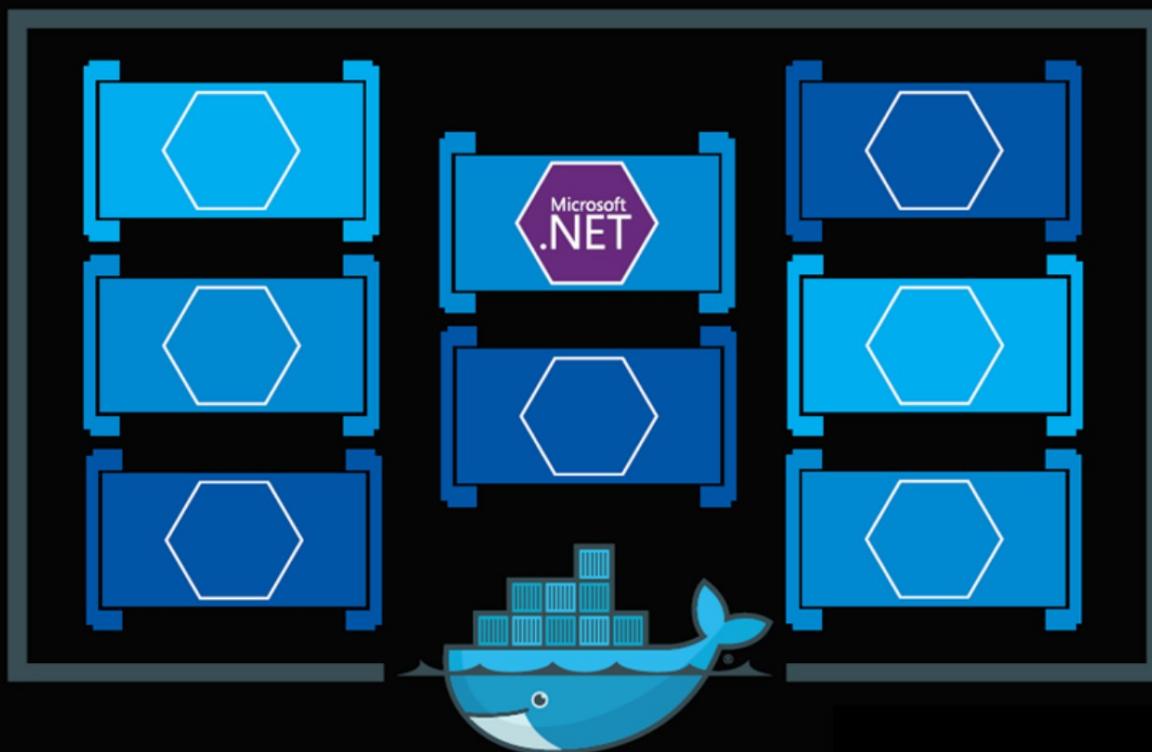


# .NET Microservices: Architecture for Containerized .NET Applications



**Cesar de la Torre  
Bill Wagner  
Mike Rousos**

Microsoft Corporation

EDICIÓN v6.0 - Actualizado a ASP.NET Core 6.0

Consulte [el registro de cambios](#) para ver las actualizaciones del libro y las contribuciones de la comunidad.

Esta guía es una introducción al desarrollo de aplicaciones basadas en microservicios y su gestión mediante contenedores. Analiza el diseño arquitectónico y los enfoques de implementación utilizando contenedores .NET y Docker.

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

---

## Enlaces de acción

- Este libro electrónico también está disponible en formato PDF (solo versión en inglés) [Descargar](#)
- Clone/bifurque la aplicación de referencia [eShopOnContainers en GitHub](#)
- Mira el [video introductorio](#)
- Conozca la [Arquitectura de Microservicios de inmediato](#)

## Introducción

Las empresas obtienen cada vez más ahorros de costos, resuelven problemas de implementación y mejoran las operaciones de producción y DevOps mediante el uso de contenedores. Microsoft ha lanzado innovaciones de contenedores para Windows y Linux al crear productos como Azure Kubernetes Service y Azure Service Fabric, y al asociarse con líderes de la industria como Docker, Mesosphere y Kubernetes.

Estos productos ofrecen soluciones de contenedores que ayudan a las empresas a crear e implementar aplicaciones a la velocidad y la escala de la nube, independientemente de la plataforma o las herramientas que elijan.

Docker se está convirtiendo en el estándar de facto en la industria de contenedores, respaldado por los proveedores más importantes en los ecosistemas de Windows y Linux. (Microsoft es uno de los principales proveedores de la nube que admite Docker). En el futuro, Docker probablemente estará en todas partes en cualquier centro de datos en la nube o en las instalaciones.

Además, la [arquitectura de microservicios](#) está emergiendo como un enfoque importante para las aplicaciones distribuidas de misión crítica. En una arquitectura basada en microservicios, la aplicación se basa en una colección de servicios que se pueden desarrollar, probar, implementar y versionar de forma independiente.

## Acerca de esta guía

Esta guía es una introducción al desarrollo de aplicaciones basadas en microservicios y su gestión mediante contenedores. Analiza el diseño arquitectónico y los enfoques de implementación utilizando contenedores .NET y Docker. Para que sea más fácil comenzar con los contenedores y los microservicios, la guía se centra en una aplicación de referencia basada en microservicios y en contenedores que puede explorar. La aplicación de muestra está disponible en el [repositorio de GitHub de eShopOnContainers](#).

---

Esta guía proporciona una guía básica de desarrollo y arquitectura principalmente a nivel de entorno de desarrollo con un enfoque en dos tecnologías: Docker y .NET. Nuestra intención es que lea esta guía cuando piense en el diseño de su aplicación sin centrarse en la infraestructura (en la nube o en las instalaciones) de su entorno de producción. Tomará decisiones sobre su infraestructura más adelante, cuando cree sus aplicaciones listas para producción. Por lo tanto, esta guía pretende ser independiente de la infraestructura y más centrada en el entorno de desarrollo.

Una vez que haya estudiado esta guía, su próximo paso será obtener información sobre los microservicios listos para la producción en Microsoft Azure.

## Versión

Esta guía se revisó para cubrir la versión .NET 6 junto con muchas actualizaciones adicionales relacionadas con la misma "ola" de tecnologías (es decir, Azure y tecnologías adicionales de terceros) que coinciden con el lanzamiento de .NET 6. Es por eso que la versión del libro también se ha actualizado a la versión 6.0.

## Lo que esta guía no cubre

Esta guía no se enfoca en el ciclo de vida de la aplicación, DevOps, canalizaciones de CI/CD o trabajo en equipo. La guía complementaria [Containerized Docker Application Lifecycle with Microsoft Platform and Tools](#) se centra en ese tema. La guía actual tampoco proporciona detalles de implementación en la infraestructura de Azure, como información sobre orquestadores específicos.

## Recursos adicionales

- Ciclo de vida de la aplicación Docker en contenedores con la plataforma y las herramientas de Microsoft (libro electrónico descargable) <https://aka.ms/dockerlifecyclebook>

---

## Quién debería usar esta guía

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

Esta guía también le resultará útil si es un tomador de decisiones técnicas, como un arquitecto empresarial, que desea una descripción general de la arquitectura y la tecnología antes de decidir qué enfoque seleccionar para las aplicaciones distribuidas nuevas y modernas.

## Como usar esta guia

La primera parte de esta guía presenta los contenedores Docker, analiza cómo elegir entre .NET 6 y .NET Framework como marco de desarrollo y proporciona una descripción general de los microservicios. Este contenido es para arquitectos y tomadores de decisiones técnicas que desean una visión general pero no necesitan concentrarse sobre los detalles de implementación del código.

La segunda parte de la guía comienza con la sección [Proceso de desarrollo para aplicaciones basadas en Docker](#). Se enfoca en el desarrollo y patrones de microservicios para implementar aplicaciones usando .NET y Docker. Esta sección será de mayor interés para los desarrolladores y arquitectos que deseen centrarse en el código y en los patrones y los detalles de implementación.

## Microservicio relacionado y referencia basada en contenedores aplicación: eShopOnContainers

La aplicación eShopOnContainers es una aplicación de referencia de código abierto para .NET y microservicios que está diseñada para implementarse mediante contenedores Docker. La aplicación consta de varios subsistemas, incluidos varios front-end de la interfaz de usuario de la tienda electrónica (una aplicación Web MVC, un Web SPA y una aplicación móvil nativa). También incluye los microservicios y contenedores de back-end para todas las operaciones del lado del servidor requeridas.

El propósito de la aplicación es mostrar patrones arquitectónicos. NO ES 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 utiliza para probar nuevas tecnologías potencialmente interesantes a medida que aparecen.

## ¡Envíenos sus comentarios!

Escribimos esta guía para ayudarlo a comprender la arquitectura de las aplicaciones en contenedores y los microservicios en .NET. La guía y la aplicación de referencia relacionada evolucionarán, por lo que agradecemos sus comentarios. Si tiene comentarios sobre cómo se puede mejorar esta guía, envíe sus comentarios a <https://aka.ms/ebookfeedback>.

---

## Créditos

### Coautores:

Cesar de la Torre, Sr. PM, equipo de producto .NET, Microsoft Corp.

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

Mike Rousos, ingeniero de software principal, equipo DevDiv CAT, Microsoft

### Editores:

mike papa

steve hoag

### Participantes y revisores:

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

Jimmy Bogard, arquitecto jefe de Headspring

Udi Dahan, fundador y director ejecutivo de Particular Software

Jimmy Nilsson, cofundador y director ejecutivo de Factor10



Glenn Condron, gerente senior de programas, equipo de ASP.NET

Mark Fussell, jefe principal de PM, equipo de Azure Service Fabric, Microsoft

Diego Vega, líder de PM, equipo de Entity Framework, Microsoft

Barry Dorrans, gerente senior del programa de seguridad

Rowan Miller, director senior de programas, Microsoft

Ankit Asthana, gerente principal de PM, equipo de .NET, Microsoft

Scott Hunter, socio director PM, equipo .NET, Microsoft

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

Dylan Reisenberger, arquitecto y líder de desarrollo en Polly

Steve "ardalis" Smith - Arquitecto de software y formador - [Ardalis.com](https://ardalis.com)

---

Ian Cooper, arquitecto de codificación en Brighter

Unai Zorrilla, Arquitecto y Dev Lead en Plain Concepts

Eduard Tomas, líder de desarrollo en Plain Concepts

Ramón Tomas, Desarrollador en Plain Concepts

David Sanz, Desarrollador en Plain Concepts

Javier Valero, Director de Operaciones de Grupo Solutio

Pierre Millet, consultor senior, Microsoft

Michael Friis, Gerente de Producto, Docker Inc.

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

Miguel Veloso, Ingeniero de Desarrollo de Software en Plain Concepts

Sumit Ghosh, consultor principal de Neudesic

## Derechos de autor

### PUBLICADO POR

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

Una división de Microsoft Corporation

Una forma de Microsoft

Redmond, Washington 98052-6399

Copyright © 2022 por Microsoft Corporation

Todos los derechos reservados. Ninguna parte del contenido de este libro puede ser reproducida o transmitida de ninguna forma o por ningún medio sin el permiso por escrito del editor.

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

Algunos ejemplos representados en este documento se proporcionan solo como ilustración y son ficticios. No se pretende ni debe inferirse ninguna asociación o conexión real.

Microsoft y las marcas comerciales enumeradas en <https://www.microsoft.com> en la página web "Marcas comerciales" son marcas comerciales del grupo de empresas de Microsoft.

Mac y macOS son marcas comerciales de Apple Inc.

El logotipo de la ballena Docker es una marca comercial registrada de Docker, Inc. Se utiliza con autorización.

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

# Contenido

Introducción a Contenedores y Docker.....	1
¿Qué es Docker? .....	2
Comparación de contenedores Docker con máquinas virtuales.....	3
Una simple analogía .....	4
Terminología acopiable .....	5
Contenedores Docker, imágenes y registros .....	7
Elección entre .NET 6 y .NET Framework para contenedores Docker.....	9
Orientaciones generales .....	9
Cuándo elegir .NET para contenedores Docker.....	10
Desarrollo e implementación multiplataforma .....	10
Uso de contenedores para proyectos nuevos ("green-field") .....	11
Crear e implementar microservicios en contenedores.....	11
Implementación de alta densidad en sistemas escalables.....	11
Cuándo elegir .NET Framework para contenedores Docker.....	12
Migración de aplicaciones existentes directamente a un contenedor de Windows Server .....	12
Uso de bibliotecas .NET de terceros o paquetes NuGet no disponibles para .NET 6 .....	12
Uso de tecnologías .NET no disponibles para .NET 6.....	12
Uso de una plataforma o API que no es compatible con .NET 6 .....	13
Migración de la aplicación ASP.NET existente a .NET 6.....	13
Tabla de decisiones: marcos .NET para usar con Docker.....	13
A qué sistema operativo apuntar con contenedores .NET.....	14
Imágenes oficiales de .NET Docker .....	dieciséis
Optimizaciones de imágenes de .NET y Docker para desarrollo frente a producción .....	dieciséis
Arquitectura de aplicaciones basadas en contenedores y microservicios .....	18
Principios de diseño de contenedores .....	18
Contenedorización de aplicaciones monolíticas .....	19
Implementación de una aplicación monolítica como contenedor.....	21
Publicación de una aplicación basada en un único contenedor en Azure App Service.....	21

Administrar el estado y los datos en las aplicaciones de Docker .....	22
Arquitectura orientada a Servicios.....	25
Arquitectura de microservicios.....	25
Recursos adicionales .....	27
Soberanía de datos por microservicio .....	27
La relación entre los microservicios y el patrón Bounded Context.....	29
Arquitectura lógica versus arquitectura física.....	30
Desafíos y soluciones para la gestión de datos distribuidos.....	31
Desafío n.º 1: cómo definir los límites de cada microservicio.....	31
Desafío n.º 2: cómo crear consultas que recuperen datos de varios microservicios.....	32
Desafío n.º 3: Cómo lograr la coherencia entre múltiples microservicios .....	33
Desafío n.º 4: cómo diseñar la comunicación a través de los límites de los microservicios .....	35
Recursos adicionales .....	36
Identificar los límites del modelo de dominio para cada microservicio.....	36
El patrón de la puerta de enlace de la API frente a la comunicación directa entre el cliente y el microservicio.....	40
Comunicación directa entre el cliente y el microservicio .....	40
¿Por qué considerar puertas de enlace API en lugar de comunicación directa entre el cliente y el microservicio? 41	41
¿Qué es el patrón API Gateway?.....	42
Características principales en el patrón API Gateway .....	44
Uso de productos con características de API Gateway.....	45
Inconvenientes del patrón API Gateway.....	47
Recursos adicionales .....	48
Comunicación en una arquitectura de microservicios .....	48
Tipos de comunicación .....	49
La integración asíncrona de microservicios refuerza la autonomía de los microservicios .....	50
Estilos de comunicación .....	52
Comunicación asíncrona basada en mensajes .....	54
Comunicación basada en mensajes de un solo receptor .....	55
Comunicación basada en mensajes de múltiples receptores .....	56
Comunicación asíncrona basada en eventos .....	56
Una nota sobre las tecnologías de mensajería para sistemas de producción .....	57
Publicación resistente en el bus de eventos .....	58

Recursos adicionales .....	58
Creación, evolución y control de versiones de contratos y API de microservicios .....	59
Recursos adicionales .....	59
Direccionabilidad de microservicios y el registro de servicios .....	60
Recursos adicionales .....	60
Creación de una interfaz de usuario compuesta basada en microservicios .....	60
Recursos adicionales .....	62
Resiliencia y alta disponibilidad en microservicios .....	63
Gestión de salud y diagnóstico en microservicios .....	63
Recursos adicionales .....	sesenta y cinco
Organice microservicios y aplicaciones multicontenedor para una alta escalabilidad y disponibilidad .....	66
Plataformas de software para agrupación en clústeres, orquestación y programación de contenedores.....	68
Uso de orquestadores basados en contenedores en Microsoft Azure.....	68
Uso del servicio Azure Kubernetes .....	68
Entorno de desarrollo para Kubernetes.....	69
Introducción a Azure Kubernetes Service (AKS) .....	70
Implementación con gráficos de Helm en clústeres de Kubernetes.....	70
Recursos adicionales .....	71
Proceso de desarrollo de aplicaciones basadas en Docker.....	72
Entorno de desarrollo para aplicaciones Docker .....	72
Opciones de herramientas de desarrollo: IDE o editor .....	72
Recursos adicionales .....	73
Lenguajes y marcos .NET para contenedores Docker .....	73
Flujo de trabajo de desarrollo para aplicaciones Docker.....	73
Flujo de trabajo para desarrollar aplicaciones basadas en contenedores Docker .....	73
Paso 1. Comience a programar y cree su línea de base de servicio o aplicación inicial .....	75
Paso 2. Cree un Dockerfile relacionado con una imagen base de .NET existente.....	76
Paso 3. Cree sus imágenes Docker personalizadas e incorpore su aplicación o servicio en ellas .....	83
Paso 4. Defina sus servicios en docker-compose.yml cuando construya una aplicación Docker de múltiples contenedores .....	84
Paso 5. Cree y ejecute su aplicación Docker .....	86
Paso 6. Pruebe su aplicación Docker usando su host Docker local.....	89

Flujo de trabajo simplificado al desarrollar contenedores con Visual Studio .....	90
Uso de comandos de PowerShell en un Dockerfile para configurar contenedores de Windows.....	91
<b>Diseño y desarrollo de aplicaciones .NET basadas en microservicios y de contenedores múltiples .....</b>	<b>93</b>
Diseñar una aplicación orientada a microservicios .....	93
Especificaciones de la aplicación .....	93
Contexto del equipo de desarrollo .....	94
La elección de una arquitectura .....	94
Beneficios de una solución basada en microservicios .....	97
Desventajas de una solución basada en microservicios .....	98
Arquitectura externa versus interna y patrones de diseño .....	99
El nuevo mundo: múltiples patrones arquitectónicos y microservicios políglotas.....	100
Creación de un microservicio CRUD basado en datos simple .....	102
Diseño de un microservicio CRUD simple .....	102
Implementación de un microservicio CRUD simple con ASP.NET Core.....	103
La cadena de conexión de la base de datos y las variables de entorno utilizadas por los contenedores de Docker .....	109
Generación de metadatos de descripción de Swagger desde su ASP.NET Core Web API .....	111
Definición de su aplicación multicontenedor con docker-compose.yml .....	116
Usar un servidor de base de datos ejecutándose como un contenedor .....	127
SQL Server ejecutándose como un contenedor con una base de datos relacionada con microservicios.....	128
Siembra con datos de prueba en el inicio de la aplicación web.....	129
Base de datos EF Core InMemory frente a SQL Server ejecutándose como un contenedor.....	132
Uso de un servicio de caché de Redis que se ejecuta en un contenedor.....	132
Implementación de comunicación basada en eventos entre microservicios (eventos de integración).....	133
Uso de intermediarios de mensajes y buses de servicios para sistemas de producción .....	134
Eventos de integración.....	135
El autobús de eventos .....	135
Recursos adicionales .....	138
Implementación de un bus de eventos con RabbitMQ para el entorno de desarrollo o prueba.....	138
Implementando un método de publicación simple con RabbitMQ.....	139
Implementación del código de suscripción con la API de RabbitMQ.....	140
Recursos adicionales .....	141

Suscripción a eventos.....	141
Publicación de eventos a través del bus de eventos.....	142
Idempotencia en eventos de mensajes de actualización.....	149
Eliminación de duplicados de mensajes de eventos de integración .....	150
Probar servicios ASP.NET Core y aplicaciones web .....	152
Pruebas en eShopOnContainers.....	155
Implementar tareas en segundo plano en microservicios con IHostedService y la clase BackgroundService .....	157
Registro de servicios alojados en suWebHost o Host .....	158
La interfaz IHostedService .....	159
Implementación de IHostedService con una clase de servicio alojada personalizada que se deriva de la clase base de BackgroundService.....	160
Recursos adicionales .....	163
Implementar API Gateways con Ocelot.....	163
Construya y diseñe sus API Gateways.....	163
Implementación de sus API Gateways con Ocelot .....	168
Uso de Kubernetes Ingress más Ocelot API Gateways .....	180
Funciones transversales adicionales en una puerta de enlace API de Ocelot .....	181
Abordar la complejidad empresarial en un microservicio con patrones DDD y CQRS .....	182
Aplicar patrones CQRS y DDD simplificados en un microservicio.....	184
Recursos adicionales .....	186
Aplicar enfoques CQRS y CQS en un microservicio DDD en eShopOnContainers .....	186
Los patrones CQRS y DDD no son arquitecturas de alto nivel.....	187
Implementar lecturas/consultas en un microservicio CQRS .....	188
Use ViewModels creados específicamente para aplicaciones cliente, independientemente de las restricciones del modelo de dominio .....	189
Usar Dapper como un micro ORM para realizar consultas.....	189
ViewModels dinámicos versus estáticos .....	190
Recursos adicionales .....	193
Diseñar un microservicio orientado a DDD .....	194
Mantenga los límites del contexto del microservicio relativamente pequeños .....	194
Capas en microservicios DDD .....	195

Diseñar un modelo de dominio de microservicios .....	198
El patrón de Entidad de Dominio .....	199
Implementar un modelo de dominio de microservicios con .NET.....	204
Estructura del modelo de dominio en una biblioteca estándar de .NET personalizada .....	204
Estructurar agregados en una biblioteca .NET Standard personalizada .....	205
Implementar entidades de dominio como clases POCO .....	206
Encapsular datos en las Entidades de Dominio .....	207
Seedwork (clases base reutilizables e interfaces para su modelo de dominio) .....	210
La clase base de Entidad personalizada .....	211
Contratos de repositorio (interfaces) en la capa del modelo de dominio .....	212
Recursos adicionales .....	213
Implementar objetos de valor.....	213
Características importantes de los objetos de valor .....	214
Implementación de objetos de valor en C#.....	215
Cómo conservar objetos de valor en la base de datos con EF Core 2.0 y versiones posteriores.....	217
Conservar objetos de valor como tipos de entidad de propiedad en EF Core 2.0 y versiones posteriores.....	218
Recursos adicionales .....	221
Usar clases de enumeración en lugar de tipos de enumeración .....	222
Implementar una clase base de Enumeración.....	222
Recursos adicionales .....	223
Validaciones de diseño en la capa del modelo de dominio .....	223
Implementar validaciones en la capa del modelo de dominio.....	224
Recursos adicionales .....	226
Validación del lado del cliente (validación en las capas de presentación).....	226
Recursos adicionales .....	227
Eventos de dominio: diseño e implementación.....	228
¿Qué es un evento de dominio? .....	228
Eventos de dominio versus eventos de integración .....	229
Los eventos de dominio como una forma preferida de desencadenar efectos secundarios en múltiples agregados dentro del mismo dominio .....	229
Implementar eventos de dominio.....	231
Conclusiones sobre los eventos de dominio .....	238

Recursos adicionales .....	.239
Diseñar la capa de persistencia de la infraestructura.....	.239
El patrón del repositorio .....	240
Recursos adicionales .....	.243
Implementar la capa de persistencia de la infraestructura con Entity Framework Core .....	.244
Introducción a Entity Framework Core.....	244
Infraestructura en Entity Framework Core desde una perspectiva de DDD.....	245
Implementar repositorios personalizados con Entity Framework Core.....	.246
Duración de la instancia de EF DbContext y IUnitOfWork en su contenedor de IoC.....	.249
El tiempo de vida de la instancia del repositorio en su contenedor IoC.....	.249
Asignación de tablas .....	.250
Implementar el patrón de Especificación de consultas.....	253
Utilizar bases de datos NoSQL como infraestructura de persistencia.....	.255
Introducción a Azure Cosmos DB y la API nativa de Cosmos DB .....	256
Implementar código .NET dirigido a MongoDB y Azure Cosmos DB .....	.258
Diseñar la capa de aplicación de microservicios y la API web .....	.266
Utilice los principios SOLID y la Inyección de Dependencia.....	266
Implementar la capa de aplicación de microservicio utilizando la API web .....	.267
Use Inyección de dependencia para injectar objetos de infraestructura en su capa de aplicación.....	267
Implementar los patrones de comando y controlador de comandos .....	.271
La canalización del proceso de comandos: cómo activar un controlador de comandos.....	.278
Implementar la canalización del proceso de comando con un patrón de mediador (MediatR) .....	281
Aplicar preocupaciones transversales al procesar comandos con los Comportamientos en MediatR .....	.287
<b>Implementar aplicaciones resilientes .....</b>	<b>.291</b>
Manejar fallas parciales .....	.292
Estrategias para manejar fallas parciales .....	.294
Recursos adicionales .....	.295
Implementar reintentos con retroceso exponencial .....	.295
Implementar conexiones resilientes de Entity Framework Core SQL.....	.295
Estrategias de ejecución y transacciones explícitas usando BeginTransaction y múltiples DbContexts	296
Recursos adicionales .....	.298
Use IHttpClientFactory para implementar solicitudes HTTP resilientes .....	.298

Problemas con la clase HttpClient original disponible en .NET .....	298
Beneficios de usar IHttpClientFactory.....	299
Múltiples formas de usar IHttpClientFactory.....	300
Cómo utilizar clientes tipificados con IHttpClientFactory.....	300
Recursos adicionales .....	303
Implementar reintentos de llamadas HTTP con retroceso exponencial con políticas IHttpClientFactory y Polly ...	304
Agregue una estrategia de fluctuación a la política de reintento .....	305
Recursos adicionales .....	305
Implementar el patrón del disyuntor .....	305
Implementar el patrón Circuit Breaker con IHttpClientFactory y Polly .....	306
Pruebe los reintentos de Http y los disyuntores en eShopOnContainers.....	307
Recursos adicionales .....	309
Vigilancia de la salud .....	309
Implementar comprobaciones de estado en los servicios de ASP.NET Core .....	310
Usar perros guardianes.....	314
Comprobaciones de estado al usar orquestadores .....	316
Monitoreo avanzado: visualización, análisis y alertas .....	316
Recursos adicionales .....	317
Cree aplicaciones web y microservicios .NET seguros.....	318
Implementar autenticación en microservicios .NET y aplicaciones web .....	318
Autenticar con ASP.NET Core Identity.....	319
Autenticarse con proveedores externos .....	321
Autenticar con tokens de portador .....	322
Autenticarse con un proveedor de identidad de OpenID Connect o OAuth 2.0 .....	323
Emitir tokens de seguridad desde un servicio ASP.NET Core .....	324
Consumir tokens de seguridad.....	325
Recursos adicionales.....	326
Acerca de la autorización en microservicios .NET y aplicaciones web .....	327
Implementar la autorización basada en roles .....	327
Implementar la autorización basada en políticas .....	328
Recursos adicionales .....	329
Almacenar los secretos de la aplicación de forma segura durante el desarrollo.....	329

Almacenar secretos en variables de entorno .....	330
Almacenar secretos con ASP.NET Core Secret Manager .....	330
Usar Azure Key Vault para proteger los secretos en el momento de la producción .....	331
Recursos adicionales .....	332
Conclusiones clave de la arquitectura de microservicios .NET .....	333

# Introducción a Contenedores y Docker

La contenedorización es un enfoque para el desarrollo de software en el que una aplicación o servicio, sus dependencias y su configuración (abstraídos como archivos de manifiesto de implementación) se empaquetan juntos 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) del host.

Así como los contenedores de envío permiten el transporte de mercancías por barco, tren o camión, independientemente de la carga que contengan, los contenedores de software actúan como una unidad estándar de implementación de software que puede contener diferentes códigos y dependencias. La creación de contenedores de software de esta manera permite a los desarrolladores y profesionales de TI implementarlos en entornos con poca o ninguna modificación.

Los contenedores también aislan las aplicaciones entre sí en un sistema operativo compartido. Las aplicaciones en contenedores se ejecutan sobre un host contenedor que, a su vez, se ejecuta en el sistema operativo (Linux o Windows). Por lo tanto, los contenedores ocupan un espacio significativamente menor que las imágenes de máquinas virtuales (VM).

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

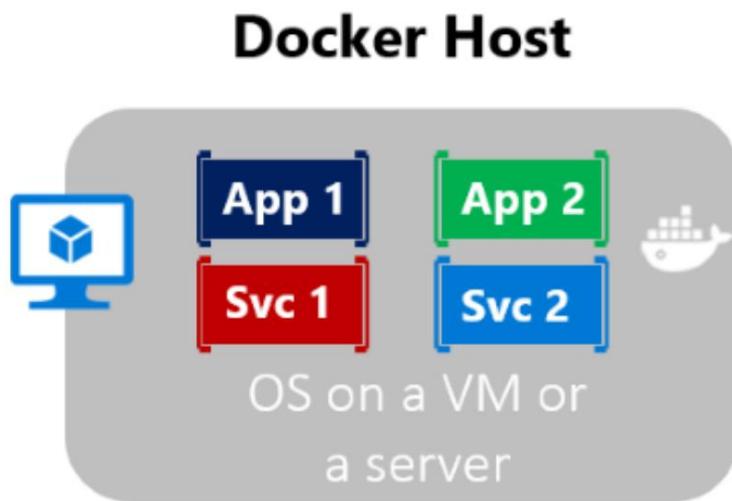


Figura 2-1. Múltiples contenedores ejecutándose en un host de contenedor

Otro beneficio de la contenedorización es la escalabilidad. Puede escalar horizontalmente rápidamente creando nuevos contenedores para tareas a corto plazo. Desde el punto de vista de una aplicación, instanciar una imagen (crear un contenedor) es similar a instanciar un proceso como un servicio o una aplicación web. Sin embargo, para mayor confiabilidad, cuando ejecuta varias instancias de la misma imagen en varios servidores de host, normalmente desea que cada contenedor (instancia de imagen) se ejecute en un servidor de host diferente o VM en diferentes dominios de error.

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

## ¿Qué es Docker?

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

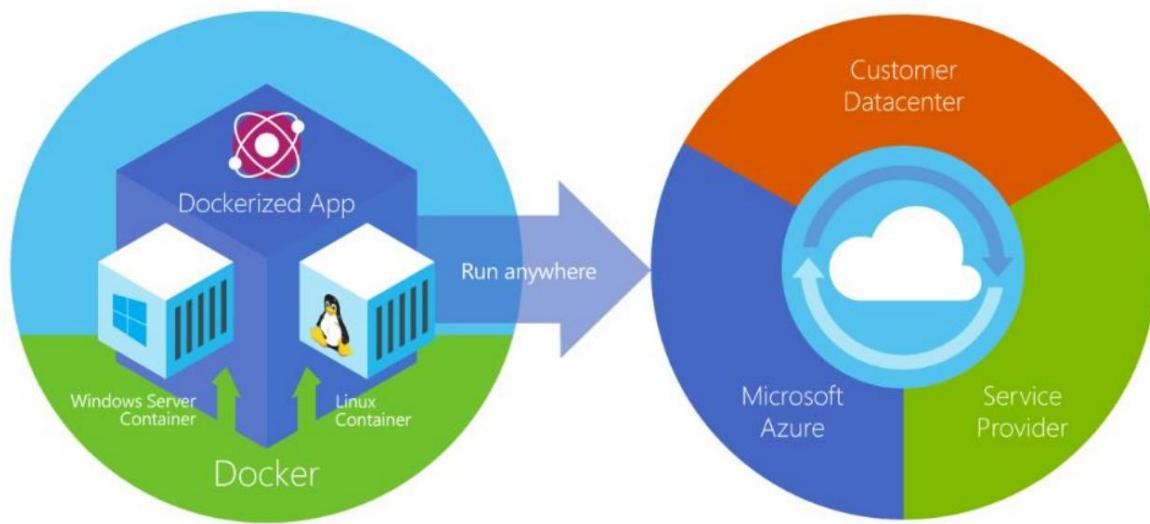


Figura 2-2. Docker implementa contenedores en todas las capas de la nube híbrida.

Los contenedores de Docker pueden ejecutarse en cualquier lugar, en las instalaciones del centro de datos del cliente, en un proveedor de servicios externo o en la nube, en Azure. Los contenedores de imágenes de Docker pueden ejecutarse 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 (hasta ahora, usando una VM Linux de Hyper-V), donde host significa un servidor o una VM.

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

directamente en macOS). Los desarrolladores que trabajan en Windows pueden crear imágenes para contenedores de Linux o Windows.

Para alojar contenedores en entornos de desarrollo y proporcionar herramientas de desarrollo adicionales, Docker incluye Docker Desktop para [Windows](#) o [macOS](#). Estos productos instalan la VM necesaria (el host de Docker) para alojar los contenedores.

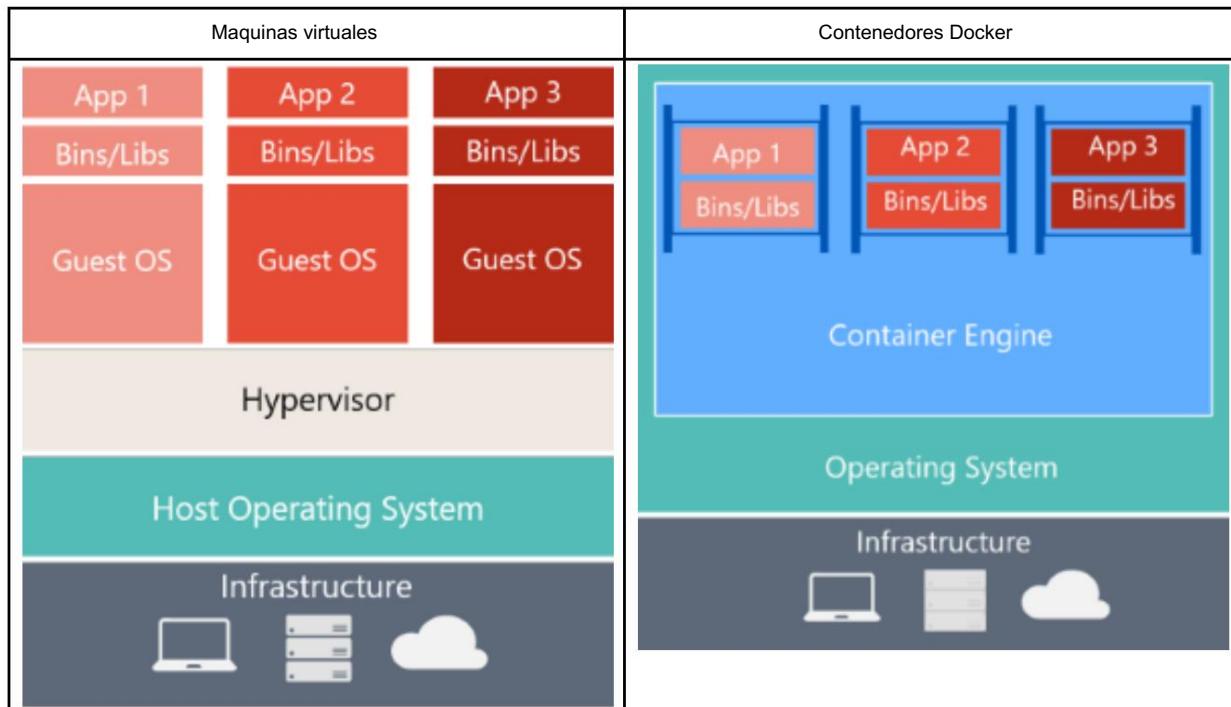
Para ejecutar [contenedores de Windows](#), hay dos tipos de tiempos de ejecución:

- Los contenedores de Windows Server brindan aislamiento de aplicaciones a través del proceso y el espacio de nombres tecnología de aislamiento. Un contenedor de Windows Server comparte un núcleo con el host del contenedor y con todos los contenedores que se ejecutan en el host.
- Los contenedores de Hyper-V amplían el aislamiento proporcionado por los contenedores de Windows Server al ejecutar cada contenedor en una máquina virtual altamente optimizada. En esta configuración, el kernel del host del contenedor no se comparte con los contenedores de Hyper-V, lo que proporciona un mejor aislamiento.

Las imágenes para estos contenedores se crean de la misma manera y funcionan de la misma manera. La diferencia está en cómo se crea el contenedor a partir de la imagen que ejecuta un contenedor Hyper-V que requiere un parámetro adicional. Para obtener más información, consulte [Contenedores de Hyper-V](#).

## Comparación de contenedores Docker con máquinas virtuales

La figura 2-3 muestra una comparación entre máquinas virtuales y contenedores Docker.



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

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

Para las máquinas virtuales, hay tres capas base en el servidor host, de abajo hacia arriba: infraestructura, sistema operativo host y un hipervisor y, además, cada máquina virtual tiene su propio sistema operativo y todas las bibliotecas necesarias. Para Docker, el servidor host solo tiene la infraestructura y el sistema operativo y, además, el motor del contenedor, que mantiene el contenedor aislado pero comparte los servicios básicos del sistema operativo.

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

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

El objetivo principal de una imagen es hacer que el entorno (dependencias) sea el mismo en diferentes implementaciones. Esto significa que puede depurarlo en su máquina y luego implementarlo en otra máquina con el mismo entorno garantizado.

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

Al usar Docker, no escuchará a los desarrolladores decir: "Funciona en mi máquina, ¿por qué no en producción?"

Simplemente pueden decir: "Se ejecuta en Docker", porque la aplicación Docker empaquetada se puede ejecutar en cualquier entorno Docker compatible y se ejecuta de la manera prevista en todos los objetivos de implementación (como desarrollo, control de calidad, preparación y producción). ).

## Una analogía sencilla

Quizás una simple analogía pueda ayudar a comprender el concepto central de Docker.

Retrocedamos en el tiempo a la década de 1950 por un momento. No había procesadores de texto y las fotocopiadoras se usaban en todas partes (más o menos).

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

En algún momento, se da cuenta de que las cartas son solo una composición de un gran conjunto de párrafos, que se seleccionan y organizan según sea necesario, de acuerdo con el propósito de la carta, por lo que diseña un sistema para emitir cartas rápidamente, esperando obtener una gran cantidad. elevar.

El sistema es sencillo:

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

2. Para emitir un conjunto de cartas, selecciona las hojas con los párrafos que necesita, luego las apila y las alinea para que se vean y se lean bien.
3. Finalmente, coloca el conjunto en la fotocopiadora y presiona comenzar para producir tantas letras como sea necesario.

Entonces, simplificando, esa es la idea central de Docker.

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

Entonces, cuando "mira" el sistema de archivos después de que se haya copiado la capa, verá todos los archivos incluidos en la capa cuando se instaló el programa.

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

De manera similar, puede pensar en un contenedor como la "computadora" con el disco duro de imagen instalado. El contenedor, al igual que una computadora, puede encenderse o apagarse.

## Terminología acoplable

Esta sección enumera términos y definiciones con los que debe estar familiarizado antes de profundizar en Docker.

Para obtener más definiciones, consulte el extenso [glosario](#) proporcionado por Docker.

Imagen de contenedor: un paquete con todas las dependencias e información necesaria para crear un contenedor.

Una imagen incluye todas las dependencias (como los marcos) más la configuración de implementación y ejecución que utilizará un tiempo de ejecución de contenedor. Por lo general, una imagen se deriva de varias imágenes base que son capas apiladas una encima de la otra para formar el sistema de archivos del contenedor. Una imagen es inmutable una vez que ha sido creada.

Dockerfile: un archivo de texto que contiene instrucciones para crear una imagen de Docker. Es como un script por lotes, la primera línea establece la imagen base para comenzar y luego sigue las instrucciones para instalar los programas necesarios, copiar archivos, etc., hasta que obtengas el entorno de trabajo que necesitas.

Construir: La acción de construir una imagen de contenedor basada en la información y el contexto proporcionado por su Dockerfile, más archivos adicionales en la carpeta donde se construye la imagen. Puede crear imágenes con el siguiente comando de Docker:

compilación de la ventana acoplable

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

Volúmenes: ofrece un sistema de archivos grabable que el contenedor puede usar. Dado que las imágenes son de solo lectura, pero la mayoría de los programas necesitan escribir en el sistema de archivos, los volúmenes agregan una capa de escritura, encima de la imagen del contenedor, para que los programas tengan acceso a un sistema de archivos de escritura. El programa no sabe que está accediendo a un

sistema de archivos en capas, es solo el sistema de archivos como de costumbre. Los volúmenes viven en el sistema host y son administrados por Docker.

Etiqueta: una marca o etiqueta que 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).

Construcción en varias etapas: es una característica, desde Docker 17.05 o superior, que ayuda a reducir el tamaño de las imágenes finales. Por ejemplo, se puede usar una imagen base grande que contenga el SDK para compilar y publicar, y luego se puede usar una imagen base pequeña solo en tiempo de ejecución para hospedar la aplicación.

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

Registro: Un servicio que proporciona acceso a los repositorios. El registro predeterminado para la mayoría de las imágenes públicas es [Docker Hub](#) (propiedad de Docker como organización). Un registro generalmente contiene repositorios de varios equipos. Las empresas suelen tener registros privados para almacenar y administrar las imágenes que han creado.

Azure Container Registry es otro ejemplo.

Imagen de múltiples arquitecturas: para arquitecturas múltiples, es una característica que simplifica la selección de la imagen adecuada, según la plataforma en la que se ejecuta Docker. Por ejemplo, cuando un Dockerfile solicita una imagen base DESDE mcr.microsoft.com/dotnet/sdk:6.0 del registro, en realidad obtiene 6.0-nanoserver-20H2, 6.0-nanoserver-1809 o 6.0-bullseye-slim, según el sistema operativo y versión donde se ejecuta Docker.

Docker Hub: Un registro público para subir imágenes y trabajar con ellas. Docker Hub proporciona alojamiento de imágenes de Docker, registros públicos o privados, activadores de compilación y enlaces web, e integración con GitHub y Bitbucket.

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

Docker Trusted Registry (DTR): un servicio de registro de Docker (de Docker) que se puede instalar en las instalaciones para que viva dentro del centro de datos y la red de la organización. Es conveniente para las imágenes privadas que deben administrarse dentro de la empresa. Docker Trusted Registry se incluye como parte del producto Docker Datacenter.

Docker Desktop: herramientas de desarrollo para Windows y macOS para crear, ejecutar y probar contenedores localmente. Docker Desktop para Windows proporciona entornos de desarrollo para contenedores de Linux y Windows. El host de Linux Docker en Windows se basa en una máquina virtual [Hyper-V](#).

El host de los contenedores de Windows se basa directamente en Windows. Docker Desktop para Mac se basa en el marco Apple Hypervisor y el [hipervisor xhyve](#), que proporciona una máquina virtual host Linux Docker en macOS. Docker Desktop para Windows y Mac reemplaza a Docker Toolbox, que estaba basado en Oracle VirtualBox.

Compose: una herramienta de línea de comandos y formato de archivo YAML con metadatos para definir y ejecutar aplicaciones de varios contenedores. Usted define una sola aplicación basada en varias imágenes con uno o más archivos .yml que pueden anular los valores según el entorno. Después de haber creado las definiciones,

puede implementar toda la aplicación de varios contenedores con un solo comando (`docker-compose up`) que crea un contenedor por imagen en el host de Docker.

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

Orquestador: una herramienta que simplifica la gestión de clústeres y hosts Docker. Los orquestadores le permiten administrar sus imágenes, contenedores y hosts a través de una interfaz de línea de comandos (CLI) o una IU gráfica. Puede administrar las redes de contenedores, las configuraciones, el equilibrio de carga, el descubrimiento de servicios, la alta disponibilidad, la configuración del host de Docker y más. Un orquestador es responsable de ejecutar, distribuir, escalar y reparar las cargas de trabajo en una colección de nodos. Por lo general, los productos orquestadores son los mismos productos que brindan infraestructura de clúster, como Kubernetes y Azure Service Fabric, entre otras ofertas en el mercado.

## Contenedores Docker, imágenes y registros

Al usar Docker, un desarrollador crea una aplicación o servicio y lo empaqueta junto con sus dependencias en una imagen de contenedor. Una imagen es una representación estática de la aplicación o servicio y su configuración y dependencias.

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

Los desarrolladores deben almacenar imágenes en un registro, que actúa como una biblioteca de imágenes y es necesario cuando se implementa en orquestadores de producción. Docker mantiene un registro público a través de [Docker Hub](#); otros proveedores proporcionan registros para diferentes colecciones de imágenes, incluido [Azure Container Registry](#).

Como alternativa, las empresas pueden tener un registro privado local para sus propias imágenes de Docker.

La Figura 2-4 muestra cómo las imágenes y los registros en Docker se relacionan con otros componentes. También muestra las múltiples ofertas de registro de los proveedores.

## Basic taxonomy in Docker

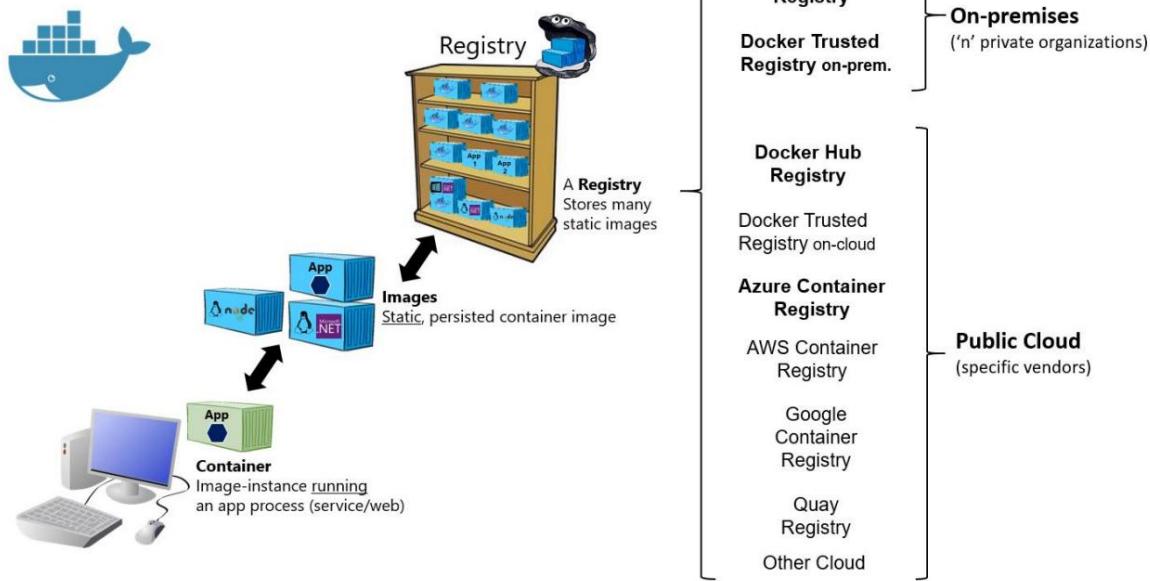


Figura 2-4. Taxonomía de términos y conceptos de Docker

El registro es como una estantería donde las imágenes se almacenan y están disponibles para extraerlas para construir contenedores para ejecutar servicios o aplicaciones web. Hay registros privados de Docker en las instalaciones y en la nube pública. Docker Hub es un registro público mantenido por Docker, junto con Docker Trusted Registry, una solución de nivel empresarial, Azure ofrece Azure Container Registry. AWS, Google y otros también tienen registros de contenedores.

Poner imágenes en un registro le permite almacenar bits de aplicación estáticos e inmutables, incluidas todas sus dependencias a nivel de marco. Luego, esas imágenes se pueden versionar e implementar en múltiples entornos y, por lo tanto, proporcionar una unidad de implementación consistente.

Los registros de imágenes privados, ya sea alojados en las instalaciones o en la nube, se recomiendan cuando:

- Sus imágenes no deben compartirse públicamente debido a la confidencialidad.
- Desea tener una latencia de red mínima entre sus imágenes y el entorno de implementación elegido. Por ejemplo, si su entorno de producción es la nube de Azure, probablemente desee almacenar sus imágenes en [Azure Container Registry](#) para que la latencia de la red sea mínima. De manera similar, si su entorno de producción es local, es posible que desee tener un Registro de confianza de Docker local disponible dentro de la misma red local.

# Elegir entre .NET 6 y .NET Framework para contenedores Docker

Hay dos marcos admitidos para crear aplicaciones Docker en contenedores del lado del servidor con .NET: .NET Framework y .NET 6. Comparten muchos componentes de la plataforma .NET y puede compartir código entre los dos. Sin embargo, existen diferencias fundamentales entre ellos, y el marco que utilice dependerá de lo que desee lograr. Esta sección proporciona orientación sobre cuándo elegir cada marco.

## Orientación general

Esta sección proporciona un resumen de cuándo elegir .NET 6 o .NET Framework. Proporcionamos más detalles sobre estas opciones en las secciones siguientes.

Utilice .NET 6, con contenedores de Linux o Windows, para su aplicación de servidor Docker en contenedores cuando:

- Tiene necesidades multiplataforma. Por ejemplo, desea utilizar Linux y Windows Contenedores.
- La arquitectura de su aplicación se basa en microservicios.
- Necesita comenzar a usar contenedores rápidamente y quiere un espacio físico pequeño por contenedor para lograr una mejor densidad o más contenedores por unidad de hardware para reducir sus costos.

En resumen, cuando crea nuevas aplicaciones .NET en contenedores, debe considerar .NET 6 como la opción predeterminada. Tiene muchos beneficios y encaja mejor con la filosofía y el estilo de trabajo de los contenedores.

Un beneficio adicional de usar .NET 6 es que puede ejecutar versiones de .NET en paralelo para aplicaciones dentro de la misma máquina. Este beneficio es más importante para servidores o máquinas virtuales que no usan contenedores, porque los contenedores aíslan las versiones de .NET que necesita la aplicación. (Siempre que sean compatibles con el sistema operativo subyacente).

Utilice .NET Framework para su aplicación de servidor Docker en contenedores cuando:

- Su aplicación actualmente usa .NET Framework y tiene fuertes dependencias en Windows.

- Debe usar las API de Windows que no son compatibles con .NET 6.
- Debe usar bibliotecas .NET de terceros o paquetes NuGet que no están disponibles para .NET 6.

El uso de .NET Framework en Docker puede mejorar sus experiencias de implementación al minimizar los problemas de implementación. Este [escenario de "lift and shift"](#) es importante para contenedores de aplicaciones heredadas que se desarrollaron originalmente con el .NET Framework tradicional, como ASP.NET WebForms, aplicaciones web MVC o servicios WCF (Windows Communication Foundation).

## Recursos adicionales

- Libro electrónico: Modernice las aplicaciones .NET Framework existentes con Azure y Windows Containers  
<https://aka.ms/liftandshiftwithcontainersebook>
- Aplicaciones de muestra: modernización de aplicaciones web ASP.NET heredadas mediante el uso de contenedores de Windows  
<https://aka.ms/eshopmodernizing>

## Cuándo elegir .NET para contenedores Docker

La naturaleza modular y liviana de .NET 6 lo hace perfecto para contenedores. Cuando implementa e inicia un contenedor, su imagen es mucho más pequeña con .NET 6 que con .NET Framework. Por el contrario, para 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 Windows Nano Server o Linux que usa para .NET 6.

Además, .NET 6 es multiplataforma, por lo que puede implementar aplicaciones de servidor con imágenes de contenedor de Linux o Windows. Sin embargo, si usa el .NET Framework tradicional, solo puede implementar imágenes basadas en Windows Server Core.

La siguiente es una explicación más detallada de por qué elegir .NET 6.

## Desarrollo y despliegue multiplataforma

Claramente, si su objetivo es tener una aplicación (aplicación web o servicio) que pueda ejecutarse en múltiples plataformas compatibles con Docker (Linux y Windows), la elección correcta es .NET 6, porque .NET Framework solo es compatible con Windows.

.NET 6 también es compatible con macOS como plataforma de desarrollo. Sin embargo, cuando implementa contenedores en un host Docker, ese host debe (actualmente) estar basado en Linux o Windows. Por ejemplo, en un entorno de desarrollo, podría usar una máquina virtual Linux que se ejecuta en una Mac.

[Visual Studio](#) proporciona un entorno de desarrollo integrado (IDE) para Windows y admite el desarrollo de 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. Esta herramienta debería ser la opción preferida para los desarrolladores que trabajan en máquinas Mac y que también desean utilizar un IDE potente.

También puede usar [Visual Studio Code](#) en macOS, Linux y Windows. Visual Studio Code es totalmente compatible con .NET 6, incluidos IntelliSense y depuración. Debido a que VS Code es un editor liviano, usted

puede usarlo para desarrollar aplicaciones en contenedores en la máquina junto con Docker CLI y .NET [CLI](#). También puede apuntar a .NET 6 con la mayoría de los editores de terceros, como Sublime, Emacs, vi y el proyecto de código abierto OmniSharp, que también brinda compatibilidad con IntelliSense.

Además de los IDE y los editores, puede usar la [CLI de .NET para todas las plataformas compatibles](#).

## Uso de contenedores para nuevos proyectos ("green-field")

Los contenedores se usan comúnmente junto con una arquitectura de microservicios, aunque también se pueden usar para contenedores de aplicaciones o servicios web que siguen cualquier patrón de arquitectura. Puede usar .NET Framework en contenedores de Windows, pero la naturaleza modular y liviana de .NET 6 lo hace perfecto para arquitecturas de contenedores y microservicios. Cuando crea e implementa un contenedor, su imagen es mucho más pequeña con .NET 6 que con .NET Framework.

## Cree e implemente microservicios en contenedores

Podría usar el .NET Framework tradicional para crear aplicaciones basadas en microservicios (sin contenedores) mediante procesos sencillos. De esa manera, debido a que .NET Framework ya está instalado y compartido entre procesos, los procesos son ligeros y rápidos para iniciar. Sin embargo, si usa contenedores, la imagen para .NET Framework tradicional también se basa en Windows Server Core y eso lo hace demasiado pesado para un enfoque de microservicios en contenedores. Sin embargo, los equipos también han estado buscando oportunidades para mejorar la experiencia de los usuarios de .NET Framework. Recientemente, el tamaño de las [imágenes del contenedor de Windows Server Core se ha reducido a >40 % más pequeño](#).

Por otro lado, .NET 6 es el mejor candidato si está adoptando un sistema orientado a microservicios que se basa en contenedores porque .NET 6 es liviano. Además, sus imágenes de contenedor relacionadas, ya sea para Linux o Windows Nano Server, son ligeras y pequeñas, lo que hace que los contenedores sean livianos y rápidos para comenzar.

Un microservicio está destinado a ser lo más pequeño posible: ser ligero cuando gira, tener una huella pequeña, tener un contexto limitado pequeño (consulte DDD, [diseño basado en dominio](#)), representar un área pequeña de preocupaciones y para poder arrancar y parar rápido. Para esos requisitos, querrá usar imágenes de contenedor pequeñas y rápidas de crear instancias como la imagen de contenedor de .NET 6.

Una arquitectura de microservicios también le permite mezclar tecnologías a través de un límite de servicio. Este enfoque permite una migración gradual a .NET 6 para nuevos microservicios 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 su sistema basado en contenedores necesita la mejor densidad, granularidad y rendimiento posibles, .NET y ASP.NET Core son sus mejores opciones. ASP.NET Core es hasta 10 veces más rápido que ASP.NET en el .NET Framework tradicional y conduce a otras tecnologías industriales populares para microservicios, como servlets de Java, Go y Node.js.

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

# Cuándo elegir .NET Framework para contenedores Docker

Si bien .NET 6 ofrece beneficios significativos para nuevas aplicaciones y patrones de aplicaciones, .NET Framework seguirá siendo una buena opción para muchos escenarios existentes.

## Migración de aplicaciones existentes directamente a un contenedor de Windows Server

Es posible que desee utilizar contenedores de Docker solo para simplificar la implementación, incluso si no está creando microservicios. Por ejemplo, tal vez desee mejorar su flujo de trabajo de DevOps con Docker: los contenedores pueden brindarle mejores entornos de prueba aislados y también pueden eliminar los problemas de implementación causados por dependencias faltantes cuando se traslada a un entorno de producción. En casos como estos, incluso si está implementando una aplicación monolítica, tiene sentido usar Docker y Windows Containers para sus aplicaciones .NET Framework actuales.

En la mayoría de los casos para este escenario, no necesitará migrar sus aplicaciones existentes a .NET 6; puede usar contenedores Docker que incluyen el .NET Framework tradicional. Sin embargo, un enfoque recomendado es usar .NET 6 a medida que amplía una aplicación existente, como escribir un nuevo servicio en ASP.NET Core.

## Uso de bibliotecas .NET de terceros o paquetes NuGet no disponibles para .NET 6

Las bibliotecas de terceros están adoptando rápidamente [.NET Standard](#), que permite compartir código en todas las variantes de .NET, incluido .NET 6. Con .NET Standard 2.0 y versiones posteriores, la compatibilidad de la superficie de la API entre diferentes marcos se ha vuelto significativamente mayor. Aún más, .NET Core 2.x y las aplicaciones más nuevas también pueden hacer referencia directamente a las bibliotecas de .NET Framework existentes ([consulte .NET Framework 4.6.1 compatible con .NET Standard 2.0](#)).

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

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

## Uso de tecnologías .NET no disponibles para .NET 6

Algunas tecnologías de .NET Framework no están disponibles en la versión actual de .NET (versión 5.0 al momento de escribir este artículo). Algunos de ellos pueden estar disponibles en versiones posteriores, pero otros no se ajustan a los nuevos patrones de aplicación a los que se dirige .NET Core y es posible que nunca estén disponibles.

La siguiente lista muestra la mayoría de las tecnologías que no están disponibles en .NET 6:

- Formularios Web ASP.NET. Esta tecnología solo está disponible en .NET Framework. Actualmente hay no hay planes para llevar ASP.NET Web Forms a .NET o posterior.
- Servicios WCF. Incluso cuando una [biblioteca WCF-Client](#) está disponible para consumir servicios WCF de .NET 6, a partir de enero de 2021, la implementación del servidor WCF solo está disponible en .NET Framework.
- Servicios relacionados con el flujo de trabajo. Windows Workflow Foundation (WF), Workflow Services (WCF + WF en un solo servicio) y WCF Data Services (anteriormente conocido como ADO.NET Data Services) solo están disponibles en .NET Framework. Actualmente no hay planes para llevarlos a .NET 6.

Además de las tecnologías enumeradas en la [hoja de ruta oficial de .NET](#), es posible que se transfieran otras funciones a la nueva [plataforma unificada de .NET](#). Podría considerar participar en las discusiones en GitHub para que su voz sea escuchada. Y si cree que falta algo, registre un nuevo problema en el repositorio de GitHub [dotnet/runtime](#).

---

## Usando una plataforma o API que no es compatible con .NET 6

Algunas plataformas de Microsoft y de terceros no son compatibles con .NET 6. Por ejemplo, algunos servicios de Azure proporcionan un SDK que aún no está disponible para su consumo en .NET 6. La mayoría de los SDK de Azure deberían migrarse eventualmente a .NET 6/.NET Standard, pero es posible que algunos no lo hagan por varios motivos. Puede ver los SDK de Azure disponibles en la página [Últimas versiones de SDK de Azure](#).

Mientras tanto, si alguna plataforma o servicio en Azure aún no es compatible con .NET 6 con su API de cliente, puede usar la API de REST equivalente del servicio de Azure o el SDK de cliente en .NET Framework.

## Portar la aplicación ASP.NET existente a .NET 6

.NET Core es un avance revolucionario desde .NET Framework. Ofrece una gran cantidad de ventajas sobre .NET Framework en todos los ámbitos, desde la productividad hasta el rendimiento, y desde el soporte multiplataforma hasta la satisfacción del desarrollador. Si usa .NET Framework y planea migrar su aplicación a .NET Core o .NET 5+, consulte [Migración de aplicaciones ASP.NET existentes a .NET Core](#).

---

## Recursos adicionales

- Fundamentos de .NET  
<https://docs.microsoft.com/dotnet/fundamentals>
  - Portar proyectos a .NET 5 <https://docs.microsoft.com/events/dotnetconf-2020/porting-projects-to-net-5>
  - .NET en la Guía de Docker  
<https://docs.microsoft.com/dotnet/core/docker/introduction>
- 

## Tabla de decisiones: marcos .NET para usar con Docker

La siguiente tabla de decisiones resume si usar .NET Framework o .NET 6. Recuerde que para contenedores Linux, necesita hosts Docker basados en Linux (VM o servidores) y que para contenedores Windows necesita hosts Docker basados en Windows Server (VM o servidores). ).

**Importante**

Sus máquinas de desarrollo ejecutarán un host Docker, ya sea Linux o Windows. Los microservicios relacionados que desea ejecutar y probar juntos en una solución deberán ejecutarse en la misma plataforma de contenedores.

Arquitectura/Tipo de aplicación	Contenedores Linux	Contenedores de Windows
Microservicios en contenedores	.NET 6	.NET 6
aplicación monolítica	.NET 6	.NET Framework .NET 6
El mejor rendimiento y escalabilidad de su clase	.NET 6	.NET 6
Migración de la aplicación heredada de Windows Server ("campo marrón") a contenedores	–	.NET Framework
Nuevo basado en contenedores desarrollo ("campo verde")	.NET 6	.NET 6
Núcleo de ASP.NET	.NET 6	.NET 6 (recomendado) .NET Framework
ASP.NET 4 (MVC 5, API web 2 y formularios web)	–	.NET Framework
Servicios SignalR	.NET Core 2.1 o versión superior	.NET Framework .NET Core 2.1 o versión superior
WCF, WF y otros marcos heredados	WCF en .NET Core (solo biblioteca cliente)	.NET Framework WCF en .NET 6 (solo biblioteca cliente)
Consumo de servicios de Azure .NET 6 (eventualmente, la mayoría de los servicios de Azure proporcionarán SDK de cliente para .NET 6)	eventualmente, la mayoría de los servicios de Azure proporcionarán SDK de cliente para .NET 6)	.NET Framework .NET 6 (eventualmente, la mayoría de los servicios de Azure proporcionarán SDK de cliente para .NET 6)

## Qué sistema operativo apuntar con contenedores .NET

Dada la diversidad de sistemas operativos compatibles con Docker y las diferencias entre .NET Framework y .NET 6, debe apuntar a un sistema operativo específico y versiones específicas según el marco que esté utilizando.

Para Windows, puede usar Windows Server Core o Windows Nano Server. Estas versiones de Windows ofrecen diferentes características (IIS en Windows Server Core frente a un servidor web autohospedado como Kestrel en Nano Server) que pueden necesitar .NET Framework o .NET 6, respectivamente.

Para Linux, varias distribuciones están disponibles y son compatibles con las imágenes oficiales de .NET Docker (como Debian).

En la Figura 3-1, puede ver la posible versión del sistema operativo según el marco .NET utilizado.

## What OS to target with .NET containers

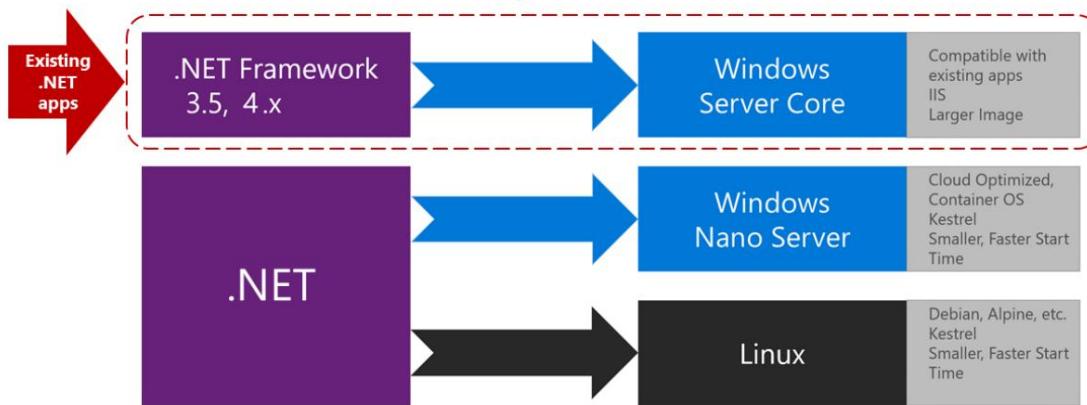


Figura 3-1. Sistemas operativos a los que apuntar dependiendo de las versiones del marco .NET

Al implementar aplicaciones heredadas de .NET Framework, debe apuntar a Windows Server Core, compatible con aplicaciones heredadas e IIS, pero tiene una imagen más grande. Al implementar aplicaciones .NET 6, puede apuntar a Windows Nano Server, que está optimizado para la nube, usa Kestrel, es más pequeño y se inicia más rápido. También puede apuntar a Linux, compatible con Debian, Alpine y otros. También usa Kestrel, es más pequeño y arranca más rápido.

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

Cuando agrega el nombre de la imagen a su archivo Dockerfile, puede seleccionar el sistema operativo y la versión según la etiqueta que use, como en los siguientes ejemplos:

Imagen	Comentarios
mcr.microsoft.com/dotnet/runtime:6.0	Arquitectura múltiple .NET 6: admite Linux y Windows Nano Server según el host de Docker.
mcr.microsoft.com/dotnet/aspnet:6.0	Arquitectura múltiple ASP.NET Core 6.0: admite Linux y Windows Nano Server según el host de Docker.  La imagen aspnetcore tiene algunas optimizaciones para ASP.NET Core.
mcr.microsoft.com/dotnet/aspnet:6.0-bullseye-slim	.NET 6 runtime-only en Linux Debian distro

Imagen	Comentarios
mcr.microsoft.com/dotnet/aspnet:6.0-nanoserver-1809	.NET 6 runtime solo en Windows Nano Server (Windows Server versión 1809)

## Imágenes oficiales de .NET Docker

Las imágenes oficiales de .NET Docker son imágenes de Docker creadas y optimizadas por Microsoft. 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, etc.).

Desde .NET Core 2.1, todas las imágenes de .NET Core o posteriores, incluidas las de ASP.NET Core, están disponibles en Docker Hub en el repositorio de imágenes de .NET: [https://hub.docker.com/\\_/microsoft-dotnet/](https://hub.docker.com/_/microsoft-dotnet/).

Desde mayo de 2018, las imágenes de Microsoft se distribuyen en Microsoft Container Registry. El catálogo oficial todavía solo está disponible en Docker Hub, y allí encontrará la dirección actualizada para extraer la imagen.

La mayoría de los repositorios de imágenes proporcionan un etiquetado extenso para ayudarlo a seleccionar no solo una versión específica del marco, sino también para elegir un sistema operativo (distribución de Linux o versión de Windows).

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

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

- Imágenes utilizadas para desarrollar y compilar aplicaciones .NET.
- Imágenes utilizadas para ejecutar aplicaciones .NET.

¿Por qué múltiples imágenes? Al desarrollar, crear y ejecutar aplicaciones en contenedores, por lo general, tiene diferentes prioridades. Al proporcionar diferentes imágenes para estas tareas separadas, Microsoft ayuda a optimizar los procesos separados de desarrollo, creación e implementación de aplicaciones.

### Durante el desarrollo y la construcción

Durante el desarrollo, lo importante es qué tan rápido puede iterar los cambios y la capacidad de depurar los cambios. El tamaño de la imagen no es tan importante como la capacidad de realizar cambios en el código y ver los cambios rápidamente. Algunas herramientas y "contenedores de agente de compilación" usan la imagen de desarrollo .NET ([mcr.microsoft.com/dotnet/sdk:6.0](https://mcr.microsoft.com/dotnet/sdk:6.0)) durante el proceso de desarrollo y compilación. Al construir dentro de un contenedor Docker, los aspectos importantes son los elementos que se necesitan para compilar su aplicación. Esto incluye el compilador y cualquier otra dependencia de .NET.

¿Por qué es importante este tipo de imagen de compilación? No implementa esta imagen en producción. En cambio, es una imagen que usa para crear el contenido que coloca en una imagen de producción. Esta imagen se usaría en su entorno de integración continua (CI) o entorno de compilación cuando use compilaciones de varias etapas de Docker.

## En producción

Lo importante en producción es qué tan rápido puede implementar e iniciar sus contenedores en función de una imagen .NET de producción. Por lo tanto, la imagen de solo tiempo de ejecución basada en mcr.microsoft.com/dotnet/aspnet:6.0 es pequeña para que pueda viajar rápidamente a través de la red desde su registro de Docker hasta sus hosts de Docker. El contenido está listo para ejecutarse, lo que permite el tiempo más rápido desde que se inicia el contenedor hasta que se procesan los resultados. En el modelo de Docker, no es necesario compilar desde el código C#, como ocurre cuando ejecuta dotnet build o dotnet Publish cuando usa el contenedor de compilación.

En esta imagen optimizada, coloca solo los binarios y otro contenido necesario para ejecutar la aplicación. Por ejemplo, el contenido creado por dotnet Publish contiene solo los archivos binarios, imágenes, .js y .css de .NET compilados. Con el tiempo, verá imágenes que contienen paquetes preestablecidos (la compilación de IL a nativo que ocurre en tiempo de ejecución).

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

Cuando explore 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, dependiendo de la versión que necesites, como las de la siguiente tabla:

Imagen	Comentarios
mcr.microsoft.com/dotnet/aspnet:6.0 ASP.NET	Core, solo con tiempo de ejecución y ASP.NET Core optimizaciones, en Linux y Windows (multi-arquitectura)
mcr.microsoft.com/dotnet/sdk:6.0	.NET 6, con SDK incluidos, en Linux y Windows (arquitectura múltiple)

Puede encontrar todas las imágenes de docker disponibles en [dotnet-docker](#) y también consultar las versiones preliminares más recientes mediante nightly build mcr.microsoft.com/dotnet/nightly/\*

# Arquitectura basada en contenedores y microservicios

## aplicaciones

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

Anteriormente en esta guía, aprendió conceptos básicos sobre contenedores y Docker. Esa información era lo mínimo que necesitaba para comenzar con los contenedores. Aunque los contenedores son habilitadores y se adaptan perfectamente a los microservicios, no son obligatorios para una arquitectura de microservicios. Muchos conceptos arquitectónicos en esta sección de arquitectura podrían aplicarse sin contenedores. Sin embargo, esta guía se enfoca en la intersección de ambos debido a la importancia ya introducida de los contenedores.

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

### Principios de diseño de contenedores

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

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

Si el proceso falla, el contenedor finaliza y el orquestador se hace cargo. Si el orquestador se configuró para mantener cinco instancias en ejecución y una falla, el orquestador creará otra instancia de contenedor para reemplazar el proceso fallido. En un trabajo por lotes, el proceso se inicia con parámetros. Cuando el proceso se completa, el trabajo está completo. Esta guía profundiza en los orquestadores, más adelante.

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

## Contenedorización de aplicaciones monolíticas

Es posible que desee crear una sola aplicación o servicio web implementado monolíticamente e implementarlo como un contenedor. Es posible que la aplicación en sí no sea internamente monolítica, sino estructurada como varias bibliotecas, componentes o incluso capas (capa de aplicación, capa de dominio, capa de acceso a datos, etc.). Sin embargo, externamente, es un solo contenedor: un solo proceso, una sola aplicación web o un solo servicio.

Para administrar este modelo, implementa un solo contenedor para representar la aplicación. Para aumentar la capacidad, escalas horizontalmente, es decir, solo agregas más copias con un balanceador de carga al frente. La simplicidad proviene de administrar una sola implementación en un solo contenedor o máquina virtual.

### Monolithic Containerized application

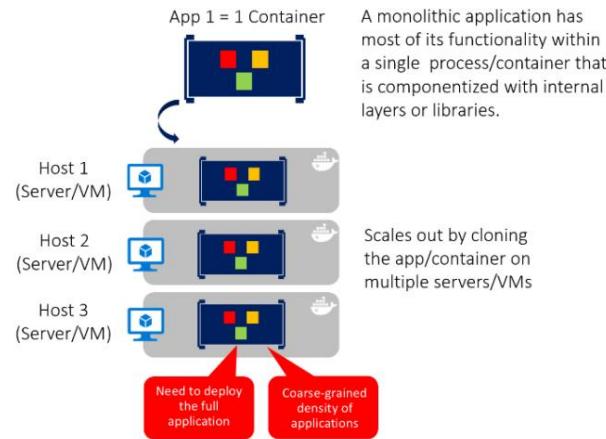


Figura 4-1. Ejemplo de la arquitectura de una aplicación monolítica en contenedores

Puede incluir múltiples componentes, bibliotecas o capas internas en cada contenedor, como se ilustra en la Figura 4-1. Una aplicación monolítica en contenedores tiene la mayor parte de su funcionalidad dentro de un solo contenedor, con capas o bibliotecas internas, y se escala mediante la clonación del contenedor en varios servidores/máquinas virtuales. Sin embargo, este patrón monolítico podría entrar en conflicto con el principio del contenedor "un contenedor hace una cosa y la hace en un proceso", pero podría estar bien para algunos casos.

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

Por ejemplo, en una aplicación típica de comercio electrónico, es probable que necesite escalar el subsistema de información del producto, porque muchos más clientes exploran los productos de los que los compran. Más clientes usan

su cesta que utilizar el canal de pago. Menos clientes agregan comentarios o ven su historial de compras. Y es posible que solo tenga un puñado de empleados que necesiten administrar el contenido y las campañas de marketing. Si escala el diseño monolítico, todo el código para estas diferentes tareas se implementa varias veces y se escala con el mismo grado.

Hay varias formas de escalar una duplicación horizontal de la aplicación, dividir diferentes áreas de la aplicación y particionar conceptos o datos comerciales similares. Pero, además del problema de escalar todos los componentes, los cambios en un solo componente requieren una nueva prueba completa de toda la aplicación y una redistribución completa de todas las instancias.

Sin embargo, el enfoque monolítico es común, porque el desarrollo de la aplicación es inicialmente más fácil que para los enfoques de microservicios. Por lo tanto, muchas organizaciones se desarrollan utilizando este enfoque arquitectónico. Si bien algunas organizaciones han obtenido resultados suficientemente buenos, otras están llegando al límite. Muchas organizaciones diseñaron sus aplicaciones utilizando este modelo porque las herramientas y la infraestructura dificultaron demasiado la creación de arquitecturas orientadas a servicios (SOA) hace años y no vieron la necesidad hasta que la aplicación creció.

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

## Host running multiple apps/containers

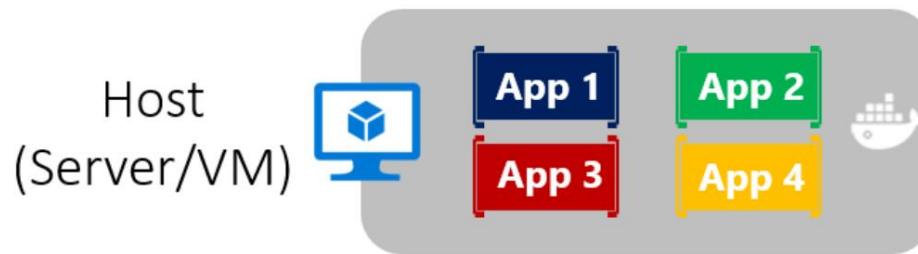


Figura 4-2. Enfoque monolítico: host que ejecuta múltiples aplicaciones, cada aplicación ejecutándose como un contenedor

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

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

## Architecture in Docker infrastructure for monolithic applications

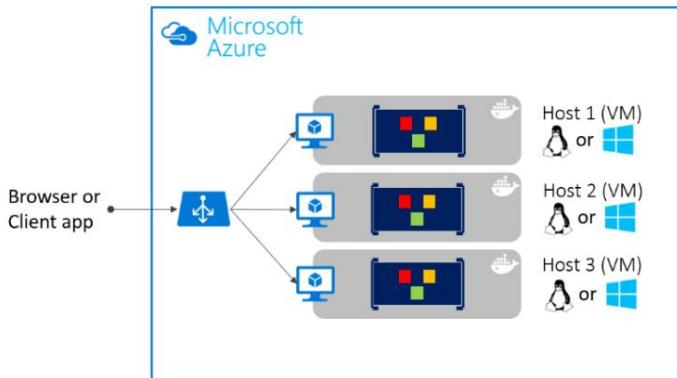


Figura 4-3. Ejemplo de varios hosts que amplían una sola aplicación de contenedor

La implementación en varios hosts se puede administrar con técnicas de implementación tradicionales. Los hosts de Docker se pueden administrar con comandos como `docker run` o `docker-compose` ejecutados manualmente, o a través de la automatización, como canalizaciones de entrega continua (CD).

### Implementación de una aplicación monolítica como contenedor

Hay beneficios en el uso de contenedores para administrar implementaciones de aplicaciones monolíticas. Escalar instancias de contenedores es mucho más rápido y fácil que implementar máquinas virtuales adicionales. Incluso si usa conjuntos de escalado de máquinas virtuales, las máquinas virtuales tardan 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 lo ideal.

La implementación de actualizaciones como imágenes de Docker es mucho más rápida y eficiente en la red. Las imágenes de Docker suelen comenzar en segundos, lo que acelera las implementaciones. Derribar una instancia de imagen de Docker es tan fácil como emitir un comando de detención de Docker y, por lo general, se completa en menos de un segundo.

Debido a que los contenedores son inmutables por diseño, nunca tendrá que preocuparse por las máquinas virtuales dañadas. Por el contrario, los scripts de actualización para una VM pueden olvidar tener en cuenta alguna configuración específica o archivo que queda en el disco.

Si bien las aplicaciones monolíticas pueden beneficiarse de Docker, solo mencionaremos los beneficios.

Los beneficios adicionales de administrar contenedores provienen de la implementación con orquestadores de contenedores, que administran las diversas instancias y el ciclo de vida de cada instancia de contenedor. Dividir la aplicación monolítica en subsistemas que se pueden escalar, desarrollar e implementar individualmente es su punto de entrada al ámbito de los microservicios.

### Publicación de una aplicación basada en un único contenedor en Azure App Service

Ya sea que desee 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 simple. Proporciona una excelente integración con Git para que sea más fácil tomar su código, compilarlo en Visual Studio e implementarlo directamente en Azure.

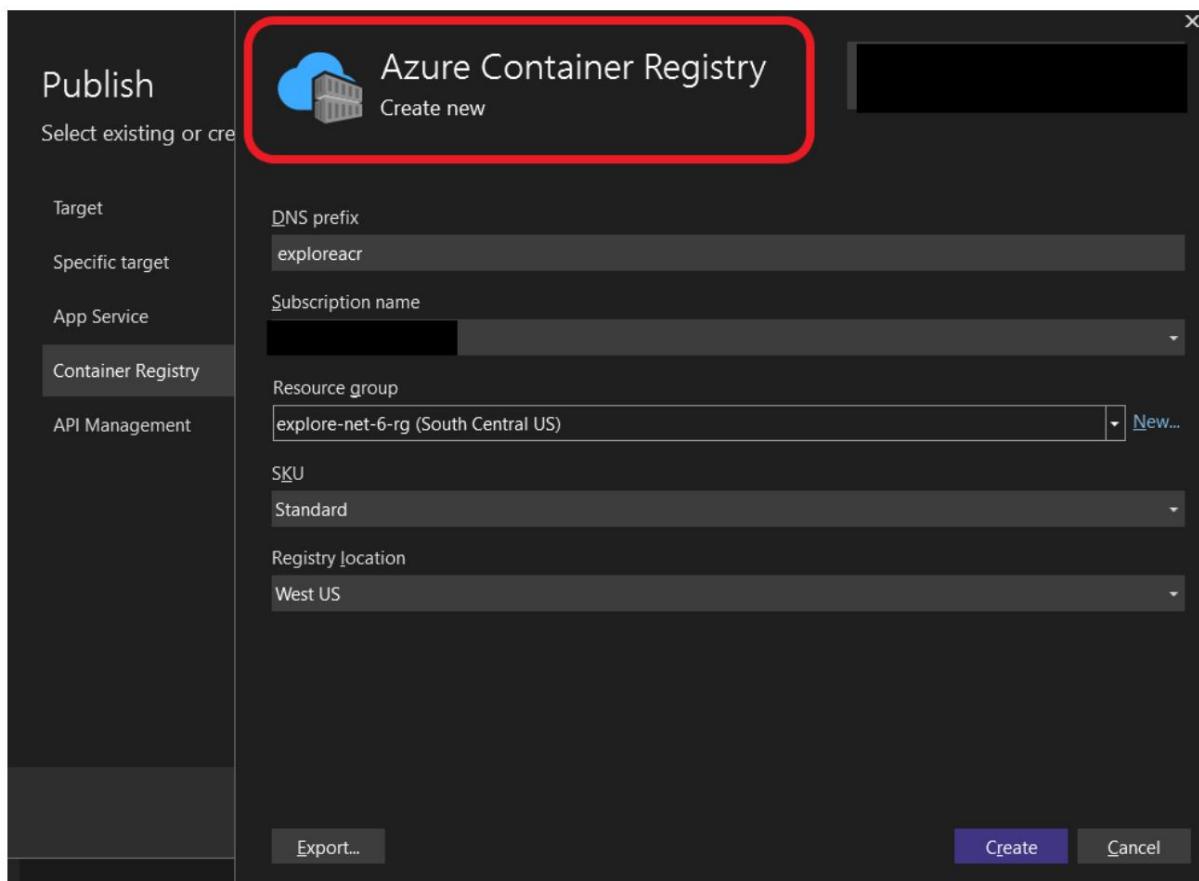


Figura 4-4. Publicación de una aplicación de un solo contenedor en Azure App Service desde Visual Studio 2022

Sin Docker, si necesitaba otras capacidades, marcos o dependencias que no son compatibles con Azure App Service, tenía que esperar hasta que el equipo de Azure actualizara esas dependencias en App Service. O tuvo que cambiar a otros servicios como Azure Cloud Services o VM, donde tenía más control y podía instalar un componente o marco requerido para su aplicación.

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

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

## Administre el estado y los datos en las aplicaciones de Docker

En la mayoría de los casos, puede pensar en un contenedor 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 existirá indefinidamente sería como suponer que una sola ubicación en la memoria será duradera. Tú

debe asumir que las imágenes de contenedor, como los procesos, tienen múltiples instancias o eventualmente se eliminarán. Si se administran con un orquestador de contenedores, debe suponer que es posible que se muevan de un nodo o máquina virtual a otro.

Las siguientes soluciones se utilizan para administrar datos en aplicaciones Docker:

Desde el host de Docker, como [Volúmenes de Docker](#):

- Los volúmenes se almacenan en un área del sistema de archivos del host que administra Docker.
- Los montajes de enlace pueden asignarse a cualquier carpeta en el sistema de archivos del host, por lo que el acceso no se puede controlar desde el proceso de Docker y puede representar un riesgo de seguridad, ya que un contenedor podría acceder a carpetas confidenciales del sistema operativo.
- Los montajes tmpfs son como carpetas virtuales que solo existen en la memoria del host y nunca se escriben al sistema de archivos.

Desde el almacenamiento remoto:

- [Azure Storage](#), que proporciona almacenamiento distribuible geográficamente, brinda una buena solución de persistencia a largo plazo para contenedores.
- Bases de datos relacionales remotas como [Azure SQL Database](#) o bases de datos NoSQL como [Azure Cosmos DB](#), o servicios de caché como [Redis](#).

Desde el contenedor Docker:

- Sistema de archivos de superposición. 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 está "encima" de la imagen original en la que se basa el contenedor. Si el contenedor se elimina del sistema, esos cambios se pierden. Por lo tanto, si bien es posible guardar el estado de un contenedor dentro de su almacenamiento local, diseñar un sistema en torno a esto entraría en conflicto con la premisa del diseño del contenedor, que por defecto no tiene estado.

Sin embargo, el uso de Docker Volumes ahora es la forma preferida de manejar datos locales en Docker. Si necesita más información sobre el almacenamiento en contenedores, consulte [los controladores de almacenamiento de Docker](#) y [Acerca de los controladores de almacenamiento](#).

A continuación se proporcionan más detalles sobre estas opciones:

Los volúmenes son directorios asignados desde el sistema operativo host a directorios en contenedores. Cuando el código en el contenedor tiene acceso al directorio, ese acceso es en realidad a un directorio en el sistema operativo anfitrión. Este directorio no está vinculado a la vida útil del contenedor en sí, y Docker administra el directorio y lo aisla de la funcionalidad principal de la máquina host. Por lo tanto, los volúmenes de datos están diseñados para conservar los datos independientemente de la vida útil del contenedor. Si elimina un contenedor o una imagen del host de Docker, los datos persistentes en el volumen de datos no se eliminan.

Los volúmenes pueden ser con nombre o anónimos (predeterminado). Los volúmenes con nombre son la evolución de los contenedores de volúmenes de datos y facilitan el intercambio 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 viven solo en la memoria del host y nunca se escriben en el sistema de archivos. Son rápidos y seguros, pero usan memoria y solo están destinados a datos temporales y no persistentes.

Como se muestra en la Figura 4-5, los volúmenes normales de Docker se pueden almacenar fuera de los contenedores, pero dentro de los límites físicos del servidor host o la VM. Sin embargo, los contenedores de Docker no pueden acceder a un volumen desde un servidor host o una máquina virtual a otro. En otras palabras, con estos volúmenes, no es posible administrar datos compartidos entre contenedores que se ejecutan en diferentes hosts de Docker, aunque podría lograrse con un controlador de volumen que admite hosts remotos.

## Data Volume and Data Volume Container

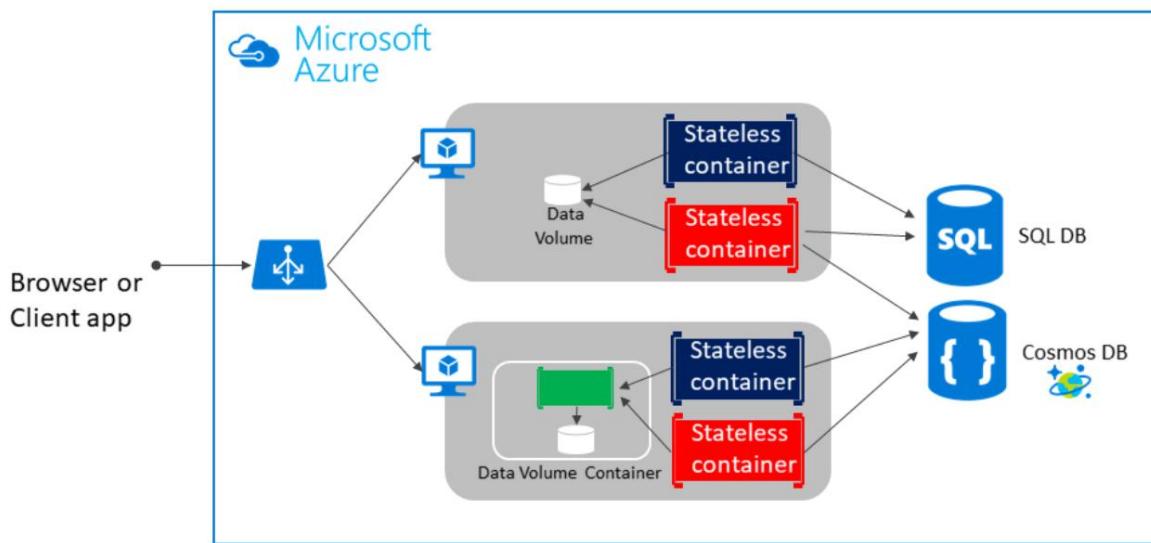


Figura 4-5. Volúmenes y orígenes de datos externos para aplicaciones basadas en contenedores

Los volúmenes se pueden compartir entre contenedores, pero solo en el mismo host, a menos que use un controlador remoto que admita hosts remotos. Además, cuando los contenedores Docker son administrados por un orquestador, los contenedores pueden "moverse" entre los hosts, según las optimizaciones realizadas por el clúster.

Por lo tanto, no se recomienda usar volúmenes de datos para datos comerciales. Pero son un buen mecanismo para trabajar con archivos de rastreo, archivos temporales o similares que no afectarán la consistencia de los datos comerciales.

Las fuentes de datos remotas y las herramientas de caché como Azure SQL Database, Azure Cosmos DB o una caché remota como Redis se pueden usar en aplicaciones en contenedores de la misma manera que se usan cuando se desarrolla sin contenedores. Esta es una forma comprobada de almacenar datos de aplicaciones comerciales.

Almacenamiento azul. Por lo general, los datos comerciales deberán colocarse en recursos o bases de datos externos, como Azure Storage. Azure Storage, en concreto, proporciona los siguientes servicios en la nube:

- Blob Storage almacena datos de objetos no estructurados. Un blob puede ser cualquier tipo de texto o datos binarios, como documentos o archivos multimedia (archivos de imágenes, audio y video). 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 y los servicios en la nube de Azure pueden compartir datos de archivos entre los componentes de la aplicación a través de recursos compartidos montados. Las aplicaciones locales pueden acceder a los datos de archivos en un recurso compartido a través de la API REST del servicio de archivos.
- El almacenamiento de tablas almacena conjuntos de datos estructurados. El almacenamiento de tablas es un almacén de datos de atributo clave NoSQL, que permite un desarrollo rápido y un acceso rápido a grandes cantidades de datos.

Bases de datos relacionales y bases de datos NoSQL. Hay muchas opciones para bases de datos externas, desde bases de datos relacionales como SQL Server, PostgreSQL, Oracle o bases de datos NoSQL como Azure Cosmos DB, MongoDB, etc. Estas bases de datos no se explicarán como parte de esta guía ya que están en un tema diferente

## Arquitectura orientada a Servicios

La arquitectura orientada a servicios (SOA) era un término usado en exceso y ha significado diferentes cosas para diferentes personas. Pero como denominador común, SOA significa que estructura su aplicación descomponiéndola en múltiples servicios (más comúnmente como servicios HTTP) que se pueden clasificar en diferentes tipos, como subsistemas o niveles.

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

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

Los microservicios se derivan de SOA, pero SOA es diferente de la arquitectura de microservicios. Las características como grandes intermediarios centrales, orquestadores centrales a nivel de organización y [Enterprise Service Bus \(ESB\)](#) son típicas en SOA. Pero en la mayoría de los casos, estos son antipatrones en la comunidad de microservicios. De hecho, algunas personas argumentan que "La arquitectura de microservicios es SOA bien hecha".

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

## Arquitectura de microservicios

Como su nombre lo indica, una arquitectura de microservicios es un enfoque para construir una aplicación de servidor como un conjunto de pequeños servicios. Eso significa que una arquitectura de microservicios está orientada principalmente al 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 específico de extremo a extremo o una capacidad empresarial dentro de un límite de contexto determinado, y cada uno debe desarrollarse de forma autónoma y desplegarse de forma independiente.

Finalmente, cada microservicio debe poseer su modelo de datos de dominio relacionado y su lógica de dominio (soberanía).

y gestión descentralizada de datos) y podría basarse en diferentes tecnologías de almacenamiento de datos (SQL, NoSQL) y diferentes lenguajes de programación.

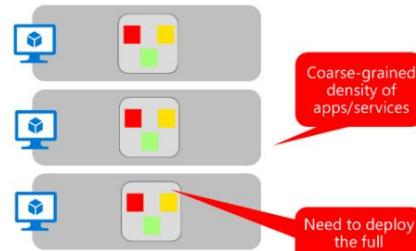
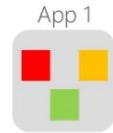
¿Qué tamaño debe tener un microservicio? Al desarrollar un microservicio, el tamaño no debería ser el punto importante. En su lugar, el punto importante debe ser crear servicios acoplados libremente para que tenga autonomía de desarrollo, implementación y escala para cada servicio. Por supuesto, al identificar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

¿Por qué una arquitectura de microservicios? En definitiva, proporciona agilidad a largo plazo. Los microservicios permiten una mejor capacidad de mantenimiento en sistemas complejos, grandes y altamente escalables al permitirle crear aplicaciones basadas en muchos servicios de implementación independiente que tienen ciclos de vida granulares y autónomos.

Como beneficio adicional, los microservicios se pueden escalar de manera independiente. En lugar de tener una sola aplicación monolítica que debe escalar horizontalmente como una unidad, puede escalar microservicios específicos. De esa forma, puede escalar solo el área funcional que necesita más potencia de procesamiento o ancho de banda de red para satisfacer la demanda, en lugar de escalar otras áreas de la aplicación que no necesitan escalarse. Eso significa ahorro de costos 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

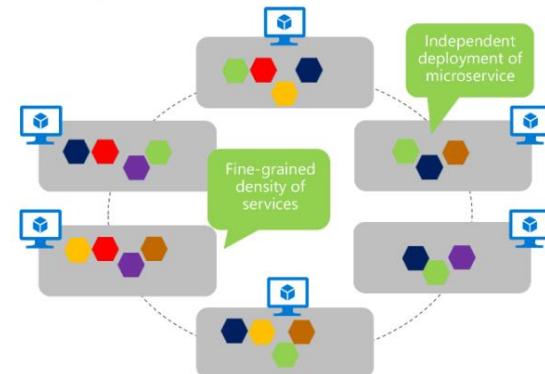
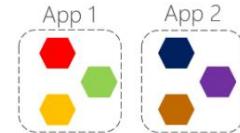


Figura 4-6. Implementación monolítica frente al enfoque de microservicios

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

La arquitectura de aplicaciones basadas en microservicios de granularidad fina permite la integración continua y las prácticas de entrega continua. También acelera la entrega de nuevas funciones en la aplicación. La composición detallada de las aplicaciones también le permite ejecutar y probar microservicios de forma aislada y

evolucionarlos de forma autónoma manteniendo contratos claros entre ellos. Mientras no cambie las interfaces o los contratos, puede cambiar la implementación interna de cualquier microservicio o agregar una nueva funcionalidad sin interrumpir otros microservicios.

Los siguientes son aspectos importantes para permitir el éxito al entrar en producción con un sistema basado en microservicios:

- Seguimiento y chequeos de salud de los servicios e infraestructuras.
- Infraestructura escalable para los servicios (es decir, nube y orquestadores).
- Diseño e implementación de seguridad a múltiples niveles: autenticación, autorización, gestión de secretos, comunicación segura, etc.
- Entrega rápida de aplicaciones, generalmente con diferentes equipos que se enfocan en diferentes microservicios.
- Prácticas e infraestructura de DevOps y CI/CD.

De estos, solo los primeros tres están cubiertos o presentados en esta guía. 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 la aplicación Docker en contenedor con la plataforma y las herramientas de Microsoft](#).

## Recursos adicionales

- Mark Russinovich. Microservicios: una revolución de aplicaciones impulsada por la nube  
<https://azure.microsoft.com/blog/microservices-an-application-revolution-powered-by-the-cloud/>


---
- Martín Fowler. microservicios  
<https://www.martinfowler.com/articles/microservices.html>


---
- Martín Fowler. Requisitos previos del microservicio  
<https://martinfowler.com/bliki/MicroservicePrerequisites.html>


---
- Jimmy Nilsson. Computación en la nube  
<https://www.infoq.com/articles/CCC-Jimmy-Nilsson>


---
- César de la Torre. Ciclo de vida de la aplicación Docker en contenedores con la plataforma y las herramientas de Microsoft (libro electrónico descargable) <https://aka.ms/dockerlifecyclebook>


---

## Soberanía de datos por microservicio

Una regla importante para la arquitectura de microservicios es que cada microservicio debe poseer su propia lógica y datos de dominio. Así como una aplicación completa posee su lógica y datos, cada microservicio debe poseer su lógica y datos bajo un ciclo de vida autónomo, con implementación independiente por microservicio.

Esto significa que el modelo conceptual del dominio diferirá entre subsistemas o microservicios.

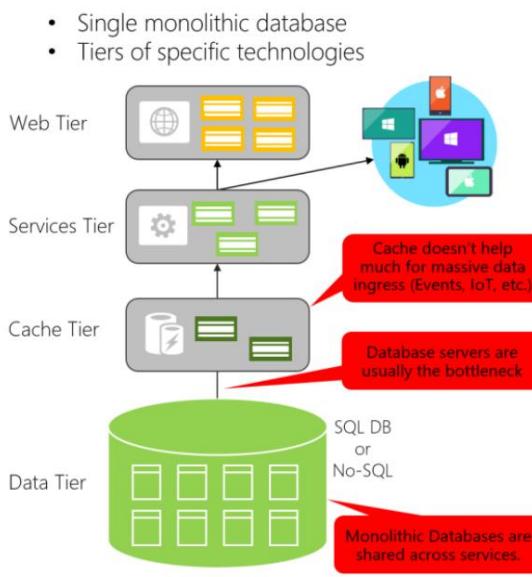
Considere las aplicaciones empresariales, donde las aplicaciones de gestión de relaciones con los clientes (CRM),

los subsistemas de compra transaccional y los subsistemas de atención al cliente requieren atributos y datos únicos de la entidad del cliente, y cada uno emplea un contexto delimitado (BC) diferente.

Este principio es similar en [el diseño basado en dominios \(DDD\)](#), donde cada [contexto delimitado](#) o subsistema o servicio autónomo debe poseer su modelo de dominio (datos más lógica y comportamiento). Cada DDD Bounded Context se correlaciona con un microservicio empresarial (uno o varios servicios). Este punto sobre el patrón Bounded Context se amplía en la siguiente sección.

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

### Data in Traditional approach



### Data in Microservices approach

- Graph of interconnected microservices
  - State typically scoped to the microservice
  - Remote Storage for cold data
- 

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

En el enfoque tradicional, hay una única base de datos compartida entre todos los servicios, normalmente en una arquitectura en niveles. En el enfoque de microservicios, cada microservicio posee su modelo/datos. El enfoque de la base de datos centralizada inicialmente parece más simple y parece permitir la reutilización de entidades en diferentes subsistemas para que todo sea coherente. Pero la realidad es que terminas con tablas enormes que sirven a muchos subsistemas diferentes y que incluyen atributos y columnas que no son necesarios en la mayoría de los casos. Es como tratar de usar el mismo mapa físico para caminar por un sendero corto, hacer un viaje en automóvil de un día y aprender geografía.

Una aplicación monolítica con normalmente una sola base de datos relacional tiene dos beneficios importantes: [las transacciones ACID](#) y el lenguaje SQL, ambos funcionan en todas las tablas y datos relacionados con su aplicación.

Este enfoque proporciona una forma de escribir fácilmente una consulta que combina datos de varias tablas.

Sin embargo, el acceso a los datos se vuelve mucho más complicado cuando se cambia a una arquitectura de microservicios. Incluso cuando se utilizan transacciones ACID dentro de un microservicio o contexto delimitado, es fundamental tener en cuenta que los datos que posee cada microservicio son privados para ese microservicio y solo deben

se puede acceder de forma síncrona a través de sus puntos finales de API (REST, gRPC, SOAP, etc.) o de forma asíncrona a través de mensajería (AMQP o similar).

Encapsular los datos garantiza que los microservicios se acoplen libremente y puedan evolucionar independientemente unos de otros. Si varios servicios accedieran a los mismos datos, las actualizaciones del esquema requerirían actualizaciones coordinadas para todos los servicios. Esto rompería la autonomía del ciclo de vida del microservicio. Pero las estructuras de datos distribuidas significan que no puede realizar una sola transacción ACID entre microservicios. Esto, a su vez, significa que debe utilizar la coherencia eventual cuando un proceso empresarial abarca varios microservicios. Esto es mucho más difícil de implementar que las uniones SQL simples, porque no puede crear restricciones de integridad o usar transacciones distribuidas entre bases de datos separadas, como explicaremos más adelante. Del mismo modo, muchas otras funciones de bases de datos relacionales no están disponibles en varios microservicios.

Yendo aún más lejos, los diferentes microservicios a menudo usan diferentes tipos de bases de datos. Las aplicaciones modernas almacenan y procesan diversos tipos de datos, y una base de datos relacional no siempre es la mejor opción. Para algunos casos de uso, una base de datos NoSQL como Azure CosmosDB o MongoDB puede tener un modelo de datos más conveniente y ofrecer 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 a menudo usan una combinación de bases de datos SQL y NoSQL, lo que a veces se denomina [enfoque de persistencia políglota](#).

Una arquitectura particionada y persistente de políglota para el almacenamiento de datos tiene muchos beneficios. Estos incluyen servicios débilmente acoplados y mejor rendimiento, escalabilidad, costos y capacidad de administración. Sin embargo, puede presentar algunos desafíos de administración de datos distribuidos, como se explica en "[Identificación de los límites del modelo de dominio](#)" más adelante en este capítulo.

## La relación entre los microservicios y el patrón Bounded Context

El concepto de microservicio se deriva del [patrón Bounded Context \(BC\) en el diseño controlado por dominio \(DDD\)](#). DDD trata con modelos grandes dividiéndolos en múltiples BC y siendo explícito acerca de sus límites. Cada BC debe tener su propio modelo y base de datos; asimismo, cada microservicio posee sus datos relacionados. Además, cada BC suele tener su propio [lenguaje ubicuo](#) para facilitar la comunicación entre los desarrolladores de software y los expertos del dominio.

Esos términos (principalmente entidades de dominio) en el lenguaje ubicuo pueden tener diferentes nombres en diferentes contextos delimitados, incluso cuando diferentes entidades de dominio comparten la misma identidad (es decir, la identificación única que se usa para leer la entidad del almacenamiento). Por ejemplo, en un contexto acotado de perfil de usuario, la entidad de dominio del usuario podría compartir identidad con la entidad de dominio del comprador en el contexto acotado que realiza el pedido.

Por lo tanto, un microservicio es como un contexto acotado, pero también especifica que es un servicio distribuido.

Está construido como un proceso separado para cada contexto delimitado y debe usar los protocolos distribuidos mencionados anteriormente, como HTTP/HTTPS, WebSockets o AMQP. El [patrón de Bounded Context](#), sin embargo, no especifica si Bounded Context es un servicio distribuido o si es simplemente un límite lógico (como un subsistema genérico) dentro de una aplicación de implementación monolítica.

Es importante resaltar que definir un servicio para cada contexto delimitado es un buen punto de partida.

Pero no tienes que limitar tu diseño a ello. A veces debe diseñar un contexto acotado o

Microservicio empresarial compuesto por varios servicios físicos. Pero en definitiva, ambos patrones -Bounded Context y microservicio- están íntimamente 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 lo que también desea en un contexto acotado.

## Recursos adicionales

- Chris Richardson. Patrón: Base de datos por servicio <https://microservices.io/patterns/data/database-per-service.html>
- Martín Fowler. BoundedContext <https://martinfowler.com/bliki/BoundedContext.html>
- Martín Fowler. Persistencia políglota <https://martinfowler.com/bliki/PolyglotPersistence.html>
- Alberto Brandolini. Diseño estratégico impulsado por dominio con mapeo de contexto <https://www.infoq.com/articles/ddd-contextmapping>

## Arquitectura lógica versus arquitectura física

Es útil en este punto detenerse y analizar la distinción entre arquitectura lógica y arquitectura física, y cómo se aplica esto 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 podrían ejecutarse como procesos simples. Los microservicios son una arquitectura lógica.

Además, incluso cuando un microservicio podría implementarse físicamente como un solo servicio, proceso o contenedor (en aras de la simplicidad, ese es el enfoque adoptado en la versión inicial de [eShopOnContainers](#)), esta paridad entre el microservicio comercial y el servicio o contenedor físico no es necesariamente. Se requiere en todos los casos cuando crea una aplicación grande y compleja compuesta por muchas docenas o incluso cientos de servicios.

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

Aunque es posible que haya identificado determinados microservicios comerciales o Bounded Contexts, eso no significa que la mejor manera de implementarlos siempre sea mediante la creación de un único servicio (como una API web de ASP.NET) o un único contenedor de Docker para cada microservicio comercial. Tener una regla que diga que cada microservicio comercial debe implementarse utilizando un solo servicio o contenedor es demasiado rígido.

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

Como muestra la Figura 4-8, el microservicio comercial de catálogo podría estar compuesto por varios servicios o procesos. Estos podrían ser múltiples servicios de API web ASP.NET o cualquier otro tipo de servicios que utilicen HTTP o cualquier otro protocolo. Más importante aún, los servicios podrían compartir los mismos datos, siempre que estos servicios sean coherentes con respecto al mismo dominio empresarial.

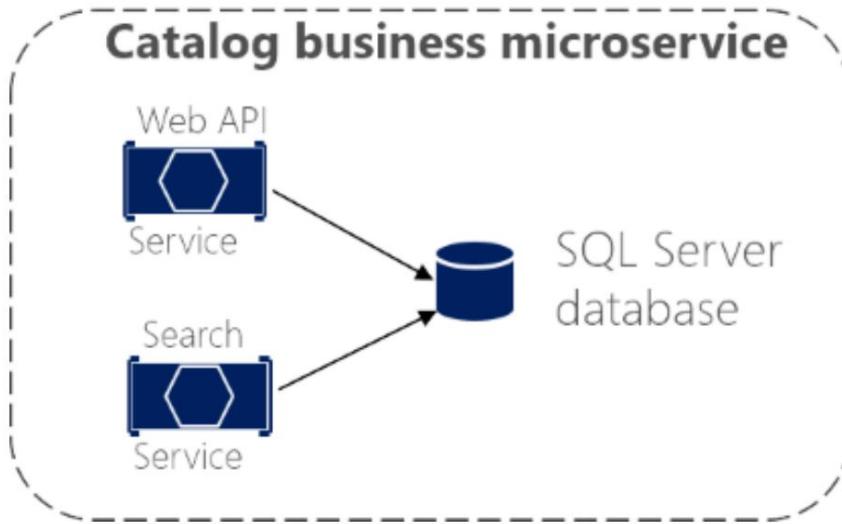


Figura 4-8. Microservicio empresarial con varios servicios físicos

Los servicios del ejemplo comparten el mismo modelo de datos porque el servicio Web API se dirige a los mismos datos que el servicio de búsqueda. Por lo tanto, en la implementación física del microservicio comercial, está dividiendo esa funcionalidad para que pueda escalar cada uno de esos servicios internos hacia arriba o hacia abajo según sea necesario. Tal vez el servicio Web API normalmente necesite más instancias que el servicio de búsqueda, o viceversa.

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

## Desafíos y soluciones para la gestión de datos distribuidos

### Desafío #1: Cómo definir los límites de cada microservicio

Definir los límites de los microservicios es probablemente el primer desafío al que se enfrenta cualquier persona. Cada microservicio debe ser una parte de su aplicación y cada microservicio debe ser autónomo con todos los beneficios y desafíos que conlleva. Pero, ¿cómo identificar esos límites?

En primer lugar, debe centrarse en los modelos de dominio lógico de la aplicación y los datos relacionados. Intente identificar islas desacopladas de datos y diferentes contextos dentro de la misma aplicación. Cada contexto podría tener un lenguaje comercial diferente (diferentes términos comerciales). Los contextos deben definirse y gestionarse de forma independiente. Los términos y entidades que se usan en esos diferentes contextos pueden sonar similares, pero es posible que descubra que en un contexto particular, un concepto de negocios con uno se usa para un concepto diferente.

propósito en otro contexto, e incluso podría tener un nombre diferente. Por ejemplo, se puede hacer referencia a un usuario como usuario en el contexto de identidad o pertenencia, como cliente en un contexto de CRM, como comprador en un contexto de pedido, etc.

La forma en que identifica los límites entre múltiples contextos de aplicaciones con un dominio diferente para cada contexto es exactamente cómo puede identificar los límites para cada microservicio comercial y su modelo de dominio y datos relacionados. Siempre intenta minimizar el acoplamiento entre esos microservicios. Esta guía entra en más detalles sobre esta identificación y el diseño del modelo de dominio en la sección [Identificación de los límites del modelo de dominio para cada microservicio](#) más adelante.

## Desafío #2: Cómo crear consultas que recuperen datos de varios microservicios

Un segundo desafío es cómo implementar consultas que recuperen datos de varios microservicios, mientras se evita la comunicación conversacional con los microservicios desde aplicaciones de clientes remotos. Un ejemplo podría ser una sola pantalla de una aplicación móvil que necesita mostrar la información del usuario que pertenece a los microservicios de cesta, catálogo e identidad del usuario. Otro ejemplo sería un informe complejo que involucra muchas tablas ubicadas en múltiples microservicios. La solución adecuada depende de la complejidad de las consultas. Pero en cualquier caso, necesitará una forma de agregar información si desea mejorar la eficiencia en las comunicaciones de su sistema. Las soluciones más populares son las siguientes.

Puerta de enlace API. Para la agregación de datos simple de varios microservicios que poseen diferentes bases de datos, el enfoque recomendado es un microservicio de agregación denominado API Gateway. Sin embargo, debe tener cuidado al implementar este patrón, ya que puede ser un cuello de botella en su sistema y puede violar el principio de autonomía de los microservicios. Para mitigar esta posibilidad, puede tener varias puertas de enlace de API de granularidad fina, cada una de las cuales se centra en un "segmento" vertical o área comercial del sistema. El patrón API Gateway se explica con más detalle en la [sección API Gateway](#) más adelante.

CQRS con tablas de consulta/lectura. Otra solución para agregar datos de múltiples microservicios es el [patrón de vista materializada](#). En este enfoque, genera, por adelantado (prepara los datos desnormalizados antes de que ocurran las consultas reales), una tabla de solo lectura con los datos que pertenecen a varios microservicios. La tabla tiene un formato adecuado a las necesidades de la aplicación cliente.

Considere algo como la pantalla de una aplicación móvil. Si tiene una sola base de datos, puede reunir los datos para esa pantalla mediante una consulta SQL que realiza una combinación compleja que involucra varias tablas. Sin embargo, cuando tiene varias bases de datos y cada base de datos es propiedad de un microservicio diferente, no puede consultar esas bases de datos y crear una unión SQL. Tu consulta compleja se convierte en un reto. Puede abordar el requisito utilizando un enfoque CQRS: crea una tabla desnormalizada en una base de datos diferente que se usa solo para consultas. La tabla se puede diseñar específicamente para los datos que necesita para la consulta compleja, con una relación de uno a uno entre los campos que necesita la pantalla de su aplicación y las columnas de la tabla de consulta. También podría servir para fines informativos.

Este enfoque no solo resuelve el problema original (cómo consultar y unir microservicios), sino que también mejora considerablemente el rendimiento en comparación con una unión compleja, porque ya tiene los datos que necesita la aplicación en la tabla de consulta. Por supuesto, el uso de la Segregación de responsabilidad de comandos y consultas (CQRS) con tablas de consulta/lectura significa un trabajo de desarrollo adicional, y deberá aceptar la coherencia final. No obstante, los requisitos de rendimiento y alta

la escalabilidad en [escenarios colaborativos](#) (o escenarios competitivos, según el punto de vista) es donde se debe aplicar CQRS con múltiples bases de datos.

"Datos fríos" en bases de datos centrales. Para informes y consultas complejas que pueden no requerir datos en tiempo real, un enfoque común es exportar sus "datos calientes" (datos transaccionales de los microservicios) como "datos fríos" en grandes bases de datos que se usan solo para informes. Ese sistema de base de datos central puede ser un sistema basado en Big Data, como Hadoop, un almacén de datos como uno basado en Azure SQL Data Warehouse, o incluso una sola base de datos SQL que se usa solo para informes (si el tamaño no es un problema).

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

Sin embargo, si el diseño de su aplicación implica la agregación constante de información de múltiples microservicios para consultas complejas, podría ser un síntoma de un mal diseño: un microservicio debe estar lo más aislado posible de otros microservicios. (Esto excluye los informes/análisis que siempre deben usar bases de datos centrales de datos fríos). Tener este problema a menudo puede ser una razón para fusionar microservicios. Debe equilibrar la autonomía de la evolución y la implementación de cada microservicio con fuertes dependencias, cohesión y agregación de datos.

## Desafío n.º 3: Cómo lograr la coherencia entre múltiples microservicios

Como se indicó anteriormente, los datos propiedad de cada microservicio son privados para ese microservicio y solo se puede acceder a ellos mediante su API de microservicio. Por lo tanto, un desafío que se presenta es cómo implementar procesos comerciales de extremo a extremo mientras se mantiene la coherencia en múltiples microservicios.

Para analizar este problema, veamos un ejemplo de la aplicación de [referencia eShopOnContainers](#).

---

El microservicio Catálogo mantiene información sobre todos los productos, incluido el precio del producto.

El microservicio Basket administra datos temporales sobre los elementos de productos que los usuarios agregan a sus carritos de compras, lo que incluye el precio de los artículos en el momento en que se agregaron a la cesta.

Cuando el precio de un producto se actualiza en el catálogo, ese precio también debe actualizarse en las cestas activas que contienen ese mismo producto, además, el sistema probablemente debería advertir al usuario que el precio de un artículo en particular ha cambiado desde que lo agregó a su cesta.

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

Sin embargo, en una aplicación basada en microservicios, las tablas Product y Basket pertenecen a sus respectivos microservicios. Ningún microservicio debería incluir tablas/almacenamiento propiedad de otro microservicio en sus propias transacciones, ni siquiera en consultas directas, como se muestra en la Figura 4-9.

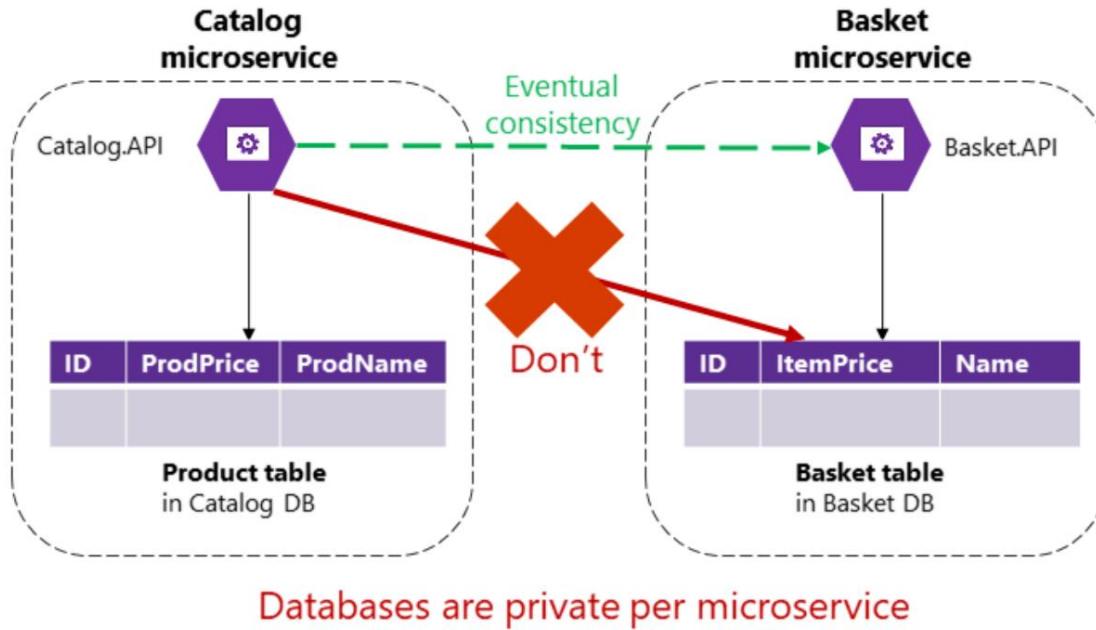


Figura 4-9. Un microservicio no puede acceder directamente a una tabla en otro microservicio

El microservicio Catalog no debe actualizar la tabla Basket directamente, porque la tabla Basket es propiedad del microservicio Basket. Para realizar una actualización del microservicio Basket, el microservicio Catalog debe usar una coherencia final probablemente basada en comunicación asíncrona, como eventos de integración (comunicación basada en mensajes y eventos). Así es como la aplicación de referencia [eShopOnContainers](#) realiza este tipo de coherencia entre microservicios.

Como establece el [teorema CAP](#), debe elegir entre disponibilidad y consistencia fuerte de ACID.

La mayoría de los escenarios basados en microservicios exigen disponibilidad y alta escalabilidad en lugar de una gran coherencia. Las aplicaciones de misión crítica deben permanecer activas y en funcionamiento, y los desarrolladores pueden trabajar con una coherencia sólida mediante el uso de técnicas para trabajar con una coherencia débil o eventual. Este es el enfoque adoptado por la mayoría de las arquitecturas basadas en microservicios.

Además, las transacciones de confirmación de dos fases o estilo ACID no solo van en contra de los principios de los microservicios; la mayoría de las bases de datos NoSQL (como Azure Cosmos DB, MongoDB, etc.) no admiten transacciones de confirmación en dos fases, típicas en escenarios de bases de datos distribuidas. Sin embargo, 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 propagar cambios a través de múltiples microservicios cuando ciertos datos deben ser redundantes, por ejemplo, cuando necesita tener el nombre o la descripción del producto en el microservicio Catálogo y el microservicio Cesta.

Una buena solución para este problema es utilizar la coherencia eventual entre los microservicios articulados a través de la comunicación basada en eventos y un sistema de publicación y suscripción. Estos temas se tratan en la sección [Comunicación asincrónica basada en eventos](#) más adelante en esta guía.

## Desafío n.º 4: cómo diseñar la comunicación a través de los límites de los microservicios

La comunicación a través de los límites de los microservicios es un verdadero desafío. En este contexto, la comunicación no se refiere a qué protocolo debe usar (HTTP y REST, AMQP, mensajería, etc.). En cambio, aborda qué estilo de comunicación debe usar y, especialmente, qué tan acoplados deben estar sus microservicios. Dependiendo del nivel de acoplamiento, cuando ocurre una falla, el impacto de esa falla en su sistema variará significativamente.

En un sistema distribuido como una aplicación basada en microservicios, con tantos artefactos moviéndose y con servicios distribuidos en muchos servidores o hosts, los componentes eventualmente fallarán. Se producirán fallas parciales e interrupciones aún mayores, por lo que debe diseñar sus microservicios y la comunicación entre ellos teniendo en cuenta los riesgos comunes en este tipo de sistema distribuido.

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

Por ejemplo, imagine que su aplicación cliente realiza una llamada API HTTP a un microservicio individual como el microservicio Pedidos. Si el microservicio Pedidos a su vez llama a microservicios adicionales mediante HTTP dentro del mismo ciclo de solicitud/respuesta, está creando una cadena de llamadas HTTP. Puede sonar razonable inicialmente. Sin embargo, hay puntos importantes a considerar al seguir este camino:

- Bloqueo y bajo rendimiento. Debido a la naturaleza síncrona de HTTP, la solicitud original no recibe una respuesta hasta que finalizan todas las llamadas HTTP internas. Imagínese si el número de estas llamadas aumenta significativamente y al mismo tiempo se bloquea una de las llamadas HTTP intermedias a un microservicio. El resultado es que el rendimiento se ve afectado y la escalabilidad general se verá afectada exponencialmente a medida que aumentan las solicitudes HTTP adicionales.
- Acoplamiento de microservicios con HTTP. Los microservicios empresariales no deben combinarse con otros microservicios empresariales. Idealmente, no deberían “saber” sobre la existencia de otros microservicios. Si su aplicación se basa en el acoplamiento de microservicios como en el ejemplo, será casi imposible lograr la autonomía por microservicio.
- Fallo en cualquier microservicio. Si implementó una cadena de microservicios vinculados por llamadas HTTP, cuando cualquiera de los microservicios falla (y eventualmente fallará), toda la cadena de microservicios fallará. Un sistema basado en microservicios debe diseñarse para continuar funcionando lo mejor posible durante fallas parciales. Incluso si implementa una lógica de cliente que utiliza reintentos con retroceso exponencial o mecanismos de disyuntores, cuanto más complejas sean las cadenas de llamadas HTTP, más complejo será implementar una estrategia de falla basada en HTTP.

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

Por lo tanto, para hacer cumplir la autonomía de los microservicios y tener una mejor resiliencia, debe minimizar el uso de cadenas de comunicación de solicitud/respuesta en los microservicios. Se recomienda que use solo la interacción asíncrona para la comunicación entre microservicios, ya sea mediante el uso de comunicación asíncrona basada en mensajes y eventos, o mediante el uso de sondeo HTTP (asíncrono) independientemente del ciclo de solicitud/respuesta HTTP original.

El uso de la comunicación asíncrona se explica con detalles adicionales más adelante en esta guía, en las secciones [La integración de microservicios asíncronos impone la autonomía de los microservicios](#) y [Comunicación asíncrona basada en mensajes](#).

## Recursos adicionales

- Teorema de la PAC  
[https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem)
- Coherencia eventual  
[https://en.wikipedia.org/wiki/Eventual\\_consistency](https://en.wikipedia.org/wiki/Eventual_consistency)
- Introducción a la coherencia  
de datos [https://docs.microsoft.com/previous-versions/msp-np/dn589800\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-np/dn589800(v=pandp.10))
- Martín Fowler. CQRS (Segregación de responsabilidad de comandos y consultas)  
<https://martinfowler.com/bliki/CQRS.html>
- Vista materializada  
<https://docs.microsoft.com/azure/architecture/patterns/materialized-view>
- Carlos Fila. ACID vs. BASE: El pH cambiante 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. Composición orientada a servicios  
<https://udidahan.com/2014/07/30/service-oriented-composition-with-video/>

## Identifique los límites del modelo de dominio para cada microservicio

El objetivo al identificar los límites del modelo y el tamaño de cada microservicio no es llegar a la separación más granular posible, aunque debe tender hacia microservicios pequeños si es posible.

En cambio, su objetivo debe ser llegar a la separación más significativa guiada por su conocimiento del dominio. El énfasis no está en el tamaño, sino en las capacidades comerciales. Además, si se necesita una cohesión clara para un área determinada de la aplicación en función de una gran cantidad de dependencias, eso también indica la necesidad de un solo microservicio. La cohesión es una forma de identificar cómo dividir o agrupar microservicios. En última instancia, mientras adquiere más conocimiento sobre el dominio, debe adaptar el tamaño de su microservicio, de manera iterativa. Encontrar el tamaño correcto no es un proceso de una sola vez.

[Sam Newman](#), un reconocido promotor de microservicios y autor del libro [Building Microservices](#), destaca que debe diseñar sus microservicios en función del patrón Bounded Context (BC) (parte del diseño basado en dominio), como se presentó anteriormente. En ocasiones, un BC puede estar compuesto por varios servicios físicos, pero no al revés.

Un modelo de dominio con entidades de dominio específicas se aplica dentro de un BC o microservicio concreto. Un BC delimita la aplicabilidad de un modelo de dominio y brinda a los miembros del equipo de desarrolladores una comprensión clara y compartida de lo que debe ser cohesivo y lo que se puede desarrollar de forma independiente. Estos son los mismos objetivos para los microservicios.

Otra herramienta que informa su elección de diseño es [la ley de Conway](#), que establece que una aplicación reflejará los límites sociales de la organización que la produjo. Pero a veces ocurre lo contrario: la organización de la empresa está formada por el software. Es posible que deba revertir la ley de Conway y construir los límites de la forma en que desea que se organice la empresa, inclinándose hacia la consultoría de procesos comerciales.

Para identificar contextos limitados, puede usar un patrón DDD llamado [patrón de asignación de contexto](#). Con Context Mapping, identifica los diversos contextos en la aplicación y sus límites. Es común tener un contexto y un límite diferentes para cada pequeño subsistema, por ejemplo. El Mapa de contexto es una forma de definir y hacer explícitos esos límites entre dominios. Un BC es autónomo e incluye los detalles de un solo dominio -detalles como las entidades del dominio- y define contratos de integración con otros BCs. 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 delimitado son buenos enfoques 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 el dominio del dominio del catálogo nombrará las entidades de manera diferente en los dominios del catálogo y el inventario que un experto en el dominio de envío. O la entidad del dominio del usuario puede ser diferente en tamaño y número de atributos cuando se trata de un experto en CRM que desea almacenar todos los detalles sobre el cliente que para un experto en el dominio de pedidos que solo necesita datos parciales sobre el cliente. Es muy difícil eliminar la ambigüedad de todos los términos de dominio en todos los dominios relacionados con una aplicación grande. Pero lo más importante es que no debes intentar unificar los términos. En su lugar, acepte las diferencias y la riqueza proporcionada por cada dominio. Si intenta tener una base de datos unificada para toda la aplicación, los intentos de un vocabulario unificado serán incómodos y no sonarán bien para ninguno de los múltiples expertos en dominios. Por lo tanto, los BC (implementados como microservicios) lo ayudarán a aclarar dónde puede usar ciertos términos de dominio y dónde deberá dividir el sistema y crear BC adicionales con diferentes dominios.

Sabrá que obtuvo los límites y tamaños correctos de cada BC y modelo de dominio si tiene pocas relaciones sólidas entre los modelos de dominio y, por lo general, no necesita combinar información de varios modelos de dominio al realizar operaciones de aplicación típicas.

Quizás la mejor respuesta a la pregunta de cuán grande debe ser un modelo de dominio para cada microservicio es la siguiente: debe tener un BC autónomo, lo más aislado posible, que le permita trabajar sin tener que cambiar constantemente a otros contextos (otros microservicios). modelos). En la Figura 4-10, puede ver cómo varios microservicios (múltiples BC) tienen cada uno su propio modelo y cómo sus

se pueden definir entidades, según los requisitos específicos para cada uno de los dominios identificados en su aplicación.

## Identifying a Domain Model per Microservice or Bounded Context

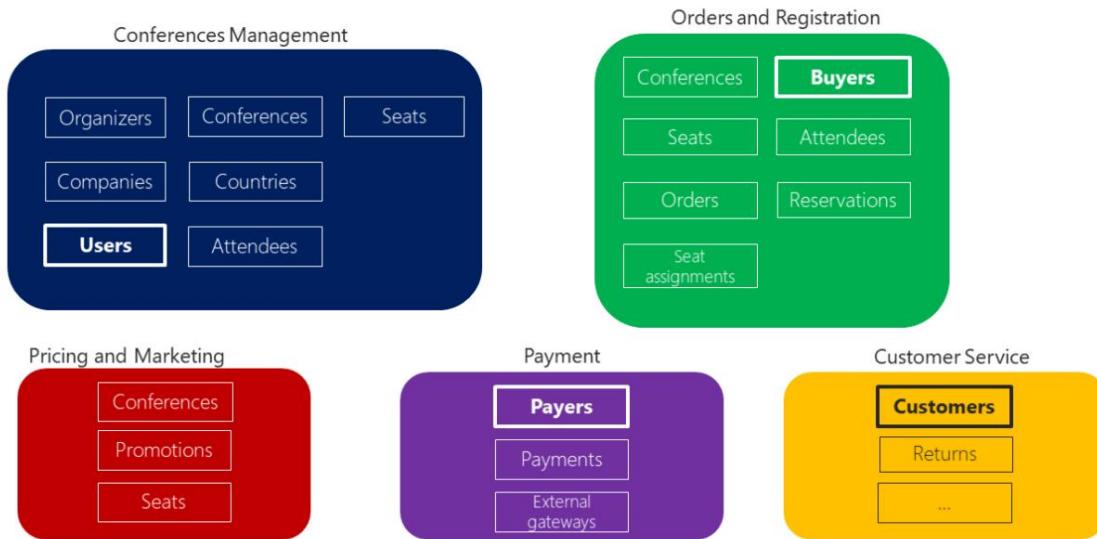


Figura 4-10. Identificación de entidades y límites del modelo de microservicio

La Figura 4-10 ilustra un escenario de ejemplo relacionado con un sistema de gestión de conferencias en línea. La misma entidad aparece como "Usuarios", "Compradores", "Pagadores" y "Clientes" según el contexto acotado. Ha identificado varios BC que podrían implementarse como microservicios, en función de los dominios que los expertos en dominios definieron para usted. Como puede ver, hay entidades que están presentes solo en un solo modelo de microservicio, como Pagos en el microservicio Pago. Esos serán fáciles de implementar.

Sin embargo, también puede tener entidades que tengan una forma diferente pero que comparten la misma identidad en los múltiples modelos de dominio de los múltiples microservicios. Por ejemplo, la entidad Usuario se identifica en el microservicio Gestión de conferencias. Ese mismo usuario, con la misma identidad, es el que se denomina Compradores en el microservicio Pedidos, o el que se denomina Pagador en el microservicio Pago, e incluso el que se denomina Cliente en el microservicio Atención al cliente. Esto se debe a que, dependiendo del lenguaje ubicuo que utilice cada experto en el dominio, un usuario puede tener una perspectiva diferente incluso con diferentes atributos. La entidad de usuario en el modelo de microservicio denominada Gestión de conferencias podría tener la mayoría de sus atributos de datos personales. Sin embargo, ese mismo usuario con forma de Pagador en el pago del microservicio o con forma de Cliente en el Servicio de atención al cliente del microservicio podría no necesitar la misma lista de atributos.

Un enfoque similar se ilustra en la Figura 4-11.

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

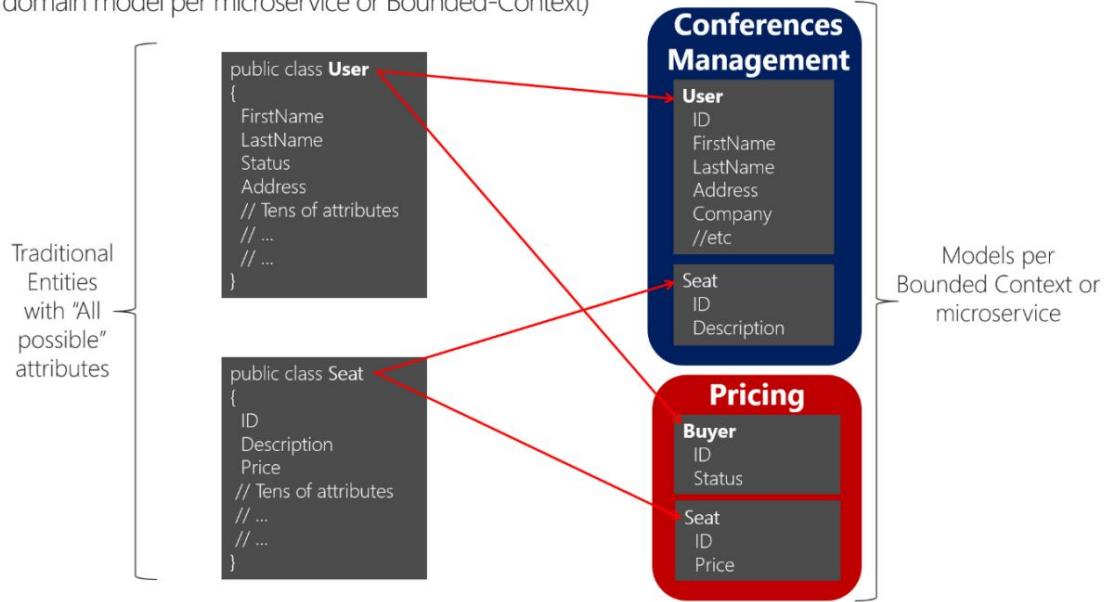


Figura 4-11. Descomposición de modelos de datos tradicionales en múltiples modelos de dominio

Al descomponer un modelo de datos tradicional entre contextos acotados, puede tener diferentes entidades que comparten la misma identidad (un comprador también es un usuario) con diferentes atributos en cada contexto acotado. Puede ver cómo el usuario está presente en el modelo de microservicio de Gestión de conferencias como la entidad Usuario y también está presente en forma de entidad Comprador en el microservicio Precios, con atributos alternativos o detalles sobre el usuario cuando en realidad es un comprador. Es posible que cada microservicio o BC no necesite todos los datos relacionados con una entidad de usuario, solo una parte, según el problema a resolver o el contexto. Por ejemplo, en el modelo de microservicio de precios, no necesita la dirección o el nombre del usuario, solo la identificación (como identidad) y el estado, lo que tendrá un impacto en los descuentos al fijar el precio de los asientos por comprador.

La entidad Asiento tiene el mismo nombre pero diferentes atributos en cada modelo de dominio. Sin embargo, Seat comparte identidad en base al mismo ID, al igual que ocurre con Usuario y Comprador.

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 puede haber detalles adicionales o diferentes sobre la entidad del usuario. Por lo tanto, debe haber una forma de asignar una entidad de usuario de un dominio (microservicio) a otro.

Hay varios beneficios de no compartir la misma entidad de usuario con la misma cantidad de atributos en todos los dominios. Un beneficio es reducir la duplicación, de modo que los modelos de microservicio no tengan datos que no necesiten. Otro beneficio es tener un microservicio principal que posee un cierto tipo de datos por entidad para que las actualizaciones y consultas de ese tipo de datos sean impulsadas solo por ese microservicio.

## El patrón de la puerta de enlace API frente a la comunicación directa entre el cliente y el microservicio

En una arquitectura de microservicios, cada microservicio expone un conjunto de puntos finales (normalmente) detallados.

Este hecho puede afectar la comunicación entre el cliente y el microservicio, como se explica en esta sección.

### Comunicación directa de cliente a microservicio

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

#### **Direct Client-To-Microservice communication**

Architecture

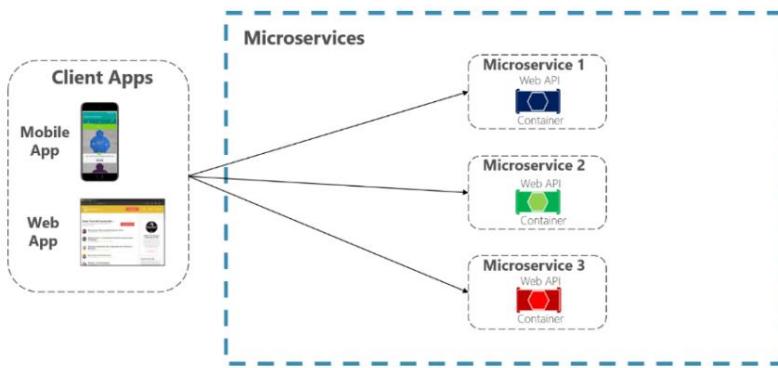


Figura 4-12. Uso de una arquitectura de comunicación directa de cliente a microservicio

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

<http://eshoponcontainers.westus.cloudapp.azure.com:88/>

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

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

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

- ¿Cómo pueden las aplicaciones cliente minimizar la cantidad de solicitudes al back-end y reducir la comunicación comunicativa a múltiples microservicios?

La interacción con varios microservicios para crear una única pantalla de interfaz de usuario aumenta la cantidad de viajes de ida y vuelta a través de Internet. Este enfoque aumenta la latencia y la complejidad en el lado de la interfaz de usuario. Idealmente, las respuestas deben agregarse de manera eficiente en el lado del servidor. Este enfoque reduce la latencia, ya que varias piezas de datos regresan en paralelo y algunas UI pueden mostrar datos tan pronto como estén listos.

- ¿Cómo puede manejar las preocupaciones transversales, como la autorización, las transformaciones de datos y envío de solicitudes dinámicas?

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

- ¿Cómo se pueden comunicar las aplicaciones de los clientes con los servicios que utilizan protocolos que no son aptos para Internet?

Los protocolos utilizados en el lado del servidor (como AMQP o protocolos binarios) no son compatibles con las aplicaciones cliente.

Por lo tanto, las solicitudes deben realizarse a través de protocolos como HTTP/HTTPS y luego traducirse a los demás protocolos. Un enfoque de hombre en el medio puede ayudar en esta situación.

- ¿Cómo se puede dar forma a una fachada especialmente diseñada para aplicaciones móviles?

Es posible que la API de múltiples microservicios no esté bien diseñada para 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, es posible que deba optimizar aún más para que las respuestas de datos puedan ser más eficientes. Puede realizar esta función agregando datos de varios microservicios y devolviendo un solo conjunto de datos y, a veces, eliminando cualquier dato en la respuesta que no necesite la aplicación móvil. Y, por supuesto, puede comprimir esos datos. Nuevamente, una fachada o API entre la aplicación móvil y los microservicios puede ser conveniente para este escenario.

## ¿Por qué considerar API Gateways en lugar de una comunicación directa entre el cliente y el microservicio?

En una arquitectura de microservicios, las aplicaciones cliente generalmente necesitan consumir funcionalidad de más de un microservicio. Si ese consumo se realiza directamente, el cliente necesita manejar múltiples llamadas a puntos finales de microservicio. ¿Qué sucede cuando la aplicación evoluciona y se introducen nuevos microservicios o se actualizan los microservicios existentes? Si su aplicación tiene muchos microservicios, manejar tantos puntos finales desde las aplicaciones cliente puede ser una pesadilla. Dado que la aplicación del cliente estaría acoplada a esos puntos finales internos, la evolución de los microservicios en el futuro puede causar un gran impacto en las aplicaciones del cliente.

Por lo tanto, tener un nivel intermedio o tier de indirección (Gateway) puede ser conveniente para aplicaciones basadas en microservicios.

Si no tiene API Gateways, las aplicaciones cliente deben enviar solicitudes directamente a los microservicios y eso genera problemas, como los siguientes:

- Acoplamiento: sin el patrón API Gateway, las aplicaciones cliente se acoplan a los microservicios internos. Las aplicaciones cliente necesitan saber cómo se descomponen las múltiples áreas de la aplicación en microservicios. Al evolucionar y refactorizar los microservicios internos, esos

las acciones afectan el mantenimiento porque provocan cambios importantes en las aplicaciones cliente debido a la referencia directa a los microservicios internos de las aplicaciones cliente. Las aplicaciones de los clientes deben actualizarse con frecuencia, lo que dificulta la evolución de la solución.

- Demasiados viajes de ida y vuelta: una sola página/pantalla en la aplicación del cliente puede requerir varias llamadas para múltiples servicios. Ese enfoque puede generar múltiples viajes de ida y vuelta en la red entre el cliente y el servidor, lo que agrega una latencia significativa. La agregación manejada en un nivel intermedio podría mejorar el rendimiento y la experiencia del usuario para la aplicación cliente.
- Problemas de seguridad: sin una puerta de enlace, todos los microservicios deben estar expuestos al "mundo externo", lo que hace que la superficie de ataque sea más grande que si oculta los microservicios internos que no utilizan directamente las aplicaciones cliente. Cuanto más pequeña sea la superficie de ataque, más segura puede ser su aplicación.
- Preocupaciones transversales: Cada microservicio publicado públicamente debe manejar preocupaciones como la autorización y SSL. En muchas situaciones, esas preocupaciones podrían manejarse en un solo nivel para simplificar los microservicios internos.

## ¿Qué es el patrón de API Gateway?

Cuando diseña y crea aplicaciones grandes o complejas basadas en microservicios con múltiples aplicaciones cliente, un buen enfoque a considerar puede ser una [API Gateway](#). Este patrón es un servicio que proporciona un punto de entrada único para ciertos grupos de microservicios. Es similar al [patrón Facade](#) del diseño orientado a objetos, pero en este caso, es parte de un sistema distribuido. El patrón API Gateway también se conoce a veces como "backend para frontend" (BFF) porque lo construye pensando en las necesidades de la aplicación cliente.

Por lo tanto, la puerta de enlace API se ubica entre las aplicaciones cliente y los microservicios. Actúa como un proxy inverso, enruteando las solicitudes de los clientes a los servicios. También puede proporcionar otras funciones transversales, como autenticación, terminación SSL y caché.

La Figura 4-13 muestra cómo una API Gateway personalizada puede encajar en una arquitectura simplificada basada en microservicios con solo unos pocos microservicios.

## Using a single custom API Gateway service

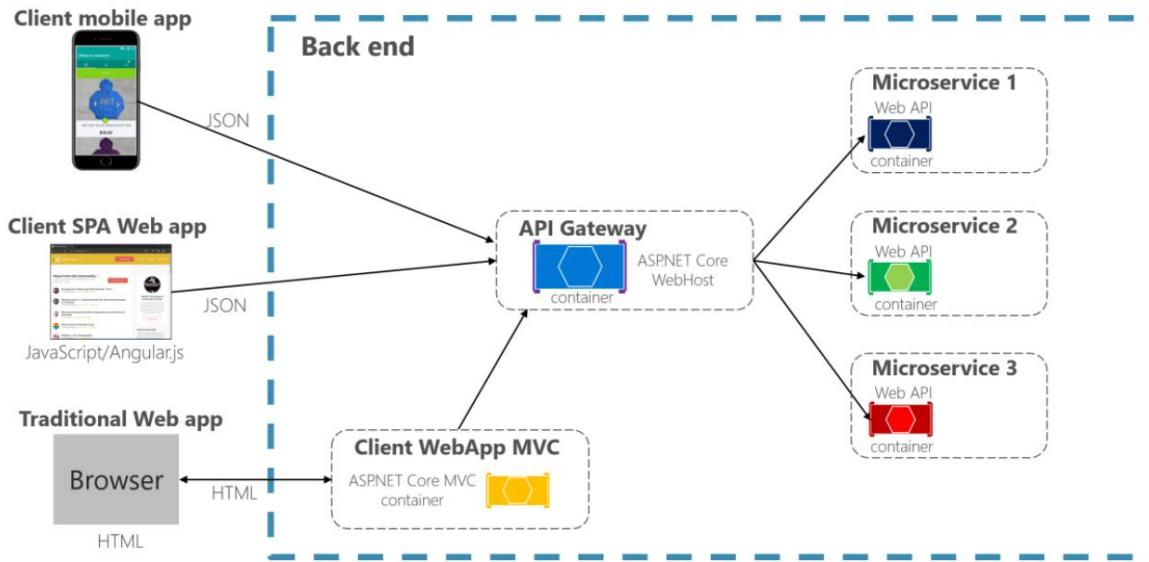


Figura 4-13. Uso de una API Gateway implementada como un servicio personalizado

Las aplicaciones se conectan a un único punto final, API Gateway, que está configurado para reenviar solicitudes a microservicios individuales. En este ejemplo, API Gateway se implementaría como un servicio ASP.NET Core WebHost personalizado que se ejecuta como un contenedor.

Es importante resaltar que en ese diagrama, estaría utilizando un único servicio API Gateway personalizado frente a múltiples y diferentes aplicaciones cliente. Ese hecho puede ser un riesgo importante porque su servicio API Gateway crecerá y evolucionará en función de muchos requisitos diferentes de las aplicaciones del cliente. Eventualmente, se inflará debido a esas diferentes necesidades y, de hecho, podría ser similar a una aplicación monolítica o un servicio monolítico. Es por eso que se recomienda encarecidamente dividir API Gateway en varios servicios o varias API Gateways más pequeñas, una por tipo de factor de forma de aplicación cliente, por ejemplo.

Debe tener cuidado al implementar el patrón API Gateway. Por lo general, no es una buena idea tener una puerta de enlace de API única que agregue todos los microservicios internos de su aplicación. Si lo hace, actúa como un agregador u orquestador monolítico y viola la autonomía de los microservicios al acoplar todos los microservicios.

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

Al dividir el nivel de puerta de enlace de API en varias puertas de enlace de API, si su aplicación tiene varias aplicaciones de cliente, eso puede ser un pivote principal al identificar los tipos de puertas de enlace de API múltiples, de modo que pueda tener una fachada diferente para las necesidades de cada aplicación de cliente. Este caso es un patrón llamado "Backend para frontend" (BFF) donde cada API Gateway puede proporcionar una API diferente adaptada para cada tipo de aplicación de cliente, posiblemente incluso en función del factor de forma del cliente mediante la implementación de un código de adaptador específico que debajo llama a múltiples microservicios internos. como se muestra en la siguiente imagen:

# Using multiple API Gateways / BFF

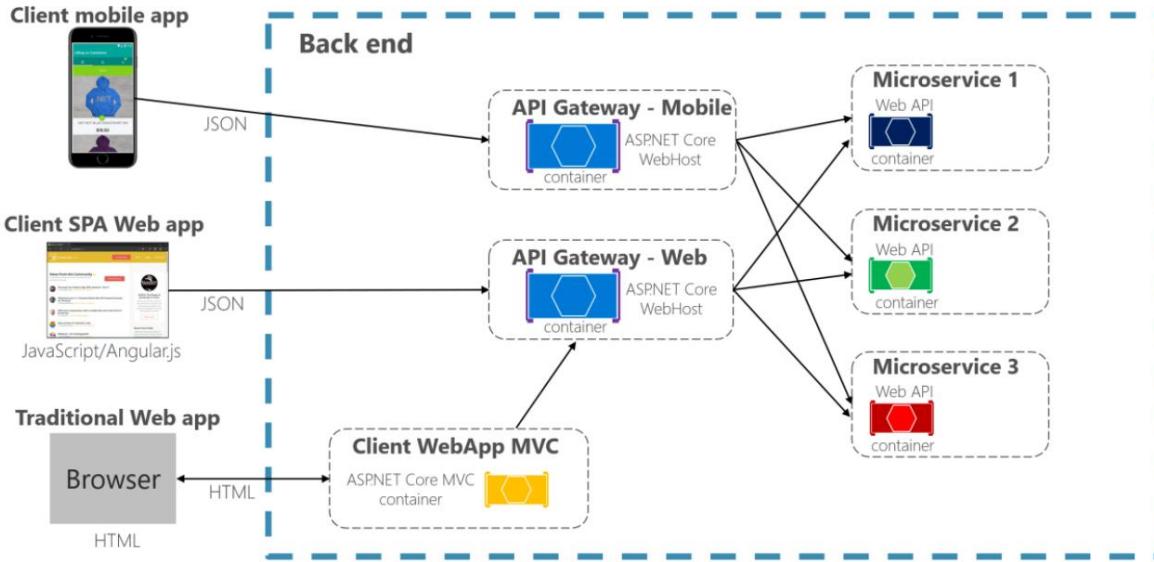


Figura 4-13.1. Uso de varias puertas de enlace de API personalizadas

La Figura 4-13.1 muestra API Gateways que están segregados por tipo de cliente; uno para clientes móviles y otro para clientes web. Una aplicación web tradicional se conecta a un microservicio MVC que utiliza la puerta de enlace API web. El ejemplo muestra una arquitectura simplificada con varias puertas de enlace API detalladas. En este caso, los límites identificados para cada puerta de enlace API se basan únicamente en el patrón "Backend para frontend" (BFF), por lo tanto, se basan solo en la API necesaria por aplicación cliente. Pero en aplicaciones más grandes, también debe ir más allá y crear otras puertas de enlace API basadas en los límites comerciales como un segundo pivote de diseño.

## Características principales en el patrón API Gateway

Una API Gateway puede ofrecer múltiples funciones. Según el producto, puede ofrecer funciones más completas o más sencillas; sin embargo, las funciones más importantes y fundamentales para cualquier API Gateway son los siguientes patrones de diseño:

Enrutamiento de puerta de enlace o proxy inverso. API Gateway ofrece un proxy inverso para redirigir o enrutar solicitudes (enrutamiento de capa 7, generalmente solicitudes HTTP) a los puntos finales de los microservicios internos. La puerta de enlace proporciona un único punto final o URL para las aplicaciones del cliente y luego 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 conveniente cuando se moderniza una API monolítica colocando API Gateway entre la API monolítica y las aplicaciones cliente, luego puede agregar nuevas API como nuevos microservicios sin dejar de usar el legado. API monolítica hasta que se divida en muchos microservicios en el futuro. Gracias a API Gateway, las aplicaciones cliente no notarán si las API que se utilizan 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 API Gateway, las aplicaciones cliente no se verá afectado por ningún cambio de URI.

Para obtener más información, consulte [Patrón de enrutamiento de puerta de enlace](#).

Agregación de solicitudes. Como parte del patrón de puerta de enlace, puede agregar varias solicitudes de clientes (generalmente solicitudes HTTP) dirigidas a múltiples microservicios internos en una sola solicitud de cliente. Este patrón es especialmente conveniente cuando la página/pantalla de un cliente necesita información de varios microservicios. Con este enfoque, la aplicación cliente envía una sola solicitud a API Gateway que envía varias solicitudes a los microservicios internos y luego agrega los resultados y envía todo de vuelta a la aplicación cliente. El principal beneficio y objetivo de este patrón de diseño es reducir la conversación 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 provenientes de aplicaciones SPA que provienen de JavaScript en los navegadores remotos del cliente. Para las aplicaciones web regulares 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 de cliente remoto.

Según el producto API Gateway que utilice, es posible que pueda realizar esta agregación.

Sin embargo, en muchos casos es más flexible crear microservicios de agregación bajo el alcance de API Gateway, por lo que define la agregación en el código (es decir, código C#):

Para obtener más información, consulte [Patrón de agregación de puerta de enlace](#).

Preocupaciones transversales o descarga de puerta de enlace. Según las características que ofrece cada producto API Gateway, puede descargar la funcionalidad de los microservicios individuales a la puerta de enlace, lo que simplifica la implementación de cada microservicio al consolidar las preocupaciones transversales en un solo nivel. Este enfoque es especialmente conveniente para funciones especializadas que pueden ser complejas de implementar correctamente en cada microservicio interno, como la siguiente funcionalidad:

- Autenticacion y autorizacion
- Integración de descubrimiento de servicios
- Almacenamiento en caché de respuestas
- Políticas de reintento, disyuntor y QoS
- Limitación de velocidad y estrangulamiento
- Balanceo de carga
- Registro, rastreo, correlación
- Encabezados, cadenas de consulta y transformación de notificaciones
- lista de IP permitida

Para obtener más información, consulte [Patrón de descarga de puerta de enlace](#).

## Uso de productos con características de API Gateway

Puede haber muchas más preocupaciones transversales ofrecidas por los productos API Gateways dependiendo de cada implementación.

Exploraremos aquí:

- [Administración de API de Azure](#)
- [Ocelote](#)

## Administración de API de Azure

[Azure API Management](#) (como se muestra en la Figura 4-14) no solo resuelve sus necesidades de API Gateway, sino que también proporciona características como la recopilación de información de sus API. Si está utilizando 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.

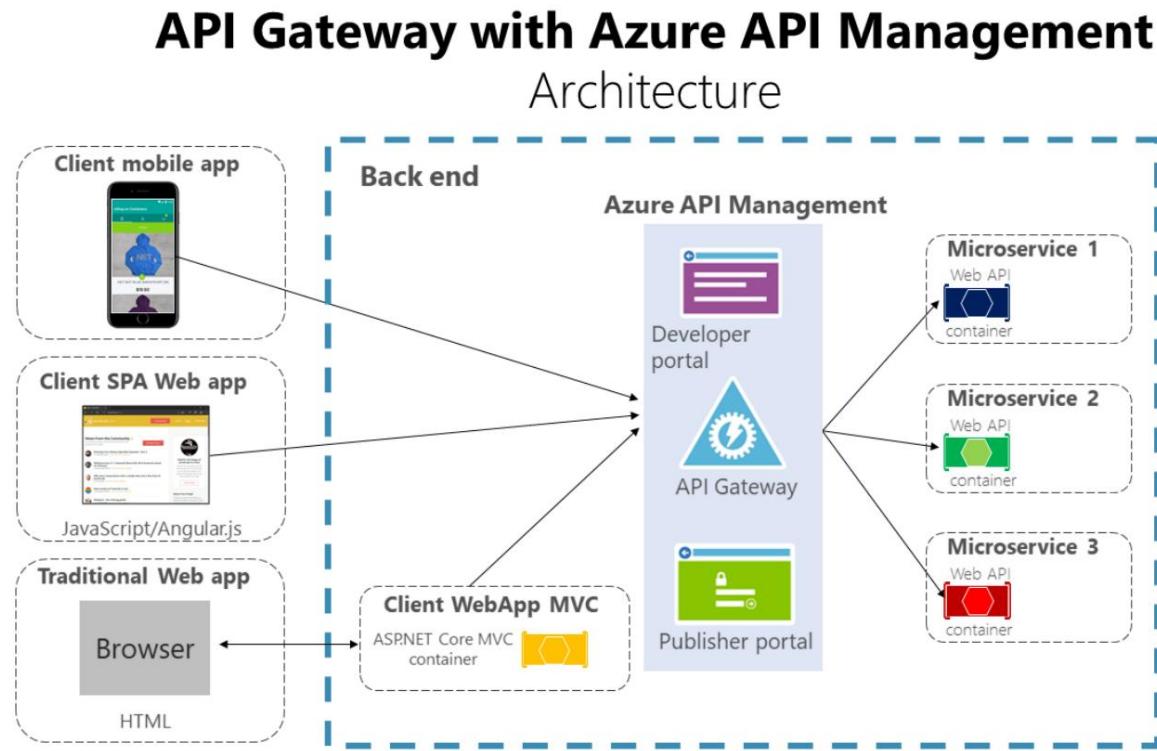


Figura 4-14. Uso de Azure API Management para API Gateway

Azure API Management resuelve sus necesidades de API Gateway y administración, como registro, seguridad, medición, etc. En este caso, al usar un producto como Azure API Management, el hecho de que pueda tener una API Gateway única no es tan riesgoso porque este tipo de API Gateways son "más delgados", lo que significa que no implementa código C# personalizado que podría evolucionar hacia un componente monolítico.

Los productos API Gateway suelen actuar como un proxy inverso para la comunicación de entrada, donde también puede filtrar las API de los microservicios internos y aplicar la autorización a las API publicadas en este único nivel.

Los conocimientos disponibles de un sistema de administración de API lo ayudan a comprender cómo se utilizan sus API y cómo se están desempeñando. Realizan esta actividad al permitirle ver informes analíticos casi en tiempo real e identificar tendencias que podrían afectar su negocio. Además, puede tener registros sobre la actividad de solicitudes y respuestas para un análisis adicional en línea y fuera de línea.

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

En esta guía y en la aplicación de muestra de referencia (eShopOnContainers), la arquitectura se limita a una arquitectura en contenedores más simple y personalizada para centrarse en contenedores simples sin

usando productos PaaS como Azure API Management. Pero para aplicaciones grandes basadas en microservicios que se implementan en Microsoft Azure, le recomendamos que evalúe Azure API Management como base para sus puertas de enlace de API en producción.

## Ocelote

[Ocelot](#) es una API Gateway liviana, recomendada para enfoques más simples. Ocelot es una puerta de enlace de API basada en .NET Core de código abierto especialmente diseñada para arquitecturas de microservicios que necesitan puntos de entrada unificados en sus sistemas. Es liviano, rápido y escalable y proporciona enruteamiento y autenticación, entre muchas otras características.

La razón principal para elegir Ocelot para la [aplicación de referencia 2.0 de eShopOnContainers](#) es que Ocelot es una puerta de enlace de API liviana de .NET Core que puede implementar en el mismo entorno de implementación de aplicaciones donde está implementando sus microservicios/contenedores, como Docker Host, Kubernetes , etc. Y dado que se basa en .NET Core, es multiplataforma, lo que le permite implementar en Linux o Windows.

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

Además, hay muchos otros productos en el mercado que ofrecen funciones de API Gateways, como Apigee, Kong, MuleSoft, WSO2 y otros productos como Linkerd e Istio para funciones de controlador de ingreso de malla de servicios.

Después de las secciones iniciales de explicación de patrones y arquitectura, las siguientes secciones explican cómo implementar API Gateways con [Ocelot](#).

## Inconvenientes del patrón API Gateway

- El inconveniente más importante es que cuando implementa una API Gateway, está acoplando ese nivel con los microservicios internos. Un acoplamiento como este podría presentar serias dificultades para su aplicación. Clemens Vaster, arquitecto del equipo de Azure Service Bus, se refiere a esta dificultad potencial como "el nuevo ESB" en la sesión "[Mensajería y microservicios](#)" en GOTO 2016.
- El uso de una puerta de enlace API de microservicios crea un posible punto único adicional de falla.
- Una API Gateway puede introducir un mayor tiempo de respuesta debido a la llamada de red adicional. Sin embargo, esta llamada adicional generalmente tiene menos impacto que tener una interfaz de cliente demasiado habladora que llama directamente a los microservicios internos.
- Si no se escala adecuadamente, API Gateway puede convertirse en un cuello de botella.
- Una API Gateway requiere un costo de desarrollo adicional y mantenimiento futuro si incluye lógica personalizada y agregación de datos. Los desarrolladores deben actualizar API Gateway para exponer los puntos finales de cada microservicio. Además, los cambios de implementación en los microservicios internos pueden provocar cambios de código en el nivel de API Gateway. Sin embargo, si API Gateway solo aplica seguridad, registro y control de versiones (como cuando se usa Azure API Management), es posible que este costo de desarrollo adicional no se aplique.

- Si API Gateway es desarrollado por un solo equipo, puede haber un cuello de botella en el desarrollo. Este aspecto es otra razón por la que un mejor enfoque es tener varias puertas de enlace API detalladas que respondan a las diferentes necesidades de los clientes. También puede segregar API Gateway internamente en varias áreas o capas que pertenecen a los diferentes equipos que trabajan en los microservicios internos.

## Recursos adicionales

- Chris Richardson. Patrón: API Gateway / Backend para Front-End <https://microservices.io/patterns/apigateway.html>
- Patrón de API Gateway <https://docs.microsoft.com/azure/architecture/microservices/gateway>
- Patrón de agregación y composición <https://microservices.io/patterns/data/api-composition.html>
- Administración de API de Azure  
<https://azure.microsoft.com/services/api-management/>
- Udi Dahan. Composición orientada a servicios  
<https://udidahan.com/2014/07/30/service-oriented-composition-with-video/>
- Clemens Vasters. Mensajería y Microservicios en GOTO 2016 (video) <https://www.youtube.com/watch?v=rXi5CLjIQ9k>
- API Gateway en pocas palabras (serie de tutoriales de ASP.NET Core API Gateway)  
<https://www.pogsdotnet.com/2018/08/api-gateway-in-nutshell.html>

## Comunicación en una arquitectura de microservicio

En una aplicación monolítica que se ejecuta en un solo proceso, los componentes se invocan entre sí mediante llamadas a funciones o métodos de nivel de idioma. Estos se pueden acoplar fuertemente si está creando objetos con código (por ejemplo, el nuevo `ClassName()`), o se pueden invocar de forma desacoplada si está usando Inyección de dependencia al hacer referencia a abstracciones en lugar de instancias de objetos concretos. De cualquier manera, los objetos se ejecutan dentro del mismo proceso. El mayor desafío al cambiar de una aplicación monolítica a una aplicación basada en microservicios radica en cambiar el mecanismo de comunicación. Una conversión directa de llamadas de método en proceso a llamadas RPC a servicios provocará una comunicación comunicativa y no eficiente que no funcionará bien en entornos distribuidos. Los desafíos de diseñar correctamente un sistema distribuido son lo suficientemente conocidos como para que exista un canon conocido como las [Falacias de la computación distribuida](#) que enumera las suposiciones que los desarrolladores suelen hacer cuando pasan de diseños monolíticos a diseños distribuidos.

No hay una solución, sino varias. Una solución consiste en aislar los microservicios empresariales tanto como sea posible. Luego, usa la comunicación asíncrona entre los microservicios internos y reemplaza la comunicación detallada que es típica en la comunicación dentro del proceso entre objetos con una comunicación más detallada. Puede hacerlo agrupando llamadas y devolviendo datos que agregan los resultados de varias llamadas internas al cliente.

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

La comunidad de microservicios promueve la filosofía de "[puntos finales inteligentes y conductos tontos](#)". Este eslogan fomenta un diseño lo más desacoplado posible entre microservicios y lo más cohesivo posible dentro de un solo microservicio. Como se explicó anteriormente, cada microservicio posee sus propios datos y su propia lógica de dominio. Pero los microservicios que componen una aplicación de un extremo a otro generalmente se coreografian simplemente mediante el uso de comunicaciones REST en lugar de protocolos complejos como WS-\* y comunicaciones flexibles basadas en eventos en lugar de orquestadores de procesos comerciales centralizados.

Los dos protocolos de uso común son la solicitud/respuesta HTTP con API de recursos (sobre todo cuando se realizan consultas) y la mensajería asíncrona ligera cuando se comunican actualizaciones a través de múltiples microservicios. Estos se explican con más detalle en las siguientes secciones.

## Tipos de comunicación

El cliente y los servicios pueden comunicarse a través de muchos tipos diferentes de comunicación, cada uno dirigido a un escenario y objetivos diferentes. Inicialmente, ese tipo de comunicaciones se pueden clasificar en dos ejes

El primer eje define si el protocolo es síncrono o asíncrono:

- Protocolo síncrono. HTTP es un protocolo síncrono. El cliente envía una solicitud y espera una respuesta del servicio. Eso es independiente de la ejecución del código del cliente que podría ser síncrona (el subproceso está bloqueado) o asíncrona (el subproceso no está bloqueado y la respuesta eventualmente llegará a una devolución de llamada). El punto importante aquí es que el protocolo (HTTP/HTTPS) es sincrónico y el código del cliente solo puede continuar su tarea cuando recibe el servidor HTTP.
- Protocolo asíncrono. Otros protocolos como AMQP (un protocolo compatible con muchos sistemas operativos y entornos de nube) utilizan mensajes asíncronos. El código de cliente o el remitente del mensaje generalmente no espera una respuesta. Simplemente envía el mensaje como cuando envía un mensaje a una cola de RabbitMQ o cualquier otro intermediario de mensajes.

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

- Receptor único. Cada solicitud debe ser procesada por exactamente un receptor o servicio. Un ejemplo de esta comunicación es el [patrón Command](#).
- Múltiples receptores. Cada solicitud puede ser procesada por cero a múltiples receptores. Este tipo de comunicación debe ser asíncrona. Un ejemplo es el mecanismo de [publicación/suscripción](#) que se utiliza en patrones como [la arquitectura basada en eventos](#). Esto se basa en una interfaz de bus de eventos o un intermediario de mensajes al propagar actualizaciones de datos entre múltiples microservicios a través de eventos; por lo general, se implementa mediante un bus de servicio o un artefacto similar, como [Azure Service Bus](#), mediante [temas y suscripciones](#).

Una aplicación basada en microservicios a menudo utilizará una combinación de estos estilos de comunicación. El tipo más común es la comunicación de un solo receptor con un protocolo síncrono como HTTP/HTTPS cuando se invoca un servicio HTTP de Web API regular. Los microservicios también suelen utilizar protocolos de mensajería para la comunicación asíncrona entre microservicios.

Es bueno conocer estos ejes para tener claridad sobre los posibles mecanismos de comunicación, pero no son las preocupaciones importantes al crear microservicios. Ni la naturaleza asíncrona de la ejecución del subproceso del cliente ni la naturaleza asíncrona del protocolo seleccionado son los puntos importantes al integrar microservicios. Lo importante es poder integrar sus microservicios de forma asíncrona manteniendo la independencia de los microservicios, como se explica en la siguiente sección.

## La integración asíncrona de microservicios refuerza la autonomía de los microservicios

Como se mencionó, el punto importante al crear una aplicación basada en microservicios es la forma en que integra sus microservicios. Idealmente, debería intentar minimizar la comunicación entre los microservicios internos. Cuantas menos comunicaciones entre microservicios, mejor. Pero en muchos casos, tendrá que integrar de alguna manera los microservicios. Cuando necesite hacer eso, la regla crítica aquí es que la comunicación entre los microservicios debe ser asíncrona. Eso no significa que tenga que usar un protocolo específico (por ejemplo, mensajería asíncrona versus HTTP sincrónico). Simplemente significa que la comunicación entre los microservicios debe realizarse solo mediante la propagación de datos de forma asíncrona, pero trate de no depender de otros microservicios internos como parte de la operación de solicitud/respuesta HTTP del servicio inicial.

Si es posible, nunca dependa de la comunicación síncrona (solicitud/respuesta) entre múltiples microservicios, ni siquiera para consultas. El objetivo de cada microservicio es ser autónomo y estar disponible para el consumidor del cliente, incluso si los demás servicios que forman parte de la aplicación de extremo a extremo están inactivos o en mal estado. Si cree que necesita realizar una llamada de un microservicio a otros microservicios (como realizar una solicitud HTTP para una consulta de datos) para poder proporcionar una respuesta a una aplicación cliente, tiene una arquitectura que no será resistente cuando algunos los microservicios fallan.

Además, tener dependencias HTTP entre microservicios, como cuando se crean 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 sus microservicios no sean autónomos, sino que también su rendimiento se ve afectado tan pronto como sea posible. uno de los servicios en esa cadena no está funcionando bien.

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

## Synchronous vs. async communication across microservices

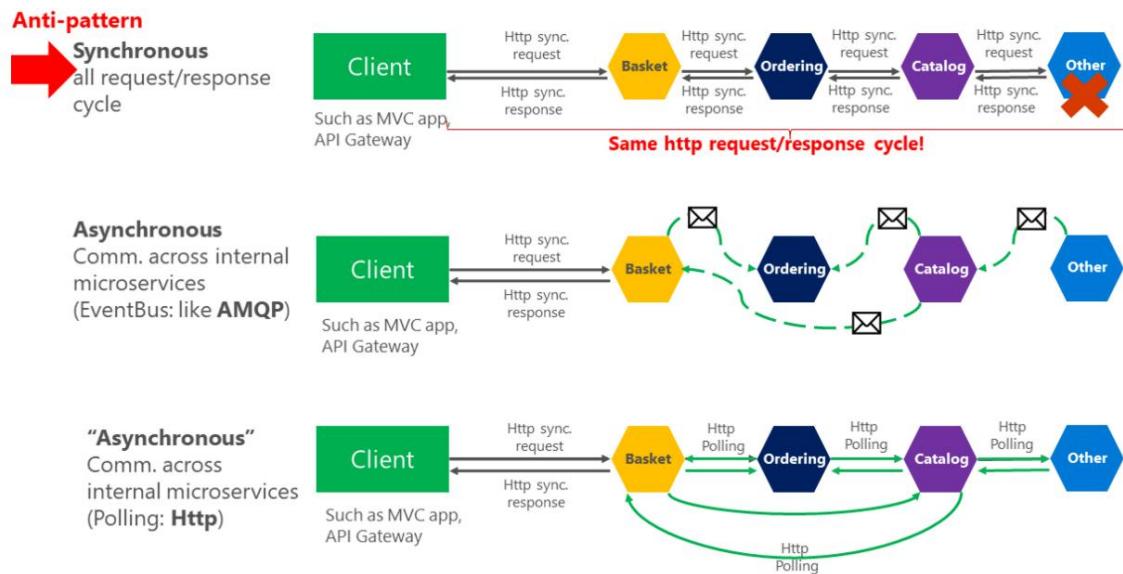


Figura 4-15. Anti-patrones y patrones en la comunicación entre microservicios

Como se muestra en el diagrama anterior, en la comunicación síncrona se crea una "cadena" de solicitudes entre microservicios mientras se atiende la solicitud del cliente. Este es un anti-patrón. En la comunicación asíncrona, los microservicios usan mensajes asíncronos o sondeos http para comunicarse con otros microservicios, pero la solicitud del cliente se atiende de inmediato.

Si su microservicio necesita generar una acción adicional en otro microservicio, si es posible, no realice esa acción sincrónicamente y como parte de la operación original de solicitud y respuesta del microservicio. En su lugar, hágalo de forma asíncrona (usando mensajes asíncronos o eventos de integración, colas, etc.). Pero, en la medida de lo posible, no invoque la acción de forma síncrona como parte de la operación de solicitud y respuesta síncrona original.

Y finalmente (y aquí es donde surgen la mayoría de los problemas al crear microservicios), si su microservicio inicial necesita datos que originalmente pertenecen a otros microservicios, no confie en realizar solicitudes síncronas de esos datos. En su lugar, replique o propague esos datos (solo los atributos que necesita) en la base de datos del servicio inicial mediante el uso de coherencia eventual (normalmente mediante el uso de eventos de integración, como se explica en las próximas secciones).

Como se señaló anteriormente en la sección [Identificar los límites del modelo de dominio para cada microservicio](#), duplicar algunos datos en varios microservicios no es un diseño incorrecto; al contrario, al hacerlo, puede traducir los datos al idioma específico o a los términos de ese dominio adicional. o Contexto acotado. Por ejemplo, en la aplicación [eShopOnContainers](#) tiene un microservicio llamado Identity-Api que está a cargo de la mayoría de los datos del usuario con una entidad llamada Usuario. Sin embargo, cuando necesita almacenar datos sobre el usuario dentro del microservicio Pedidos , los almacena como una entidad diferente denominada Comprador. La entidad Compradora comparte la misma identidad con la entidad Usuario original , pero es posible que solo tenga los pocos atributos que necesita el dominio de pedidos , y no el perfil de usuario completo.

Puede usar cualquier protocolo para comunicar y propagar datos de forma asíncrona entre microservicios para tener coherencia final. Como se mencionó, podría usar eventos de integración usando un bus de eventos o un intermediario de mensajes o incluso podría usar HTTP sondeando los otros servicios en su lugar. No importa. La regla importante es no crear dependencias síncronas entre sus microservicios.

Las siguientes secciones explican los múltiples estilos de comunicación que puede considerar usar en una aplicación basada en microservicios.

## Estilos de comunicación

Hay muchos protocolos y opciones que puede usar para la comunicación, según el tipo de comunicación que desee usar. Si está utilizando un mecanismo de comunicación síncrono basado en solicitud/respuesta, los protocolos como los enfoques HTTP y REST son los más comunes, especialmente si está publicando sus servicios fuera del host de Docker o del clúster de microservicios. Si se está comunicando entre servicios internamente (dentro de su host de Docker o clúster de microservicios), es posible que también quiera usar mecanismos de comunicación de formato binario (como WCF usando TCP y formato binario). Como alternativa, puede utilizar mecanismos de comunicación asíncronos basados en mensajes, como AMQP.

También existen múltiples formatos de mensajes como JSON o XML, o incluso formatos binarios, que pueden ser más eficientes. Si su formato binario elegido no es un estándar, probablemente no sea una buena idea publicar públicamente sus servicios usando ese formato. Podría usar un formato no estándar para la comunicación interna entre sus microservicios. Puede hacer esto cuando se comunica entre microservicios dentro de su host de Docker o clúster de microservicios (por ejemplo, orquestadores de Docker) o para aplicaciones de cliente patentadas que se comunican con los microservicios.

## Comunicación de solicitud/respuesta con HTTP y REST

Cuando un cliente utiliza la comunicación de solicitud/respuesta, envía una solicitud a un servicio, luego el servicio procesa la solicitud y devuelve una respuesta. La comunicación de solicitud/respuesta es especialmente adecuada para consultar datos para una interfaz de usuario en tiempo real (una interfaz de usuario en vivo) desde aplicaciones cliente. Por lo tanto, en una arquitectura de microservicio probablemente usará 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

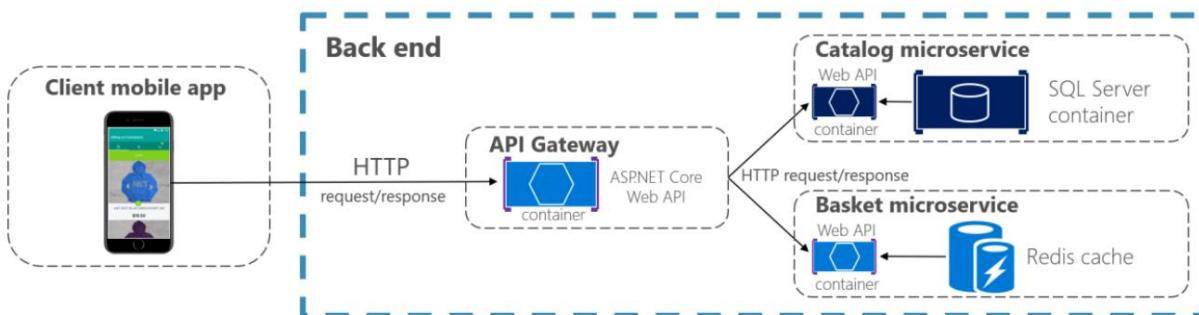


Figura 4-16. Uso de comunicación de solicitud/respuesta HTTP (sincrónica o asíncrona)

Cuando un cliente utiliza la comunicación de solicitud/respuesta, asume que la respuesta llegará en poco tiempo, generalmente menos de un segundo o unos pocos segundos como máximo. Para las respuestas retrasadas, debe implementar la comunicación asincrónica basada en [patrones de mensajería y tecnologías de mensajería, que es un enfoque](#) diferente que explicamos en la siguiente sección.

Un estilo arquitectónico popular para la comunicación de solicitud/respuesta es [REST](#). Este enfoque se basa en el protocolo [HTTP](#) y está estrechamente relacionado con él, y abarca verbos HTTP como GET, POST y PUT. REST es el enfoque de comunicación arquitectónica más utilizado al crear servicios. Puede implementar servicios REST cuando desarrolla servicios de API web de ASP.NET Core.

Hay un valor adicional cuando se utilizan servicios HTTP REST como su lenguaje de definición de interfaz. Por ejemplo, si usa [metadatos de Swagger](#) para describir la API de su servicio, puede usar herramientas que generan stubs de clientes que pueden descubrir y consumir directamente sus servicios.

### Recursos adicionales

- Martín Fowler. Modelo de madurez de Richardson Una descripción del modelo REST. <https://martinfowler.com/articles/richardsonMaturityModel.html>
- Swagger El sitio oficial. <https://swagger.io/>

### Comunicación push y en tiempo real basada en HTTP

Otra posibilidad (generalmente para propósitos diferentes a REST) es una comunicación en tiempo real y de uno a muchos con marcos de trabajo de nivel superior como [ASP.NET SignalR](#) y [protocolos](#) como [WebSockets](#).

---

Como muestra la Figura 4-17, la comunicación HTTP en tiempo real significa que puede hacer que el código del servidor envíe contenido a los clientes conectados a medida que los datos estén disponibles, en lugar de que el servidor espere a que un cliente solicite nuevos datos.

## Push and real-time communication based on HTTP

### One-to-many communication

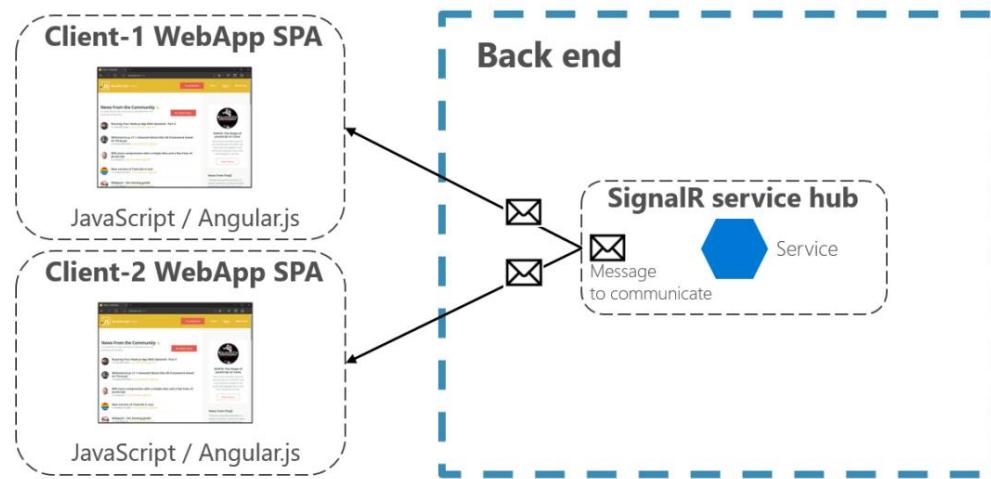


Figura 4-17. Comunicación de mensajes asíncronos en tiempo real de uno a muchos

SignalR es una buena manera de lograr una comunicación en tiempo real para enviar contenido a los clientes desde un servidor back-end. Dado que la comunicación es en tiempo real, las aplicaciones de los clientes muestran los cambios casi al instante. Esto generalmente lo maneja un protocolo como WebSockets, utilizando muchas conexiones WebSockets (una por cliente). Un ejemplo típico es cuando un servicio comunica un cambio en la puntuación de un juego deportivo a muchas aplicaciones web de clientes simultáneamente.

## Comunicación asíncrona basada en mensajes

La mensajería asíncrona y la comunicación basada en eventos son fundamentales cuando se propagan cambios en varios microservicios y sus modelos de dominio relacionados. Como se mencionó anteriormente en la discusión, los microservicios y los Bounded Contexts (BC), los modelos (Usuario, Cliente, Producto, Cuenta, etc.) pueden significar diferentes cosas para diferentes microservicios o BC. Eso significa que cuando ocurren cambios, necesita alguna forma de conciliar los cambios en los diferentes modelos. Una solución es la consistencia eventual y la comunicación basada en eventos basada en mensajería asíncrona.

Cuando se utiliza la mensajería, los procesos se comunican intercambiando mensajes de forma asíncrona. Un cliente realiza un comando o una solicitud a un servicio enviándole un mensaje. Si el servicio necesita responder, envía un mensaje diferente al cliente. Dado que se trata de una comunicación basada en mensajes, el cliente asume que la respuesta no se recibirá de inmediato 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. Los mensajes generalmente se envían a través de protocolos asíncronos como AMQP.

La infraestructura preferida para este tipo de comunicación en la comunidad de microservicios es un intermediario de mensajes ligero, que es diferente de los grandes intermediarios y orquestadores que se utilizan en SOA.

En un intermediario de mensajes liviano, la infraestructura suele ser "tonta", actuando solo como un intermediario de mensajes, con implementaciones simples como RabbitMQ o un bus de servicio escalable en la nube como

Autobús de servicio de Azure. En este escenario, la mayor parte del pensamiento "inteligente" todavía vive en los puntos finales que producen y consumen mensajes, es decir, en los microservicios.

Otra regla que debe tratar de seguir, en la medida de lo posible, es usar solo mensajería asíncrona entre los servicios internos y usar comunicación síncrona (como HTTP) solo desde las aplicaciones cliente a los servicios front-end (API Gateways más el primer nivel de microservicios).

Hay dos tipos de comunicación de mensajería asíncrona: comunicación basada en mensajes de un solo receptor y comunicación basada en mensajes de múltiples receptores. En las siguientes secciones se proporcionan detalles sobre ellos.

## Comunicación basada en mensajes de un solo receptor

La comunicación asíncrona basada en mensajes con un solo receptor significa que hay una comunicación punto a punto que entrega un mensaje exactamente a uno de los consumidores que está leyendo del canal, y que el mensaje se procesa solo una vez. Sin embargo, hay situaciones especiales. Por ejemplo, en un sistema en la nube que intenta recuperarse automáticamente de fallas, el mismo mensaje podría enviarse varias veces. Debido a fallas en la red u otras fallas, el cliente debe poder volver a intentar enviar mensajes, y el servidor debe implementar una operación para ser idempotente a fin de procesar un mensaje en particular solo una vez.

La comunicación basada en mensajes de un solo receptor es especialmente adecuada para enviar comandos asíncronos de un microservicio a otro, como se muestra en la Figura 4-18 que ilustra este enfoque.

Una vez que comience a enviar comunicación basada en mensajes (ya sea con comandos o eventos), debe evitar mezclar la comunicación basada en mensajes con la comunicación HTTP síncrona.

## Single receiver message-based communication (i.e. Message-based Commands)

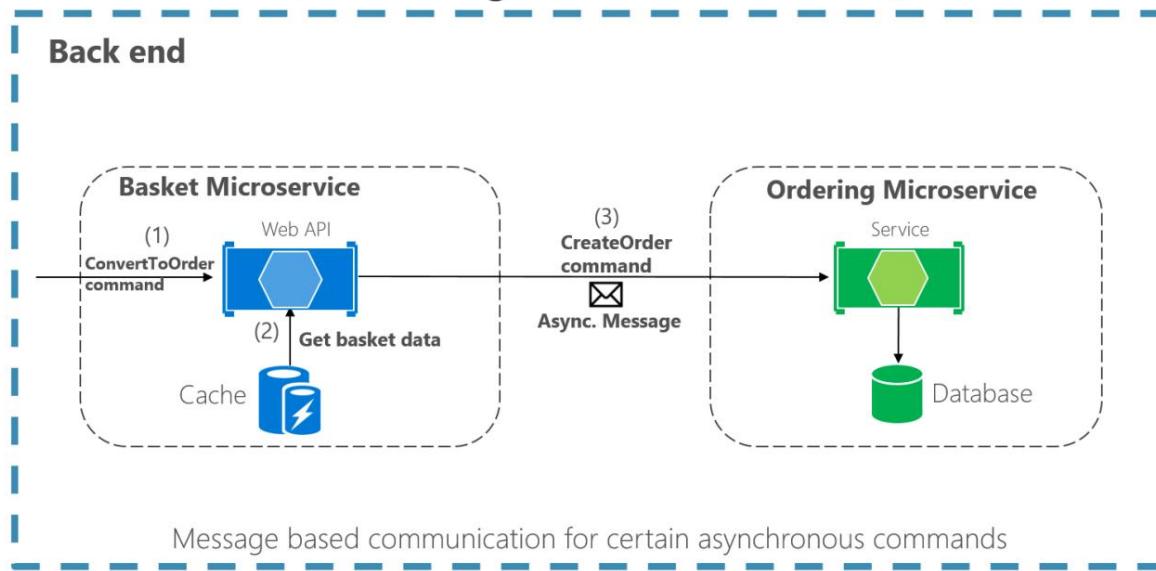


Figura 4-18. Un único microservicio que recibe un mensaje asíncrono

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

## Comunicación basada en mensajes de múltiples receptores

Como un enfoque más flexible, es posible que desee utilizar un mecanismo de publicación/suscripción para que su comunicación del remitente esté disponible para microservicios de suscriptor adicionales o para aplicaciones externas. Por lo tanto, le ayuda a seguir el [principio abierto/cerrado](#) en el servicio de envío. De esa manera, se pueden agregar suscriptores adicionales en el futuro sin necesidad de modificar el servicio del remitente.

Cuando utiliza una comunicación de publicación/suscripción, puede estar utilizando una interfaz de bus de eventos para publicar eventos para cualquier suscriptor.

## Comunicación asincrónica basada en eventos

Cuando se utiliza la comunicación asincrónica basada en eventos, un microservicio publica un evento de integración cuando sucede algo dentro de su dominio y otro microservicio debe estar al tanto, como un cambio de precio en un microservicio de catálogo de productos. Los microservicios adicionales se suscriben a los eventos para que puedan recibirlas de forma asíncrona. Cuando eso sucede, los receptores pueden actualizar sus propias entidades de dominio, lo que puede hacer que se publiquen más eventos de integración. Este sistema de publicación/suscripción se realiza utilizando 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 eventos y publicar eventos. El bus de eventos también puede tener una o más implementaciones basadas en cualquier agente de mensajería y entre procesos, como una cola de mensajería o un bus de servicio que admite la comunicación asíncrona y un modelo de publicación/suscripción.

Si un sistema utiliza una consistencia eventual impulsada por eventos de integración, se recomienda que este enfoque quede claro para el usuario final. El sistema no debe usar un enfoque que imite los eventos de integración, como SignalR o los sistemas de sondeo del cliente. El usuario final y el propietario de la empresa deben adoptar explícitamente la consistencia eventual en el sistema y darse cuenta de que, en muchos casos, la empresa no tiene ningún problema con este enfoque, siempre que sea explícito. Este enfoque es importante porque los usuarios pueden esperar ver algunos resultados inmediatamente y este aspecto podría no ocurrir con la consistencia eventual.

Como se señaló anteriormente en la sección [Desafíos y soluciones para la administración de datos distribuidos](#), puede usar eventos de integración para implementar tareas comerciales que abarquen varios microservicios. Por lo tanto, tendrá coherencia final entre esos servicios. Una transacción eventualmente consistente 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 integral.

Un punto importante es que es posible que desee comunicarse con varios microservicios que están suscritos al mismo evento. Para hacerlo, puede usar mensajes de publicación/suscripción basados en comunicación impulsada por eventos, como se muestra en la Figura 4-19. Este mecanismo de publicación/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 propaga las actualizaciones desde la base de datos de escritura a la base de datos de lectura en el comando

y patrón de arquitectura de segregación de responsabilidad de consulta (CQRS). El objetivo es tener coherencia final entre múltiples fuentes de datos en su sistema distribuido.

## Asynchronous event-driven communication

### Multiple receivers

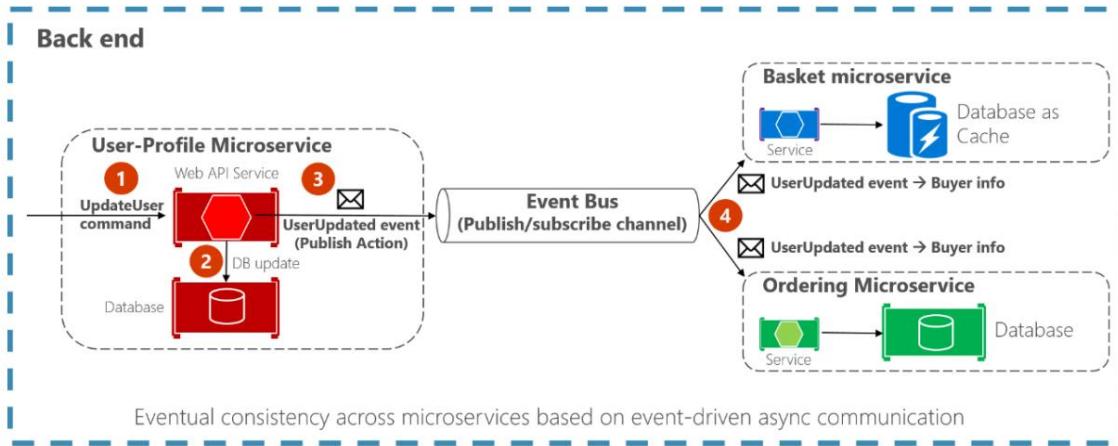


Figura 4-19. Comunicación asincrónica de mensajes basada en eventos

En la comunicación asincrónica basada en eventos, un microservicio publica eventos en un bus de eventos y muchos microservicios pueden suscribirse para recibir notificaciones y actuar en consecuencia. Su implementación determinará qué protocolo usar para las comunicaciones basadas en mensajes y basadas en eventos. [AMQP](#) puede ayudar a lograr una comunicación en cola confiable.

Cuando usa un bus de eventos, es posible que desee usar un nivel de abstracción (como una interfaz de bus de eventos) basado en una implementación relacionada en clases con código que usa la API de un agente de mensajes como [RabbitMQ](#) o un bus de servicio como [Azure Service Bus con temas](#). Como alternativa, es posible que desee utilizar un bus de servicio de nivel superior como NServiceBus, MassTransit o Brighter para articular su bus de eventos y el sistema de publicación/suscripción.

### Una nota sobre tecnologías de mensajería para sistemas de producción

Las tecnologías de mensajería disponibles para implementar su bus de eventos abstractos se encuentran en diferentes niveles. Por ejemplo, productos como RabbitMQ (un agente de transporte de mensajería) y Azure Service Bus se ubican en un nivel más bajo que otros productos como NServiceBus, MassTransit o Brighter, que pueden funcionar sobre RabbitMQ y Azure Service Bus. Su elección depende de la cantidad de características enriquecidas a nivel de aplicación y la escalabilidad lista para usar que necesita para su aplicación. Para implementar solo un bus de eventos de prueba de concepto para su entorno de desarrollo, como se hizo en el ejemplo de eShopOnContainers, una implementación simple sobre RabbitMQ que se ejecuta en un contenedor Docker podría ser suficiente.

Sin embargo, para los sistemas de producción y de misión crítica que necesitan hiperescalabilidad, es posible que desee evaluar Azure Service Bus. Para funciones y abstracciones de alto nivel que facilitan el desarrollo de aplicaciones distribuidas, le recomendamos que evalúe otros buses de servicio comerciales y de código abierto, como NServiceBus, MassTransit y Brighter. Por supuesto, puedes construir tu propio

funciones de bus de servicio además de tecnologías de nivel inferior como RabbitMQ y Docker. Pero ese trabajo de plomería puede costar demasiado para una aplicación empresarial personalizada.

## Publicación resistente en el bus de eventos

Un desafío al implementar una arquitectura basada en eventos en múltiples microservicios es cómo actualizar atómicamente el estado en el microservicio original mientras se publica de manera resiliente su evento de integración relacionado en el bus de eventos, de alguna manera basado en transacciones. Las siguientes son algunas formas de lograr esta funcionalidad, aunque también podría haber enfoques adicionales.

- Usar una cola transaccional (basada en DTC) como MSMQ. (Sin embargo, este es un enfoque heredado).
- Uso de la minería de registros de transacciones.
- Uso del patrón [de abastecimiento de eventos](#) completo.
- Uso del [patrón Bandeja de salida](#): una tabla de base de datos transaccional como una cola de mensajes que será la base para un componente creador de eventos que crearía el evento y lo publicaría.

Los temas adicionales que se deben tener en cuenta al usar la comunicación asíncrona son la idempotencia de mensajes y la desduplicación de mensajes. Estos temas se tratan en la sección [Implementación de comunicación basada en eventos entre microservicios \(eventos de integración\)](#) más adelante en esta guía.

## Recursos adicionales

- Mensajería basada en eventos  
[https://patterns.arcitura.com/soa-patterns/design\\_patterns/event\\_driven\\_messaging](https://patterns.arcitura.com/soa-patterns/design_patterns/event_driven_messaging)


---
- Canal de publicación/suscripción  
<https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>


---
- Udi Dahan. CQRS aclarado  
<https://udidahan.com/2009/12/09/clarified-cqrs/>


---
- Segregación de responsabilidad de consultas y comandos (CQRS) <https://docs.microsoft.com/azure/architecture/patterns/cqrs>


---
- Comunicación entre contextos acotados  
[https://docs.microsoft.com/versiones-anteriores/msp-np/jj591572\(v=pandp.10\)](https://docs.microsoft.com/versiones-anteriores/msp-np/jj591572(v=pandp.10))


---
- Coherencia eventual  
[https://en.wikipedia.org/wiki/Eventual\\_consistency](https://en.wikipedia.org/wiki/Eventual_consistency)


---
- Jimmy Bogard. Refactorización hacia la resiliencia: evaluación del acoplamiento  
<https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/>


---

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

Una API de microservicio es un contrato entre el servicio y sus clientes. Podrá desarrollar un microservicio de forma independiente solo si no rompe su contrato de API, razón por la cual el contrato es tan importante. Si cambia el contrato, afectará a sus aplicaciones cliente o su puerta de enlace API.

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

Sin embargo, incluso si está pensando en su contrato inicial, una API de servicio deberá cambiar con el tiempo. Cuando eso sucede, y especialmente si su API es una API pública consumida por varias aplicaciones de clientes, normalmente no puede obligar a todos los clientes a actualizarse a su nuevo contrato de API. Por lo general, debe implementar de forma incremental nuevas versiones de un servicio de manera que tanto la versión antigua como la nueva de un contrato de servicio se ejecuten simultáneamente. Por lo tanto, es importante tener una estrategia para el control de versiones de su servicio.

Cuando los cambios de API son pequeños, como si agrega atributos o parámetros a su API, los clientes que usan una API anterior deben cambiar y trabajar con la nueva versión del servicio. Es posible que pueda proporcionar valores predeterminados para cualquier atributo que falte y que sea necesario, y los clientes podrían ignorar cualquier atributo de respuesta adicional.

Sin embargo, a veces es necesario realizar cambios importantes e incompatibles en una API de servicio. Debido a que es posible que no pueda forzar que las aplicaciones o los servicios del cliente se actualicen inmediatamente a la nueva versión, un servicio debe admitir versiones anteriores de la API durante algún tiempo. Si está utilizando un mecanismo basado en HTTP como REST, un enfoque es incrustar el número de versión de la API en la URL o en un encabezado HTTP. Luego, puede decidir entre implementar ambas versiones del servicio simultáneamente dentro de la misma instancia de servicio o implementar diferentes instancias que manejen cada una una versión de la API. Un buen enfoque para esta funcionalidad es el [patrón Mediator](#) (por ejemplo, [la biblioteca MediatR](#)) para desacoplar las diferentes [versiones de implementación](#) en controladores independientes.

Finalmente, si usa una arquitectura REST, [Hypermedia es la mejor](#) solución para versionar sus servicios y permitir API evolutivas.

### Recursos adicionales

- Scott Hanselman. Control de versiones de ASP.NET Core RESTful Web API simplificado  
<https://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>

---
- Versión 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, hipermedia y REST  
<https://www.infoq.com/articles/roy-fielding-on-versioning>

---

## Direccionabilidad de microservicios y registro de servicios

Cada microservicio tiene un nombre único (URL) que se usa para resolver su ubicación. Su microservicio debe ser direccionable dondequiera que se esté ejecutando. Si tiene que pensar en qué computadora está ejecutando un microservicio en particular, las cosas pueden salir mal rápidamente. De la misma manera que el DNS resuelve una URL en una computadora en particular, su microservicio debe tener un nombre único para que su ubicación actual sea reconocible. Los microservicios necesitan nombres direccionables que los hagan independientes de la infraestructura en la que se ejecutan. Este enfoque implica que existe una interacción entre cómo se implementa su servicio y cómo se descubre, porque debe haber un [registro de servicio](#). Del mismo modo, cuando falla una computadora, el servicio de registro debe poder indicar dónde se está ejecutando el [servicio](#).

El [patrón de registro de servicios](#) es una parte clave del descubrimiento de servicios. El registro es una base de datos que contiene las ubicaciones de red de las instancias de servicio. Un registro de servicios debe tener alta disponibilidad y estar actualizado. Los clientes pueden almacenar en caché las ubicaciones de red obtenidas del registro de servicios. Sin embargo, esa información eventualmente queda obsoleta y los clientes ya no pueden descubrir instancias de servicio. Por lo tanto, un registro de servicios consta de un grupo de servidores que utilizan un protocolo de replicación para mantener la coherencia.

En algunos entornos de implementación de microservicios (llamados clústeres, que se tratarán en una sección posterior), el descubrimiento de servicios está integrado. Por ejemplo, un entorno de Azure Kubernetes Service (AKS) puede manejar el registro y la anulación del registro de instancias de servicio. También ejecuta un proxy en cada host de clúster que desempeña la función de enrutador de descubrimiento del lado del servidor.

## Recursos adicionales

- Chris Richardson. Patrón: registro de servicio <https://microservices.io/patterns/service-registry.html>
- Autor0. El registro de servicios <https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/>
- Gabriel Schenker. Descubrimiento de servicios <https://lostechies.com/gabrielschenker/2016/01/27/service-discovery/>

## Creación de una interfaz de usuario compuesta basada en microservicios

La arquitectura de microservicios a menudo comienza con el manejo de datos y lógica del lado del servidor, pero, en muchos casos, la interfaz de usuario todavía se maneja como un monolito. Sin embargo, un enfoque más avanzado, llamado [micro frontends](#), es diseñar la interfaz de usuario de su aplicación también en función de los microservicios. Eso significa tener una interfaz de usuario compuesta producida por los microservicios, en lugar de tener microservicios en el servidor y solo una aplicación de cliente monolítica que consume los microservicios. Con este enfoque, los microservicios que crea pueden estar completos con representación lógica y visual.

La Figura 4-20 muestra el enfoque más simple de solo consumir microservicios de una aplicación de cliente monolítica. Por supuesto, podría tener un servicio ASP.NET MVC entre la producción de HTML y JavaScript. La figura es una simplificación que destaca que tiene una única interfaz de usuario de cliente (monolítica)

consumiendo 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

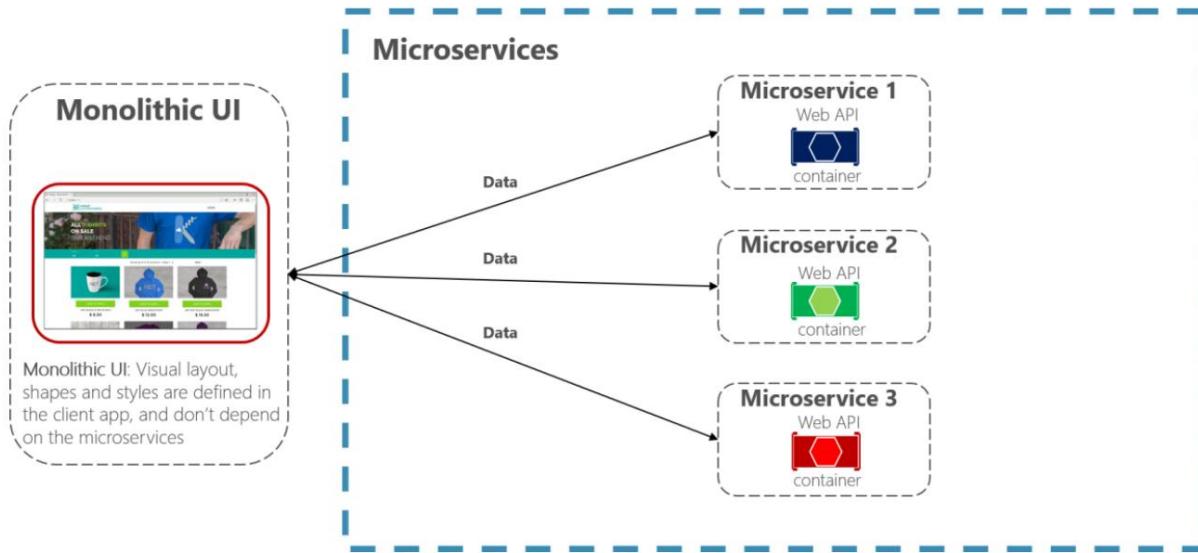


Figura 4-20. Una aplicación de interfaz de usuario monolítica que consume microservicios de back-end

Por el contrario, una interfaz de usuario compuesta es generada y compuesta con precisión por los propios microservicios. Algunos de los microservicios impulsan la forma visual de áreas específicas de la interfaz de usuario. La diferencia clave es que tiene componentes de IU de cliente (clases de TypeScript, por ejemplo) basados en plantillas, y el ViewModel de IU de modelado de datos para esas plantillas proviene de cada microservicio.

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

La Figura 4-21 muestra una versión de este enfoque de interfaz de usuario compuesta. Este enfoque se simplifica porque es posible que tenga otros microservicios que agregan partes granulares que se basan en diferentes técnicas. Depende de si está creando un enfoque web tradicional (ASP.NET MVC) o una SPA (aplicación de página única).

## Composite UI generated by microservices

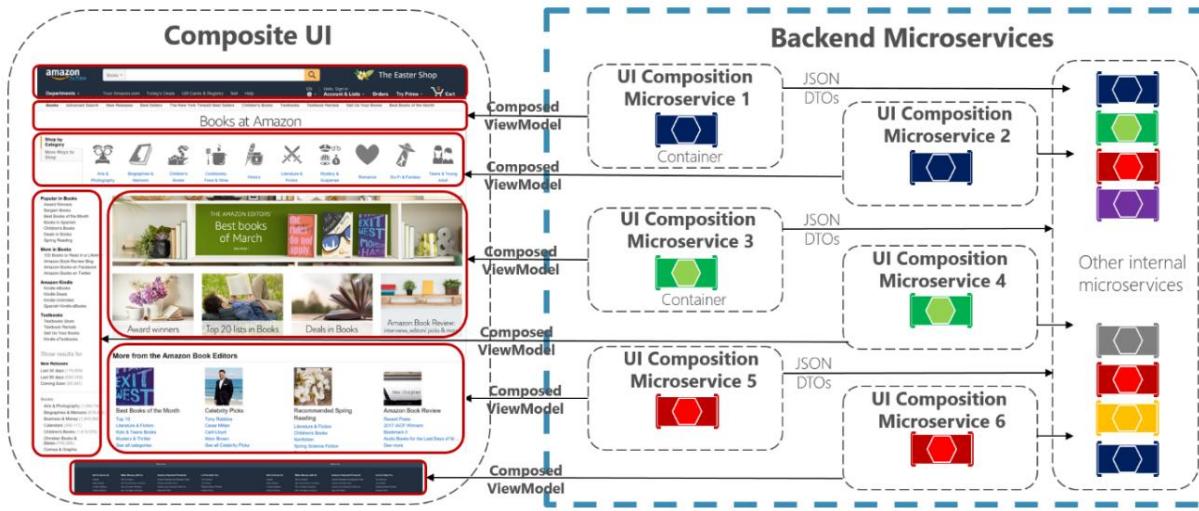


Figura 4-21. Ejemplo de una aplicación de interfaz de usuario compuesta formada por microservicios de back-end

Cada uno de esos microservicios de composición de interfaz de usuario sería similar a una puerta de enlace 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 compuesto impulsado por microservicios puede ser más desafiante o menos, según las tecnologías de interfaz de usuario que esté utilizando. Por ejemplo, no usará las mismas técnicas para crear una aplicación web tradicional que usa para crear un SPA o una aplicación móvil nativa (como cuando desarrolla aplicaciones de Xamarin, que pueden ser más desafiantes para este enfoque).

La aplicación de ejemplo [eShopOnContainers](#) utiliza el enfoque de interfaz de usuario monolítica por varias razones. Primero, es una introducción a los microservicios y contenedores. Una interfaz de usuario compuesta es más avanzada, pero también requiere una mayor complejidad al diseñar y desarrollar la interfaz de usuario. En segundo lugar, eShopOnContainers también proporciona una aplicación móvil nativa basada en Xamarin, lo que la haría más compleja en el lado C# del cliente.

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

### Recursos adicionales

- Micro Frontends (blog de Martin Fowler) <https://martinfowler.com/articles/micro-frontends.html>
- Micro Frontends (sitio de Michael Geers) <https://micro-frontends.org/>
- Interfaz de usuario compuesta usando ASP.NET (Taller de Particular) <https://github.com/Particular/Workshop/tree/master/demos/asp-net-core>
- Rubén Oostinga. El frontend monolítico en la arquitectura de microservicios <https://xebia.com/blog/el-frontend-monolitico-en-la-arquitectura-de-microservicios/>

- Mauro Servienti. El secreto de una mejor composición de la interfaz de usuario  
<https://particular.net/blog/secret-of-better-ui-composition>


---
- Víctor Farcic. Inclusión de componentes web front-end en microservicios <https://technologyconversations.com/2015/08/09/incluyendo-componentes-web-front-end-into-microservices/>


---
- Gestión de Frontend en la Arquitectura de Microservicios  
<https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html>


---

## Resiliencia y alta disponibilidad en microservicios

Lidiar con fallas inesperadas es uno de los problemas más difíciles de resolver, especialmente en un sistema distribuido. Gran parte del código que escriben los desarrolladores implica el manejo de excepciones, y aquí también es donde se dedica la mayor parte del tiempo a las pruebas. El problema es más complicado que escribir código para manejar fallas.

¿Qué sucede cuando falla la máquina donde se ejecuta el microservicio? No solo necesita detectar esta falla de microservicio (un problema difícil en sí mismo), sino que también necesita algo para reiniciar su microservicio.

Un microservicio debe ser resistente a las fallas y poder reiniciarse con frecuencia en otra máquina para disponibilidad. Esta resiliencia también se reduce al estado que se guardó en nombre del microservicio, desde dónde el microservicio puede recuperar este estado y si el microservicio puede reiniciarse correctamente. En otras palabras, debe haber resiliencia en la capacidad informática (el proceso puede reiniciarse en cualquier momento), así como resiliencia en el estado o los datos (sin pérdida de datos y los datos permanecen consistentes).

Los problemas de resiliencia se agravan en otros escenarios, como cuando se producen errores durante la actualización de una aplicación. El microservicio, que trabaja con el sistema de implementación, debe determinar si puede continuar avanzando a la versión más nueva o, en cambio, retroceder a una versión anterior para mantener un estado coherente. Deben considerarse cuestiones tales como si hay suficientes máquinas disponibles para seguir avanzando y cómo recuperar versiones anteriores del microservicio. Este enfoque requiere que el microservicio emita información de estado para que la aplicación general y el orquestador puedan tomar estas decisiones.

Además, la resiliencia está relacionada con cómo deben comportarse los sistemas basados en la nube. Como se mencionó, un sistema basado en la nube debe aceptar las fallas y debe tratar de recuperarse automáticamente de ellas. Por ejemplo, en caso de fallas en la red o en el contenedor, las aplicaciones de los clientes o los servicios de los clientes deben tener una estrategia para volver a intentar enviar mensajes o reintentar solicitudes, ya que en muchos casos las fallas en la nube son parciales. La sección [Implementación de aplicaciones resilientes](#) de esta guía aborda cómo manejar fallas parciales. Describe técnicas como reintentos con retroceso exponencial o el patrón Circuit Breaker en .NET mediante el uso de bibliotecas como [Polly](#), que ofrece una gran variedad de políticas para manejar este tema.

## Gestión de salud y diagnóstico en microservicios

Puede parecer obvio y, a menudo, se pasa por alto, pero un microservicio debe informar sobre su estado y diagnóstico. De lo contrario, hay poca información desde la perspectiva de las operaciones. Correlacionar eventos de diagnóstico a través de un conjunto de servicios independientes y tratar con sesgos de reloj de máquina para dar sentido a

el orden de los eventos es desafiante. De la misma manera que interactúa con un microservicio sobre protocolos y formatos de datos acordados, existe la necesidad de estandarizar la forma de registrar eventos de estado y diagnóstico que, en última instancia, terminan en un almacén de eventos para consulta y visualización. En un enfoque de microservicios, es clave que los diferentes equipos acuerden un único formato de registro. Debe haber un enfoque coherente para ver los eventos de diagnóstico en la aplicación.

### controles de salud

La salud es diferente del diagnóstico. La salud se trata de que el microservicio informe su estado actual para tomar las medidas apropiadas. Un buen ejemplo es trabajar con mecanismos de actualización e implementación para mantener la disponibilidad. Aunque es posible que un servicio no esté en buen estado debido a un bloqueo del proceso o al reinicio de la máquina, es posible que el servicio aún esté operativo. Lo último que necesita es empeorar esto realizando una actualización. El mejor enfoque es hacer una investigación primero o dejar tiempo para que el microservicio se recupere. Los eventos de salud de un microservicio nos ayudan a tomar decisiones informadas y, de hecho, ayudan a crear servicios de recuperación automática.

En la sección [Implementación de comprobaciones de estado en los servicios ASP.NET Core](#) de esta guía, explicamos cómo usar una nueva biblioteca ASP.NET HealthChecks en sus microservicios para que puedan informar su estado a un servicio de supervisión para tomar las medidas adecuadas.

También tiene la opción de usar una excelente biblioteca de código abierto llamada `AspNetCore.Diagnostics.HealthChecks`, disponible en [GitHub](#) y [como paquete NuGet](#). Esta biblioteca también realiza comprobaciones de estado, con un giro, maneja dos tipos de comprobaciones:

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

### Uso de flujos de eventos de diagnósticos y registros

Los registros brindan información sobre cómo se ejecuta una aplicación o servicio, incluidas excepciones, advertencias y mensajes informativos simples. Por lo general, cada registro tiene un formato de texto con una línea por evento, aunque las excepciones también suelen mostrar el seguimiento de la pila en varias líneas.

En aplicaciones monolíticas basadas en servidor, puede escribir registros en un archivo en el disco (un archivo de registro) y luego analizarlo con cualquier herramienta. Dado que la ejecución de la aplicación está limitada a un servidor fijo o VM, generalmente no es demasiado complejo analizar el flujo de eventos. Sin embargo, en una aplicación distribuida donde se ejecutan múltiples servicios a través de muchos nodos en un clúster orquestador, poder correlacionar eventos distribuidos es un desafío.

Una aplicación basada en microservicios no debe tratar de almacenar el flujo de salida de eventos o archivos de registro por sí misma, y ni siquiera tratar de administrar el enruteamiento de los eventos a un lugar central. Debe ser transparente, lo que significa que cada proceso debe simplemente escribir su flujo de eventos en una salida estándar que debajo será recopilada por la infraestructura del entorno de ejecución donde se ejecuta. Un ejemplo de estos enruteadores de flujo de eventos es [Microsoft.Diagnostic.EventFlow](#), que [recopila flujos de eventos de varias fuentes](#) y los publica en los sistemas de salida. Estos pueden incluir una salida estándar simple para un entorno de desarrollo o sistemas en la nube como [Azure Monitor](#) y [Azure Diagnostics](#). También existen buenas plataformas y [herramientas de análisis de registros de terceros](#) que pueden buscar, alertar, informar y monitorear registros, incluso en tiempo real, como [Splunk](#).

## Orquestadores que gestionan la información de salud y diagnóstico

Cuando crea una aplicación basada en microservicios, debe lidiar con la complejidad. Por supuesto, un solo microservicio es fácil de manejar, pero docenas o cientos de tipos y miles de instancias de microservicios son un problema complejo. No se trata solo de construir su arquitectura de microservicio, también necesita alta disponibilidad, capacidad de direccionamiento, resiliencia, estado y diagnóstico si pretende tener un sistema estable y cohesivo.



Figura 4-22. Una Plataforma de Microservicios es fundamental para la gestión de la salud de una aplicación

Los problemas complejos que se muestran en la figura 4-22 son difíciles de resolver por sí mismo. Los equipos de desarrollo deben centrarse en resolver problemas comerciales y crear aplicaciones personalizadas con enfoques basados en microservicios. No deben enfocarse en resolver problemas complejos de infraestructura; si lo hicieran, el costo de cualquier aplicación basada en microservicios sería enorme. Por lo tanto, existen plataformas orientadas a microservicios, denominadas orquestadores o clústeres de microservicios, que intentan resolver los problemas difíciles de crear y ejecutar un servicio y utilizar los recursos de infraestructura de manera eficiente. Este enfoque reduce las complejidades de crear aplicaciones que utilizan un enfoque de microservicios.

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

## Recursos adicionales

- La aplicación de Doce Factores. XI. Registros: trate los registros como flujos de eventos  
<https://12factor.net/logs>
- Repositorio de GitHub de la biblioteca EventFlow de Microsoft Diagnostic .  
<https://github.com/Azure/diagnostics-eventflow>
- ¿Qué es Azure Diagnostics?  
<https://docs.microsoft.com/azure/azure-diagnostics>

- Conecte equipos con Windows al servicio Azure Monitor <https://docs.microsoft.com/azure/azure-monitor/platform/agent-windows>
- Registro de lo que quiere decir: uso del bloque de aplicación de registro semántico  
[https://docs.microsoft.com/previous-versions/msp-np/dn440729\(v=pandp.60\)](https://docs.microsoft.com/previous-versions/msp-np/dn440729(v=pandp.60))
- Sitio oficial de Splunk .  
<https://www.splunk.com/>
- API de clase EventSource para seguimiento de eventos para Windows (ETW) <https://docs.microsoft.com/dotnet/api/system.diagnostics.tracing.eventsource>

## Organice microservicios y aplicaciones de múltiples contenedores para una alta escalabilidad y disponibilidad

El uso de orquestadores para aplicaciones listas para producción es esencial si su aplicación se basa en microservicios o simplemente se divide en varios contenedores. Como se presentó anteriormente, en un enfoque basado en microservicios, cada microservicio posee su modelo y sus datos, por lo que será autónomo desde el punto de vista del desarrollo y la implementación. Pero incluso si tiene una aplicación más tradicional que se compone de múltiples servicios (como SOA), también tendrá múltiples contenedores o servicios que comprenden una sola aplicación comercial que debe implementarse como un sistema distribuido. Estos tipos de sistemas son complejos de escalar y administrar; por lo tanto, es absolutamente necesario un orquestador si desea tener una aplicación multicontenedor escalable y lista para la producción.

La figura 4-23 ilustra la implementación en un clúster de una aplicación compuesta 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

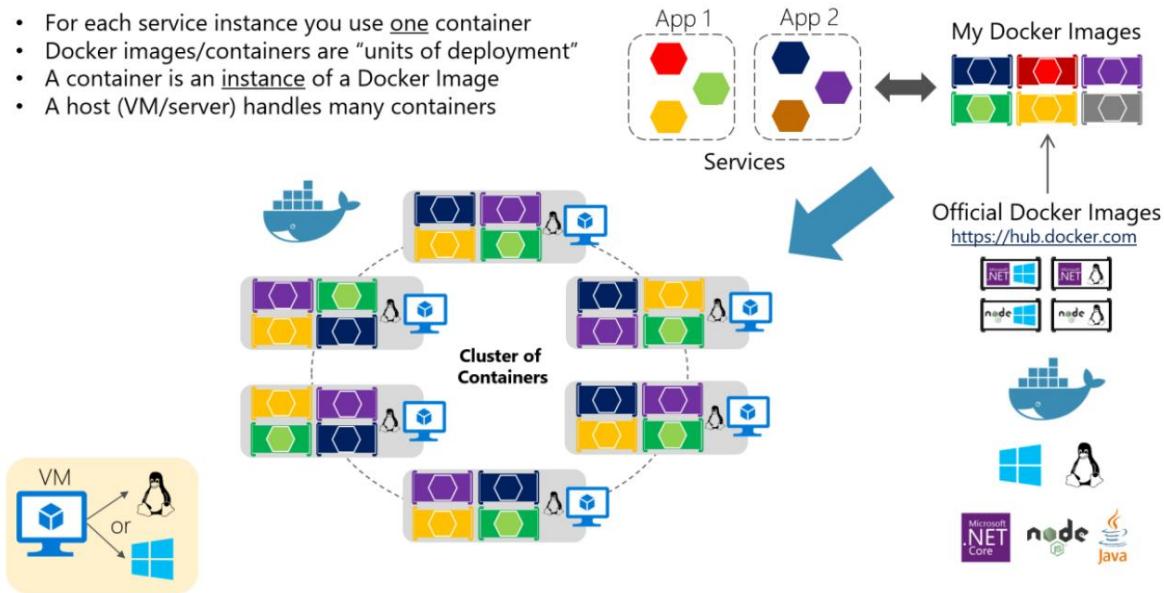


Figura 4-23. Un grupo de contenedores

Utiliza un contenedor para cada instancia de servicio. Los contenedores Docker son "unidades de implementación" y un contenedor es una instancia de Docker. Un anfitrión maneja muchos contenedores. Parece un enfoque lógico.

Pero, ¿cómo maneja el equilibrio de carga, el enruteamiento y la orquestación de estas aplicaciones compuestas?

El Docker Engine simple en hosts Docker únicos satisface las necesidades de administrar instancias de una sola imagen en un host, pero se queda corto cuando se trata de administrar múltiples contenedores implementados en múltiples hosts para aplicaciones distribuidas más complejas. En la mayoría de los casos, necesita una plataforma de administración que inicie contenedores automáticamente, los escale horizontalmente con varias instancias por imagen, los suspenda o apague cuando sea necesario e, idealmente, también controle cómo acceden a recursos como la red y el almacenamiento de datos.

Para ir más allá de la administración de contenedores individuales o aplicaciones compuestas simples y avanzar hacia aplicaciones empresariales más grandes con microservicios, debe recurrir a plataformas de orquestación y agrupación.

Desde el punto de vista de la arquitectura y el desarrollo, si está creando una gran empresa compuesta por aplicaciones basadas en microservicios, es importante comprender las siguientes plataformas y productos que admiten escenarios avanzados:

**Clústeres y orquestadores.** Cuando necesita escalar aplicaciones en muchos hosts de Docker, como cuando se trata de una gran aplicación basada en microservicios, es fundamental poder administrar todos esos hosts como un solo clúster abstrayendo la complejidad de la plataforma subyacente. Eso es lo que proporcionan los clústeres de contenedores 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 contenedores en un clúster para que también proporcionen una interfaz de usuario. Un planificador de clústeres tiene varias responsabilidades: utilizar el clúster

recursos de manera eficiente, para establecer las restricciones proporcionadas por el usuario, para equilibrar de manera eficiente la carga de contenedores entre nodos o hosts, y para ser robusto contra errores al tiempo que proporciona alta disponibilidad.

Los conceptos de un clúster y un programador están estrechamente relacionados, por lo que los productos proporcionados por diferentes proveedores a menudo brindan ambos conjuntos de capacidades. La siguiente lista muestra las opciones de plataforma y software más importantes que tiene para clústeres y programadores. Estos orquestadores generalmente se ofrecen en nubes públicas como Azure.

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

Plataforma	Descripción
Kubernetes 	<p><a href="#">Kubernetes</a> es un producto de código abierto que proporciona una funcionalidad que va desde la infraestructura del clúster y la programación de contenedores hasta las capacidades de orquestación. Le permite automatizar la implementación, el escalado y las operaciones de contenedores de aplicaciones en clústeres de hosts.</p> <p>Kubernetes proporciona una infraestructura centrada en contenedores que agrupa contenedores de aplicaciones en unidades lógicas para facilitar la gestión y el descubrimiento.</p> <p>Kubernetes es maduro en Linux, menos maduro en Windows.</p>
Azure Kubernetes Servicio (AKS) 	<p><a href="#">AKS</a> es un servicio de orquestación de contenedores de Kubernetes administrado en Azure que simplifica la administración, la implementación y las operaciones del clúster de Kubernetes.</p>
Contenedor azul aplicaciones 	<p><a href="#">Azure Container Apps</a> es un servicio de contenedor sin servidor administrado para crear e implementar aplicaciones modernas a escala.</p>

## Uso de orquestadores basados en contenedores en Microsoft Azure

Varios proveedores de la nube ofrecen compatibilidad con contenedores Docker, además de clústeres de Docker y compatibilidad con orquestación, incluidos Microsoft Azure, Amazon EC2 Container Service y Google Container Engine. Microsoft Azure proporciona compatibilidad con orquestadores y clústeres de Docker a través de Azure Kubernetes Service (AKS).

### Uso del servicio Azure Kubernetes

Un clúster de Kubernetes agrupa varios hosts de Docker y los expone como un único host de Docker virtual, por lo que puede implementar varios contenedores en el clúster y escalar horizontalmente con cualquier cantidad de instancias de contenedor. El clúster manejará toda la plomería de administración compleja, como escalabilidad, salud, etc.

AKS proporciona una manera de simplificar la creación, configuración y administración de un clúster de máquinas virtuales en Azure que están preconfiguradas para ejecutar aplicaciones en contenedores. Usando un optimizado

Con la configuración de herramientas populares de programación y orquestación de código abierto, AKS le permite usar sus habilidades existentes o aprovechar un cuerpo grande y creciente de experiencia de la comunidad para implementar y administrar aplicaciones basadas en contenedores en Microsoft Azure.

Azure Kubernetes Service optimiza la configuración de las populares herramientas y tecnologías de código abierto de clústeres de Docker específicamente para Azure. Obtiene una solución abierta que ofrece portabilidad tanto para sus contenedores como para la configuración de su aplicación. Usted selecciona el tamaño, la cantidad de hosts y las herramientas de orquestación, y AKS se encarga de todo lo demás.

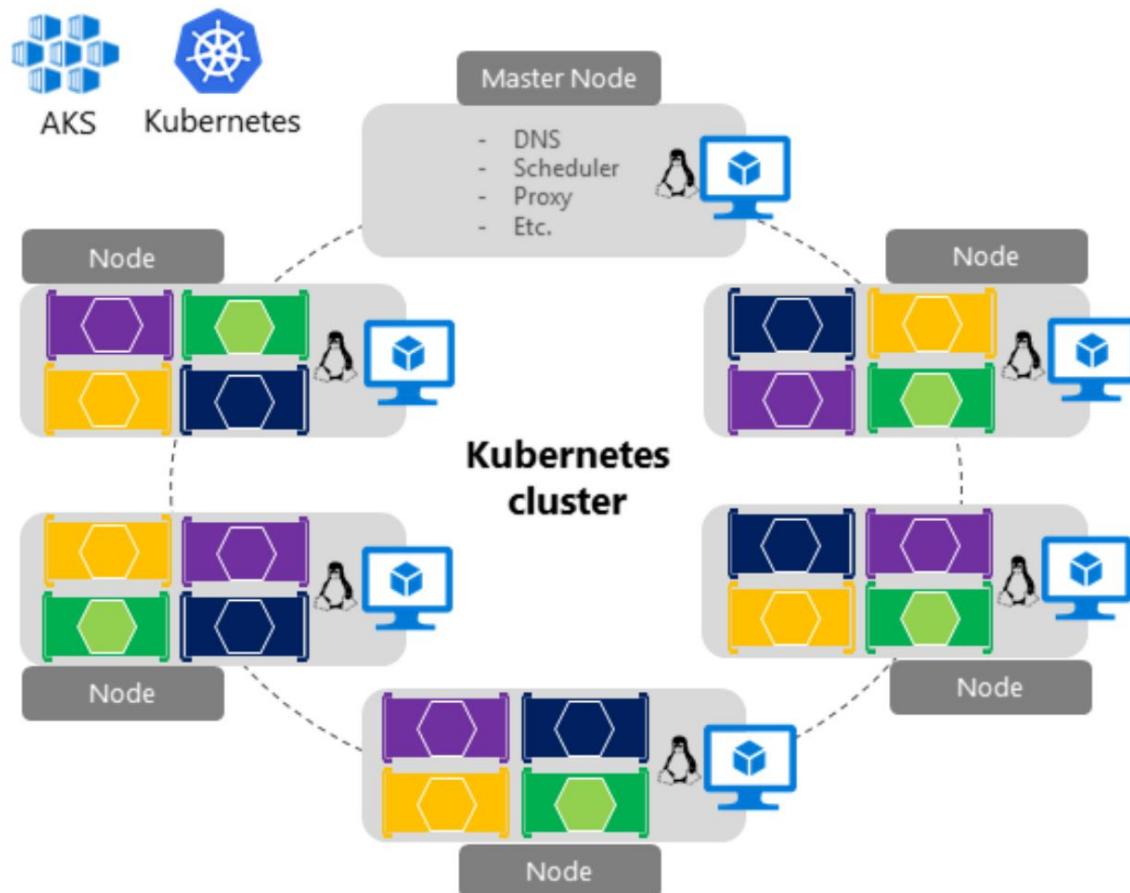


Figura 4-24. Estructura y topología simplificadas del clúster de Kubernetes

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 solo grupo desde un punto de vista de aplicación y le permite escalar a miles o incluso 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 una sola máquina de desarrollo (Windows 10 o macOS) al instalar [Docker Desktop](#). Posteriormente, [puede realizar](#) la implementación en la nube (AKS) para realizar más pruebas de integración, como se muestra en la figura 4-25.

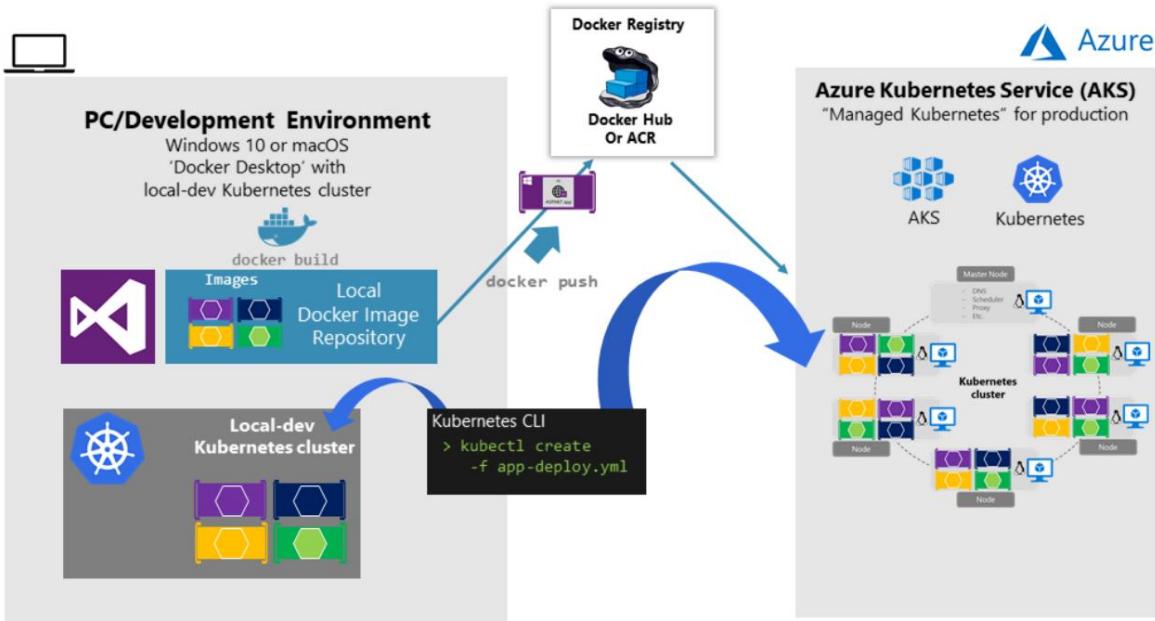


Figura 4-25. Ejecución de Kubernetes en la máquina de desarrollo y la nube

## Introducción a Azure Kubernetes Service (AKS)

Para comenzar a usar AKS, implemente un clúster de AKS desde Azure Portal o mediante la CLI. Para obtener más información sobre la implementación de un clúster de Kubernetes en Azure, consulte [Implementación de un clúster de Azure Kubernetes Service \(AKS\)](#).

No hay tarifas por ningún software instalado de forma predeterminada como parte de AKS. Todas las opciones predeterminadas se implementan con software de código abierto. AKS está disponible para varias máquinas virtuales en Azure.

Solo se le cobra por las instancias informáticas que elija y los demás recursos de infraestructura subyacente consumidos, como el almacenamiento y las redes. No hay cargos incrementales para AKS en sí.

La opción de implementación de producción predeterminada para Kubernetes es usar gráficos de Helm, que se presentan en la siguiente sección.

## Despliegue con gráficos de Helm en clústeres de Kubernetes

Al implementar una aplicación en un clúster de Kubernetes, puede usar la herramienta CLI kubectl.exe original mediante archivos de implementación basados en el formato nativo (archivos .yaml), como ya se mencionó en la sección anterior. Sin embargo, para aplicaciones de Kubernetes más complejas, como cuando se implementan aplicaciones complejas basadas en microservicios, se recomienda usar [Helm](#).

Helm Charts lo ayuda a definir, versionar, instalar, compartir, actualizar o revertir incluso la aplicación de Kubernetes más compleja.

Yendo más allá, también se recomienda el uso de Helm porque otros entornos de Kubernetes en Azure, como [Azure Dev Spaces](#), también se basan en gráficos de Helm.

Helm es mantenido por [Cloud Native Computing Foundation \(CNCF\)](#), 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, consulte la publicación [Uso de gráficos de Helm para implementar eShopOnContainers en AKS](#).

## Recursos adicionales

- Introducción a Azure Kubernetes Service (AKS) <https://docs.microsoft.com/azure/aks/kubernetes-walkthrough-portal>
- Espacios de desarrollo de Azure <https://docs.microsoft.com/azure/dev-spaces/azure-dev-spaces>
- Kubernetes El sitio oficial. <https://kubernetes.io/>

# Proceso de desarrollo para Aplicaciones basadas en Docker

Desarrolle aplicaciones .NET en contenedores de la manera que desee, ya sea un entorno de desarrollo integrado (IDE) enfocado con Visual Studio y las herramientas de Visual Studio para Docker o CLI/Editor enfocado con Docker CLI y Visual Studio Code.

## Entorno de desarrollo para aplicaciones Docker

### Opciones de herramientas de desarrollo: IDE o editor

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

Visual Studio (para Windows). El desarrollo de aplicaciones .NET 6 basadas en Docker con Visual Studio requiere Visual Studio 2022 versión 17.0 o posterior. Visual Studio 2022 viene con herramientas para Docker ya integradas. Las herramientas para Docker le permiten desarrollar, ejecutar y validar sus aplicaciones directamente en el entorno Docker de destino. Puede presionar F5 para ejecutar y depurar su aplicación (un solo contenedor o varios contenedores) directamente en un host de Docker, o presionar CTRL + F5 para editar y actualizar su aplicación sin tener que reconstruir el contenedor. Este IDE es la opción de desarrollo más poderosa para aplicaciones basadas en Docker.

Estudio Visual para Mac. Es un IDE, evolución de Xamarin Studio, que se ejecuta en macOS. Para el desarrollo de .NET 6, requiere la versión 8.4 o posterior. Esta herramienta debería ser la opción preferida para los desarrolladores que trabajan en máquinas macOS que también desean usar un IDE potente.

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

Al instalar [Docker Desktop](#), puede usar una sola CLI de Docker para crear aplicaciones para Windows y Linux.

## Recursos adicionales

- Estudio visual. Sitio oficial.  
<https://visualstudio.microsoft.com/vs/>
- Código de estudio visual. Sitio oficial.  
<https://code.visualstudio.com/download>
- Docker Desktop para Windows <https://hub.docker.com/editions/community/docker-ce-desktop-windows>
- Docker Desktop para Mac  
<https://hub.docker.com/editions/community/docker-ce-desktop-mac>

## Lenguajes y marcos .NET para contenedores Docker

Como se mencionó en secciones anteriores de esta guía, puede usar .NET Framework, .NET 6 o el proyecto Mono de código abierto al desarrollar aplicaciones .NET en contenedores de Docker. Puede desarrollar en C#, F# o Visual Basic cuando se dirige a contenedores de Linux o Windows, según el marco de .NET que esté en uso. Para obtener más detalles sobre los lenguajes .NET, consulte la publicación de blog [The .NET Language Strategy](#).

---

## Flujo de trabajo de desarrollo para aplicaciones Docker

El ciclo de vida del desarrollo de aplicaciones comienza en su computadora, como desarrollador, donde codifica la aplicación usando su idioma preferido y la prueba localmente. Con este flujo de trabajo, independientemente del idioma, el marco y la plataforma que elija, siempre está desarrollando y probando contenedores Docker, pero lo hace localmente.

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 Núcleo de servidor de Windows.
- Archivos agregados durante el desarrollo, por ejemplo, código fuente y binarios de aplicaciones.
- Información de configuración, como la configuración del entorno y las dependencias.

## Flujo de trabajo para desarrollar aplicaciones basadas en contenedores Docker

Esta sección describe el **flujo de trabajo de desarrollo** de bucle interno para aplicaciones basadas en contenedores de Docker. El flujo de trabajo de ciclo interno significa que no está considerando el flujo de trabajo de DevOps más amplio, que puede incluir hasta la implementación de producción, y solo se enfoca en el trabajo de desarrollo realizado en la computadora del desarrollador. Los pasos iniciales para configurar el entorno no están incluidos, ya que esos pasos se realizan solo una vez.

Una aplicación se compone de sus propios servicios más bibliotecas adicionales (dependencias). Los siguientes son los pasos básicos que suele seguir al crear una aplicación Docker, como se ilustra en la Figura 5-1.

## Inner-Loop development workflow for Docker apps

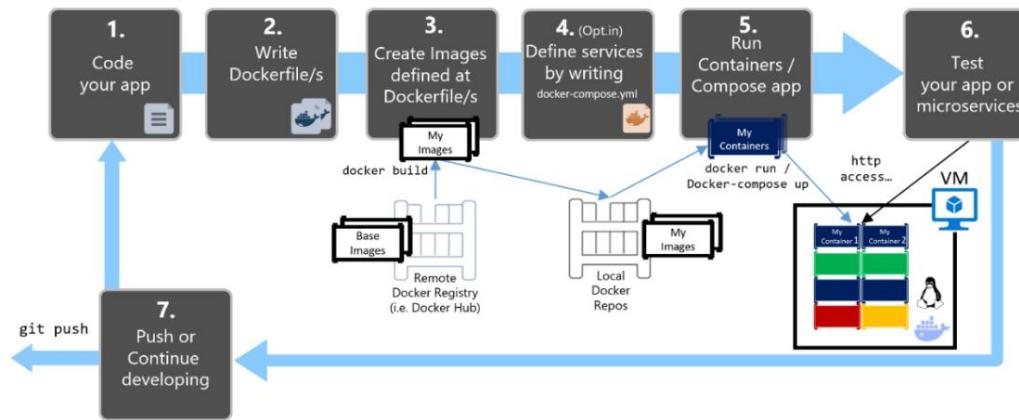


Figura 5-1. Flujo de trabajo paso a paso para desarrollar aplicaciones en contenedores de Docker

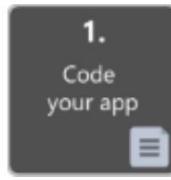
En esta sección, se detalla todo este proceso y se explica cada paso importante centrándose en un entorno de Visual Studio.

Cuando usa un enfoque de desarrollo de editor/CLI (por ejemplo, Visual Studio Code más Docker CLI en macOS o Windows), necesita conocer cada paso, generalmente con más detalle que si usa Visual Studio. Para obtener más información sobre cómo trabajar en un entorno CLI, consulte el libro electrónico [Ciclo de vida de la aplicación Docker en contenedores con plataformas y herramientas de Microsoft](#).

Cuando usa Visual Studio 2022, muchos de esos pasos se manejan por usted, lo que mejora drásticamente su productividad. Esto es especialmente cierto cuando usa Visual Studio 2022 y se dirige a aplicaciones de varios contenedores. Por ejemplo, con solo un clic del mouse, Visual Studio agrega el archivo Dockerfile y docker-compose.yml a sus proyectos con la configuración de su aplicación.

Cuando ejecuta la aplicación en Visual Studio, crea la imagen de Docker y ejecuta la aplicación de contenedores múltiples directamente en Docker; incluso le permite depurar varios contenedores a la vez. Estas características impulsarán su velocidad de desarrollo.

Sin embargo, el hecho de que Visual Studio haga que esos pasos sean automáticos no significa que no necesite saber qué sucede debajo de Docker. Por lo tanto, la siguiente guía detalla cada paso.



## Paso 1. Comience a codificar y cree su aplicación inicial o línea base de servicio

El desarrollo de una aplicación Docker es similar a la forma en que desarrolla una aplicación sin Docker. La diferencia es que mientras desarrolla para Docker, está implementando y probando su aplicación o servicios que se ejecutan dentro de los contenedores de Docker en su entorno local (ya sea una configuración de máquina virtual de Linux por Docker o directamente Windows si usa contenedores de Windows).

### Configure su entorno local con Visual Studio

Para comenzar, asegúrese de tener [Docker Desktop para Windows](#) para Windows, como se explica en las siguientes instrucciones:

#### [Comience con Docker Desktop para Windows](#)

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

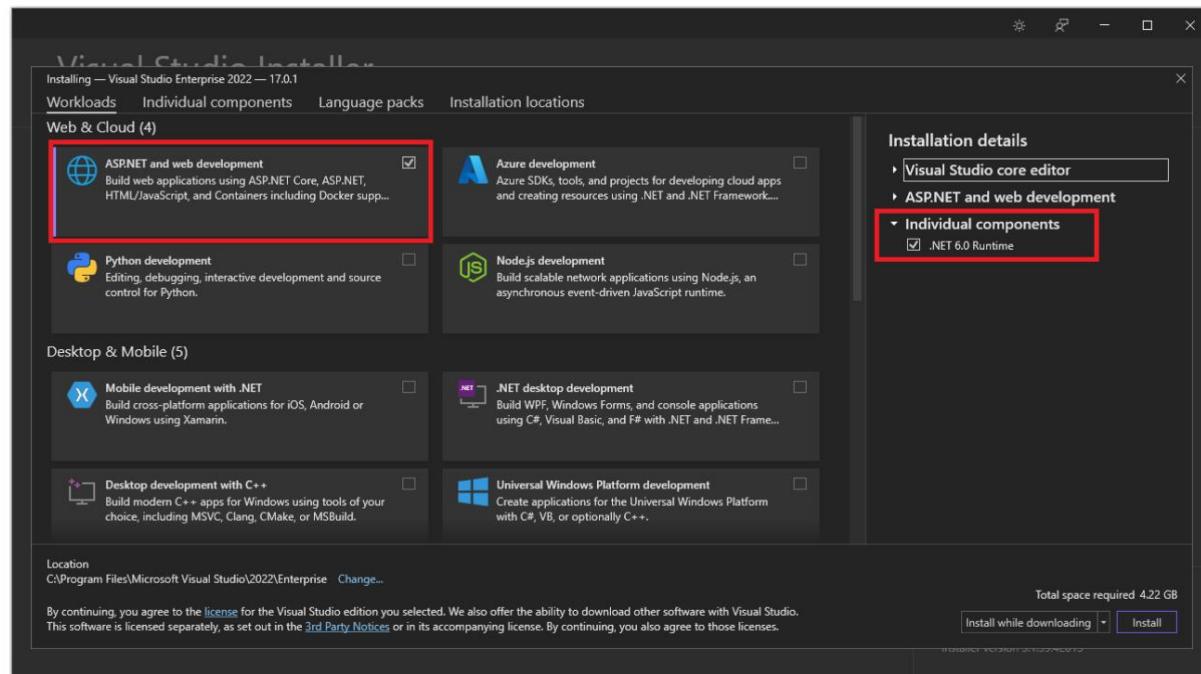


Figura 5-2. Selección de la carga de trabajo de ASP.NET y desarrollo web durante la configuración de Visual Studio 2022

Puede comenzar a codificar su aplicación en .NET simple (generalmente en .NET Core o posterior si planea usar contenedores) incluso antes de habilitar Docker en su aplicación e implementar y probar en Docker.

Sin embargo, se recomienda que comience a trabajar en Docker lo antes posible, porque ese será el entorno real y cualquier problema se puede descubrir lo antes posible. esto se anima

porque Visual Studio facilita tanto el trabajo con Docker que casi se siente transparente: el mejor ejemplo cuando se depuran aplicaciones de varios contenedores desde Visual Studio.

## Recursos adicionales

- Comience con Docker Desktop para Windows <https://docs.docker.com/docker-for-windows/>
- estudio visual 2022  
<https://visualstudio.microsoft.com/downloads/>



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

Necesita un Dockerfile para cada imagen personalizada que desee crear; también necesita un Dockerfile para cada contenedor que se implementará, ya sea que implemente automáticamente desde Visual Studio o manualmente mediante la CLI de Docker (comandos docker run y docker-compose). Si su aplicación contiene un solo servicio personalizado, necesita un solo Dockerfile. Si su 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 su aplicación o servicio. Contiene los comandos que le indican a Docker cómo configurar y ejecutar su aplicación o servicio en un contenedor. Puede crear manualmente un Dockerfile en código y agregarlo a su proyecto junto con sus dependencias de .NET.

Con Visual Studio y sus herramientas para Docker, esta tarea requiere solo unos pocos clics del mouse. Cuando crea un nuevo proyecto en Visual Studio 2022, hay una opción llamada Habilitar compatibilidad con Docker, como se muestra en la Figura 5-3.

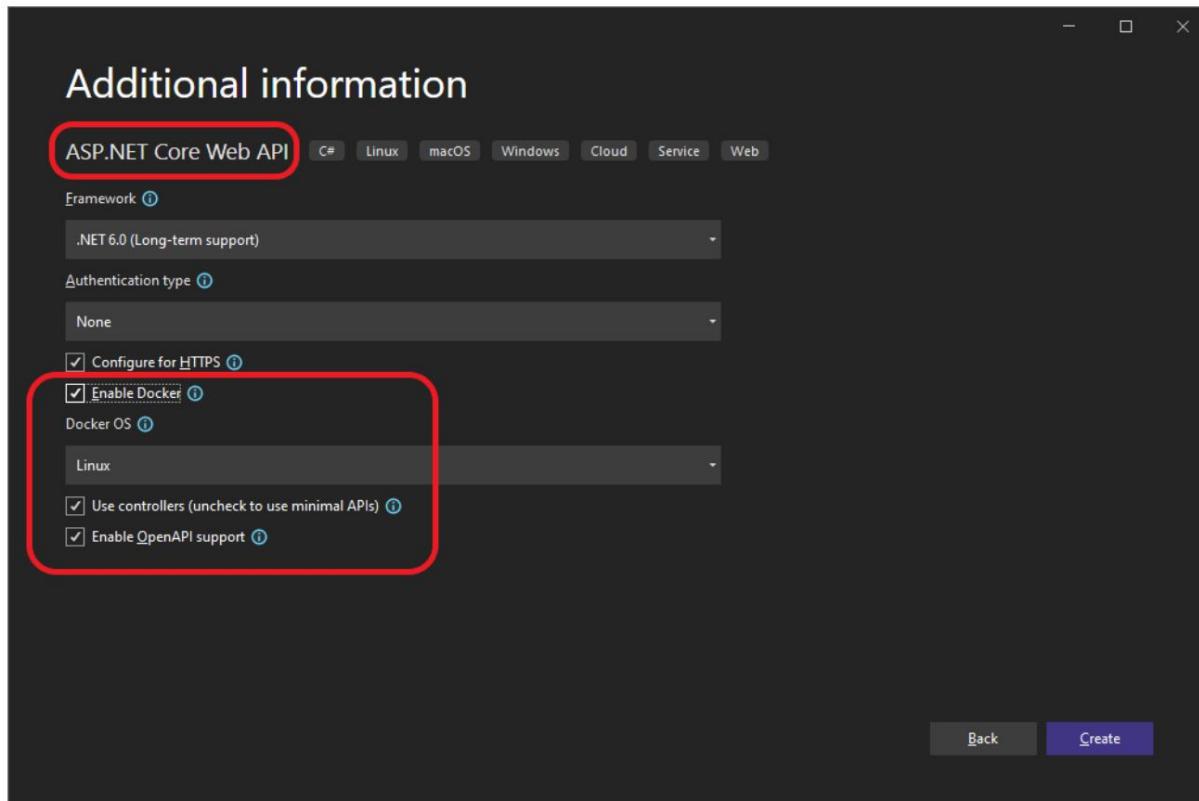


Figura 5-3. Habilidades de Docker Support al crear un nuevo proyecto ASP.NET Core en Visual Studio 2022

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

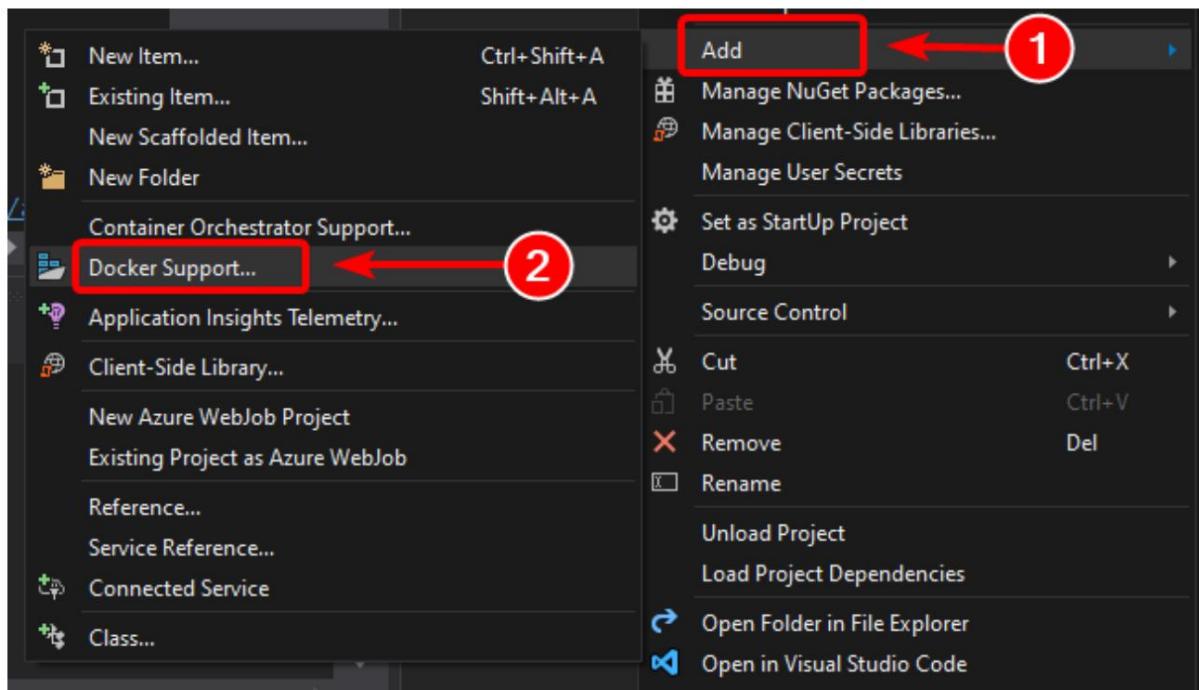


Figura 5-4. Habilidades de la compatibilidad con Docker en un proyecto de Visual Studio 2022 existente

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

De manera similar, Visual Studio también puede agregar un archivo docker-compose.yml para toda la solución con la opción Agregar > Compatibilidad con Container Orchestrator.... En el paso 4, exploraremos esta opción con mayor detalle.

### Uso de una imagen Docker oficial de .NET existente

Por lo general, crea una imagen personalizada para su contenedor sobre una imagen base que obtiene de un repositorio oficial como [el registro de Docker Hub](#). Eso es precisamente lo que sucede bajo las sábanas cuando habilita la compatibilidad con Docker en Visual Studio. Su Dockerfile utilizará una imagen dotnet/core/aspnet existente .

Anteriormente explicamos qué imágenes y repositorios de Docker puede usar, según el marco y el sistema operativo que haya elegido. Por ejemplo, si desea usar ASP.NET Core (Linux o Windows), la imagen que debe usar es mcr.microsoft.com/dotnet/aspnet:6.0. Por lo tanto, solo necesita especificar qué imagen base de Docker utilizará para su contenedor. Lo hace agregando FROM mcr.microsoft.com/dotnet/aspnet:6.0 a su Dockerfile. Visual Studio lo realizará automáticamente, pero si tuviera que actualizar la versión, actualice este valor.

El uso de un repositorio de imágenes .NET oficial de Docker Hub con un número de versión garantiza que las mismas funciones de idioma estén disponibles en todas las máquinas (incluido el desarrollo, las pruebas y la producción).

El siguiente ejemplo muestra un Dockerfile de muestra para un contenedor de ASP.NET Core.

```
DESDE mcr.microsoft.com/dotnet/aspnet:6.0 Fuente ARG
WORKDIR /aplicación
EXPOSER 80
COPIAR ${fuente:-obj/Docker/publish}
PUNTO DE ENTRADA ["dotnet", "MySingleContainerWebApp.dll"]
```

En este caso, la imagen se basa en la versión 6.0 de la imagen oficial de ASP.NET Core Docker (multiarquitectura para Linux y Windows). Esta es la configuración de mcr.microsoft.com/dotnet/aspnet:6.0. (Para obtener más información sobre esta imagen base, consulte la [página Imagen de Docker de ASP.NET Core](#)). En Dockerfile, también debe indicarle a Docker que escuche en el puerto TCP que usará en tiempo de ejecución (en este caso, el puerto 80, como configurado con el ajuste EXPOSE).

Puede especificar ajustes de configuración adicionales en el Dockerfile, según el idioma y el marco que esté utilizando. Por ejemplo, la línea ENTRYPOINT con ["dotnet", "MySingleContainerWebApp.dll"] le indica a Docker que ejecute una aplicación .NET. Si usa el SDK y la CLI de .NET (CLI de dotnet) para compilar y ejecutar la aplicación .NET, esta configuración sería diferente. La conclusión es que la línea ENTRYPOINT y otras configuraciones serán diferentes según el idioma y la plataforma que elija para su aplicación.

### Recursos adicionales

- Creación de imágenes de Docker para aplicaciones .NET  
6 <https://docs.microsoft.com/dotnet/core/docker/building-net-docker-images>

- Construye tu propia imagen. En la documentación oficial de Docker. <https://docs.docker.com/engine/tutorials/dockerimages/>
  - Mantenerse actualizado con las imágenes de contenedor .NET  
<https://devblogs.microsoft.com/dotnet/staying-up-to-date-with-net-container-images/>
  - Uso de .NET y Docker juntos: actualización de DockerCon 2018 <https://devblogs.microsoft.com/dotnet/using-net-and-docker-together-dockercon-2018-update/>
- 

### Uso de repositorios de imágenes de varias arquitecturas

Un solo repositorio puede contener variantes de plataforma, como una imagen de Linux y una imagen de Windows. Esta función permite a proveedores como Microsoft (creadores de imágenes base) crear un solo repositorio para cubrir múltiples plataformas (es decir, Linux y Windows). Por ejemplo, el **repositorio .NET** disponible en el registro de Docker Hub brinda soporte para Linux y Windows Nano Server usando el mismo nombre de repositorio.

Si especifica una etiqueta, apuntando a una plataforma que es explícita como en los siguientes casos:

- `mcr.microsoft.com/dotnet/aspnet:6.0-bullseye-slim` Destinos: .NET 6  
runtime-only en Linux
- `mcr.microsoft.com/dotnet/aspnet:6.0-nanoserver-ltsc2022` Destinos: .NET 6  
runtime-only en Windows Nano Server

Pero, si especifica el mismo nombre de imagen, incluso con la misma etiqueta, las imágenes de varias arquitecturas (como la **imagen aspnet**) usarán la versión de Linux o Windows según el sistema operativo del host Docker que esté implementando, como se muestra a continuación. ejemplo:

- `mcr.microsoft.com/dotnet/aspnet:6.0 Multi-arch: .NET`  
6 runtime-only en Linux o Windows Nano Server según el sistema operativo del host Docker

De esta manera, cuando extraiga una imagen de un host de Windows, extraerá la variante de Windows, y extraer el mismo nombre de imagen de un host de Linux extraerá la variante de Linux.

### Compilaciones de varias etapas en Dockerfile

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

Comienza con una imagen base que establece el contexto inicial, es como el sistema de archivos de inicio, que se encuentra en la parte superior del sistema operativo host. No es un sistema operativo, pero puede pensar en él como "el" sistema operativo dentro del contenedor.

La ejecución de cada línea de comando crea una nueva capa en el sistema de archivos con los cambios de la anterior, de modo que, cuando se combinan, producen 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 volverse muy grandes si tienen que incluir, por ejemplo, el SDK necesario para crear y publicar una aplicación.

Aquí es donde las compilaciones de varias etapas entran en la trama (desde Docker 17.05 y versiones posteriores) para hacer su magia.

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

En resumen, las compilaciones de múltiples etapas permiten dividir la creación en diferentes "fases" y luego ensamblar la imagen final tomando solo los directorios relevantes de las etapas intermedias. La estrategia general para usar esta característica es:

1. Use una imagen SDK base (no importa cuán grande sea), con todo lo necesario para compilar y publicar la aplicación a una carpeta y luego
2. Use una imagen base, pequeña y solo para tiempo de ejecución, y copie la carpeta de publicación de la etapa anterior para producir una imagen final pequeña.

Probablemente, la mejor manera de comprender las etapas múltiples es revisar un Dockerfile en detalle, línea por línea, así que comenzemos con el Dockerfile inicial creado por Visual Studio al agregar la compatibilidad con Docker a un proyecto y analizaremos algunas optimizaciones más adelante.

El Dockerfile inicial podría verse así:

```

1 DESDE mcr.microsoft.com/dotnet/aspnet:6.0 COMO base 2
WORKDIR /app 3 EXPOSE 80

4 5 DESDE mcr.microsoft.com/dotnet/sdk:6.0 COMO compilación 6
WORKDIR /src 7 COPY src/Services/Catalog/Catalog.API/
Catalog.API.csproj ...
8 COPIAR src/BuildingBlocks/HealthChecks/src/Microsoft.AspNetCore.HealthChecks ...
9 COPIAR src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions.HealthChecks ...
10 COPIAR src/BuildingBlocks/EventBus/IntegrationEventLogEF/ ...
11 COPIAR src/BuildingBlocks/EventBus/EventBus/EventBus.csproj ...
12 COPIAR src/BuildingBlocks/EventBus/EventBusRabbitMQ/EventBusRabbitMQ.csproj ...
13 COPIAR src/BuildingBlocks/EventBus/EventBusServiceBus/EventBusServiceBus.csproj ...
14 COPIA src/BuildingBlocks/WebHostCustomization/WebHost.Customization ...
15 COPIAR src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions ...
16 COPIAR src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions ...
17 EJECUTAR dotnet restore src/Services/Catalog/Catalog.API/Catalog.API.csproj 18 COPIAR ...

19 WORKDIR /src/src/Services/Catalog/Catalog.API 20 EJECUTAR
dotnet compilar Catalog.API.csproj -c Release -o /app 21 22 FROM compilar
COMO publicar 23 EJECUTAR dotnet publicar Catalog.API.csproj -c Release -o /
aplicación 24

25 DESDE base COMO final
26 WORKDIR /app 27 COPY
--from=publish /app .
28 PUNTO DE ENTRADA ["dotnet", " Catálogo.API.dll"]

```

Y estos son los detalles, línea por línea:

- Línea #1: Comience una etapa con una imagen base "pequeña" solo en tiempo de ejecución, llámela base como referencia.
- Línea #2: Cree el directorio /app en la imagen.
- Línea #3: Expone el puerto 80.

- Línea #5: Comienza una nueva etapa con la imagen “grande” para construir/publicar. Llámalo build como referencia.
- Línea #6: Crear directorio /src en la imagen.
- Línea #7: Hasta la línea 16, copie los archivos de proyecto .csproj a los que se hace referencia para poder restaurar los paquetes más tarde.
- Línea #17: paquetes de restauración para el proyecto Catalog.API y los proyectos a los que se hace referencia.
- Línea #18: Copie todo el árbol de directorios para la solución (excepto los archivos/directorios incluidos en el archivo .dockerignore ) al directorio /src en la imagen.
- Línea #19: Cambie la carpeta actual al proyecto Catalog.API .
- Línea #20: Compile el proyecto (y otras dependencias del proyecto) y envíelo al directorio /app en la imagen.
- Línea #22: comienza una nueva etapa que continúa desde la construcción. Llámalo publicar como referencia.
- Línea #23: Publique el proyecto (y las dependencias) y envíelo al directorio /app en la imagen.
- Línea #25: Comience una nueva etapa continuando desde la base y llámela final.
- Línea #26: Cambie el directorio actual a /app.
- Línea n.º 27: Copie el directorio /app de la etapa de publicación al directorio actual.
- Línea n.º 28: defina el comando que se ejecutará cuando se inicie el contenedor.

Ahora exploremos algunas optimizaciones para mejorar el rendimiento de todo el proceso que, en el caso de eShopOnContainers, significa alrededor de 22 minutos o más para construir la solución completa en contenedores de Linux.

Aprovechará la función de caché de capa de Docker, que es bastante simple: si la imagen base y los comandos son los mismos que algunos ejecutados anteriormente, puede usar la capa resultante sin necesidad de ejecutar los comandos, ahorrando así algo de tiempo. .

Entonces, concentrémonos en la **etapa de construcción** , las líneas 5-6 son en su mayoría iguales, pero las líneas 7-17 son diferentes para cada servicio de eShopOnContainers, por lo que tienen que ejecutarse cada vez, sin embargo, si cambió las líneas 7-16 a:

#### **COPiar\_**

Entonces sería igual para todos los servicios, copiaría toda la solución y crearía una capa más grande pero:

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

La siguiente optimización significativa implica el comando de restauración ejecutado en la línea 17, que también es diferente para cada servicio de eShopOnContainers. Si cambia esa línea a solo:

**EJECUTAR restauración dotnet**

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

Sin embargo, dotnet restore solo se ejecuta si hay un solo proyecto o archivo de solución en la carpeta, por lo que lograr esto es un poco más complicado y la forma de resolverlo, sin entrar en demasiados detalles, es la siguiente:

1. Agregue las siguientes líneas a .dockerignore:

- \*.sln, para ignorar todos los archivos de solución en el árbol de carpetas principal
- eShopOnContainers-ServicesAndWebApps.sln, para incluir solo este archivo de solución.

2. Incluya el argumento /ignoreprojectextensions:.dcproj para dotnet restore, de modo que también ignore el proyecto docker-compose y solo restaure los paquetes para la solución eShopOnContainers-ServicesAndWebApps.

Para la optimización final, sucede que la línea 20 es redundante, ya que la línea 23 también crea la aplicación y viene, en esencia, justo después de la línea 20, por lo que ahí va otro comando que consume mucho tiempo.

El archivo resultante es entonces:

```
1 DESDE mcr.microsoft.com/dotnet/aspnet:6.0 COMO base 2
WORKDIR /app 3 EXPOSE 80

4 5 DESDE mcr.microsoft.com/dotnet/sdk:6.0 COMO publicar 6
WORKDIR /src 7 COPY ..

8 EJECUTAR dotnet restore /ignoreprojectextensions:.dcproj 9 WORKDIR /
src/src/Services/Catalog/Catalog.API 10 EJECUTAR dotnet publicar
Catalog.API.csproj -c Release -o /app 11 12 DESDE base AS final 13 WORKDIR /
app 14 COPIAR --desde=publicar /aplicación.

15 PUNTO DE ENTRADA ["dotnet", " Catalog.API.dll"]
```

## Creando tu imagen base desde cero

Puede crear su propia imagen base de Docker desde cero. Este escenario no se recomienda para alguien que está comenzando con Docker, pero si desea configurar los bits específicos de su propia imagen base, puede hacerlo.

## Recursos adicionales

- Imágenes .NET Core de varias arquitecturas. <https://github.com/dotnet/announcements/issues/14>
- Crea una imagen base. Documentación oficial de Docker. <https://docs.docker.com/develop/develop-images/baseimages/>



## Paso 3. Cree sus imágenes Docker personalizadas e incoruste su aplicación o servicio en ellas

Para cada servicio de su aplicación, debe crear una imagen relacionada. Si su aplicación se compone de un solo servicio o aplicación web, solo necesita una sola imagen.

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

Usted, como desarrollador, necesita desarrollar y probar localmente hasta que envíe una característica completa o cambie su sistema de control de fuente (por ejemplo, a GitHub). Esto significa que debe crear las imágenes de Docker e implementar contenedores en un host de Docker local (VM de Windows o Linux) y ejecutar, probar y depurar en esos contenedores locales.

Para crear una imagen personalizada en su entorno local mediante la CLI de Docker y su Dockerfile, puede usar el comando de compilación de docker, 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
5c90d4a2dia8: Downloading [=====] 18.34 MB/51.35 MB
ab30cc3719b1: Downloading [=====] 18.48 MB/18.55 MB
c6072700a242: Downloading [=====] 18.34 MB/42.53 MB
121d7eef6c20: Waiting
eb57cf4f29ee: Waiting
b2c5ae2d325b: Waiting
```

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

Opcionalmente, en lugar de ejecutar directamente la compilación de docker desde la carpeta del proyecto, primero puede generar una carpeta implementable con las bibliotecas y archivos binarios de .NET requeridos ejecutando dotnet Publish y luego use el comando de compilación de docker .

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

Cuando una aplicación está compuesta por varios contenedores (es decir, es una aplicación de varios contenedores), también puede usar el comando docker-compose up --build para crear todas las imágenes relacionadas con un solo comando usando los metadatos expuestos en los archivos docker-compose.yml relacionados.

Puede encontrar las imágenes existentes en su repositorio local usando el comando de imágenes acopiables, como se muestra en la Figura 5-6.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
cesardl/netcore-webapi-microservice-docker	first	384c4ac1809b	4 minutes ago	579.8 MB
microsoft/dotnet	latest	49aaF5daa850	30 hours ago	548.6 MB
ubuntu	latest	cf62323fa025	5 days ago	125 MB
hello-world	latest	c54a2cc56ccb	12 days ago	1.848 kB

Figura 5-6. Visualización de imágenes existentes con el comando de imágenes acopiables

## Crear imágenes de Docker con Visual Studio

Cuando usa Visual Studio para crear un proyecto compatible con Docker, no crea una imagen explícitamente. En su lugar, la imagen se crea para usted cuando presiona F5 (o Ctrl-F5) para ejecutar la aplicación o el servicio dockerizado. Este paso es automático en Visual Studio y no verá que sucede, pero es importante que sepa lo que sucede debajo.



## Paso 4. Defina sus servicios en docker-compose.yml al crear una aplicación Docker de varios contenedores

El archivo [docker-compose.yml](#) le permite definir un conjunto de servicios relacionados que se implementarán como una aplicación compuesta con comandos de implementación. También configura sus relaciones de dependencia y la configuración del tiempo de ejecución.

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

```

versión: '3.4'

servicios:

  webmvc:
    imagen: eshop/
    entorno web:
      - CatalogUrl=http://catalog-api
      - OrderingUrl=http://ordering-api
    puertos:
      - "80:80"
    depende_de:
      - catalog-api
      - ordering-api

  catalog-api:
    imagen: eshop/catalog-api
    entorno:
      - ConnectionString=Server=sqldata;Port=1433;Database=CatalogDB;...
    puertos:
      - "81:80"

    depende_de:
      - sqldata

  ordering-api:
    imagen: eshop/ordering-api
    entorno:
      - ConnectionString=Server=sqldata;Database=OrderingDb;...
    puertos:
      - "82:80"
    extra_hosts:
      - "CESARDLBOOKVHD:10.0.75.1"
    depende_de:

```

```

- sqldata

sqldata:
  imagen: mcr.microsoft.com/mssql/server :último entorno:

  - SA_PASSWORD=Contraseña@palabra
  - ACCEPT_EULA=
Puertos Y: - "5433:1433"

```

Este archivo docker-compose.yml es una versión simplificada y fusionada. Contiene datos de configuración estáticos para cada contenedor (como el nombre de la imagen personalizada), que siempre se requiere, e información de configuración que puede depender del entorno de implementación, como la cadena de conexión. En secciones posteriores, aprenderá a dividir la configuración de docker-compose.yml en varios archivos de composición de docker y anular los valores según el entorno y el tipo de ejecución (depuración o lanzamiento).

El ejemplo del archivo docker-compose.yml define cuatro servicios: el servicio webmvc (una aplicación web), dos microservicios (ordering-api y basket-api) y un contenedor de origen de datos, sqldata, basado en SQL Server para Linux que se ejecuta como un envase. Cada servicio se implementará como un contenedor, por lo que se requiere una imagen de Docker para cada uno.

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

- Utiliza una imagen eshop/web:latest prediseñada . Sin embargo, también puede configurar la imagen para que se cree como parte de la ejecución de docker-compose con una configuración adicional basada en una sección build: en el archivo docker-compose.
- Inicializa dos variables de entorno (CatalogUrl y OrderingUrl).
- Reenvía el puerto 80 expuesto en el contenedor al puerto externo 80 en la máquina host.
- Vincula la aplicación web al catálogo y al servicio de pedidos con la configuración depend\_on. Esto hace que el servicio espere hasta que se inicien esos servicios.

Revisaremos el archivo docker-compose.yml en una sección posterior cuando cubramos cómo implementar microservicios y aplicaciones de contenedores múltiples.

## Trabajar con docker-compose.yml en Visual Studio 2022

Además de agregar un Dockerfile a un proyecto, como mencionamos antes, Visual Studio 2017 (desde la versión 15.8 en adelante) puede agregar soporte de orquestador para Docker Compose a una solución.

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

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

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

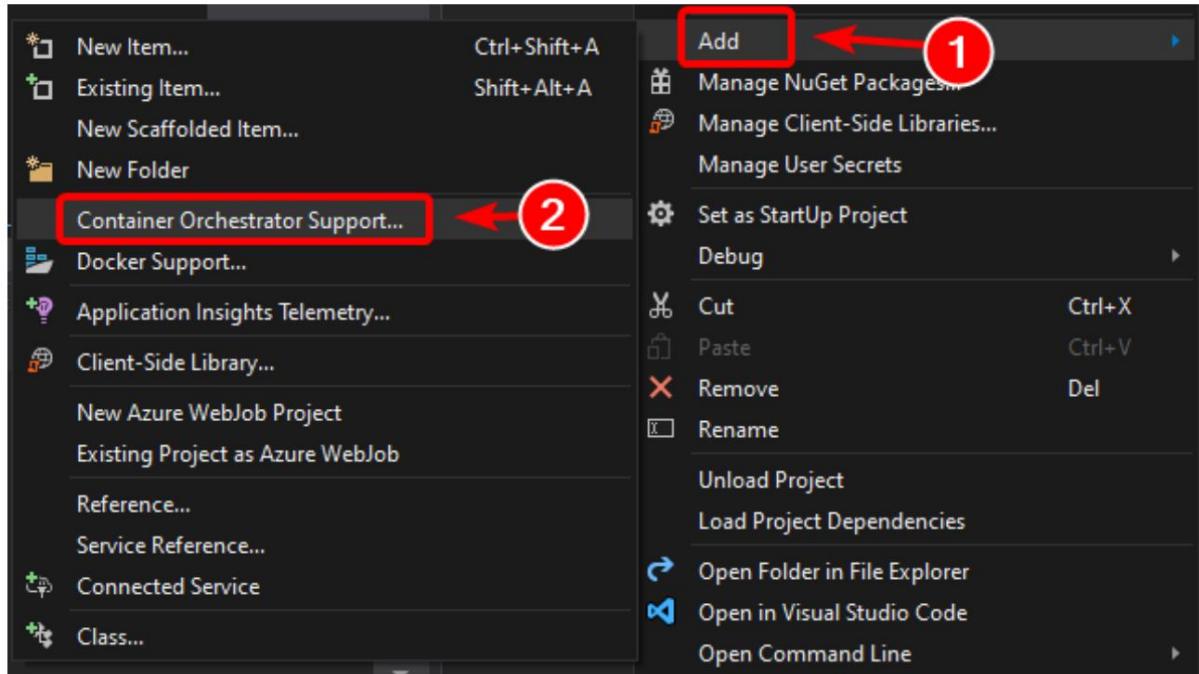


Figura 5-7. Adición de compatibilidad con Docker en Visual Studio 2022 haciendo clic con el botón derecho en un proyecto de ASP.NET Core

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

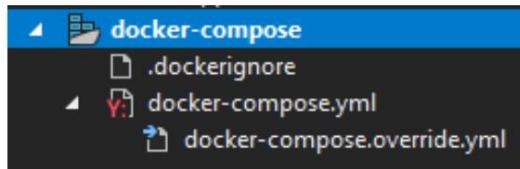


Figura 5-8. El nodo de árbol docker-compose agregado en Visual Studio 2022 Solution Explorer

Puede implementar una aplicación de varios contenedores con un único archivo docker-compose.yml mediante el comando docker-compose up . Sin embargo, Visual Studio agrega un grupo de ellos para que pueda anular los valores según el entorno (desarrollo o producción) y el tipo de ejecución (lanzamiento o depuración). Esta capacidad se explicará en secciones posteriores.



## Paso 5. Cree y ejecute su aplicación Docker

Si su aplicación solo tiene un único contenedor, puede ejecutarlo implementándolo en su host Docker (VM o servidor físico). Sin embargo, si su aplicación contiene varios servicios, puede implementarla como una aplicación compuesta, ya sea usando un solo comando CLI (docker-compose up) o con Visual Studio, que usará ese comando en secreto. Veamos las diferentes opciones.

## Opción A: ejecutar una aplicación de un solo contenedor

### Uso de la CLI de Docker

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

```
docker ejecutar -t -d -p 80:5000 cesardl/netcore-webapi-microservice-docker:primero
```

El comando anterior creará una nueva instancia de contenedor a partir de la imagen especificada, cada vez que se ejecute.

Puede usar el parámetro `--name` para dar un nombre al contenedor y luego usar `docker start {name}` (o usar el ID 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 Docker con el comando `docker run`

En este caso, el comando vincula el puerto interno 5000 del contenedor al puerto 80 de la máquina host. Esto significa que el host escucha en el puerto 80 y reenvía al puerto 5000 en el contenedor.

El hash que se muestra es el ID del contenedor y también se le asigna un nombre legible aleatorio si no se usa la opción `--name`.

### Uso de Visual Studio

Si no ha agregado la compatibilidad con el orquestador de contenedores, también puede ejecutar una sola aplicación de contenedor en Visual Studio presionando Ctrl-F5 y también puede usar F5 para depurar la aplicación dentro del contenedor. El contenedor se ejecuta localmente mediante `docker run`.

## Opción B: ejecutar una aplicación de varios contenedores

En la mayoría de los escenarios empresariales, una aplicación Docker estará compuesta por varios servicios, lo que significa que debe ejecutar una aplicación de varios contenedores, como se muestra en la Figura 5-10.

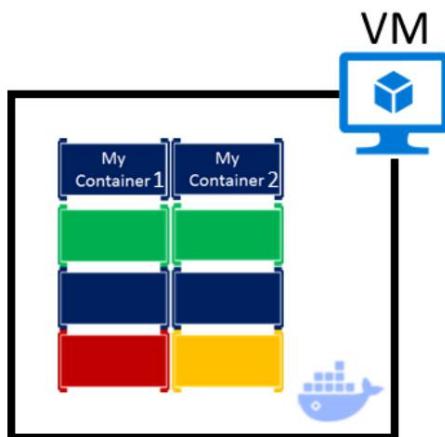


Figura 5-10. Máquina virtual con contenedores Docker implementados

### Uso de la CLI de Docker

Para ejecutar una aplicación de varios contenedores con la CLI de Docker, utilice el comando docker-compose up .

Este comando usa el archivo docker-compose.yml que tiene en el nivel de solución para implementar una aplicación de varios contenedores. La Figura 5-11 muestra los resultados al ejecutar el comando desde el directorio principal de su 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 de ejemplo al ejecutar el comando docker-compose up

Después de que se ejecuta el comando docker-compose up, la aplicación y sus contenedores relacionados se implementan en su host Docker, como se muestra en la Figura 5-10.

### Uso de Visual Studio

Ejecutar una aplicación de varios contenedores con Visual Studio 2019 no puede ser más simple. Simplemente presione Ctrl-F5 para ejecutar o F5 para depurar, como de costumbre, configurando el proyecto docker-compose como el proyecto de inicio. Visual Studio maneja toda la configuración necesaria, por lo que puede crear puntos de interrupción como de costumbre y depurar lo que finalmente se convierten en procesos independientes que se ejecutan en "servidores remotos", con el depurador ya conectado, así de fácil.

Como se mencionó anteriormente, cada vez que agrega compatibilidad con la solución Docker a un proyecto dentro de una solución, ese proyecto se configura en el archivo docker-compose.yml global (nivel de solución), que le permite ejecutar o depurar toda la solución a la vez. Visual Studio iniciará un contenedor para cada proyecto que tenga habilitada la compatibilidad con la solución Docker y realizará todos los pasos internos por usted (publicación de dotnet, compilación de docker, etc.).

Si desea echar un vistazo a todo el trabajo pesado, eche un vistazo al archivo:

```
{carpeta raíz de la solución}\obj\Docker\docker-compose.vs.debug.g.yml
```

El punto importante aquí es que, como se muestra en la Figura 5-12, en Visual Studio 2019 hay un comando Docker adicional para la acción de la tecla F5. Esta opción le permite ejecutar o depurar una aplicación de varios contenedores ejecutando todos los contenedores definidos en los archivos docker-compose.yml en el nivel de solución. La capacidad de depurar soluciones de varios contenedores significa que puede establecer varios puntos de interrupción, cada punto de interrupción en un proyecto (contenedor) diferente y, mientras realiza la depuración desde Visual Studio, se detendrá en los puntos de interrupción definidos en diferentes proyectos y que se ejecutan en diferentes contenedores.

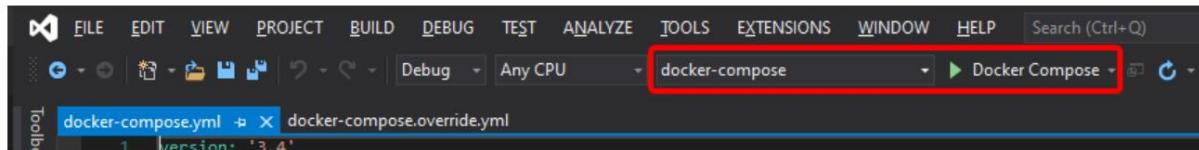


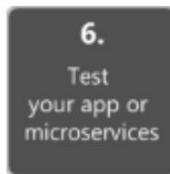
Figura 5-12. Ejecución de aplicaciones de varios contenedores en Visual Studio 2022

## Recursos adicionales

- Implemente un contenedor ASP.NET en un host Docker remoto  
<https://docs.microsoft.com/visualstudio/containers/hosting-web-apps-in-docker>

## Una nota sobre las pruebas y la implementación con orquestadores

Los comandos docker-compose up y docker run (o ejecutar y depurar los contenedores en Visual Studio) son adecuados para probar contenedores en su entorno de desarrollo. Pero no debe usar este enfoque para implementaciones de producción, donde debe apuntar a orquestadores como **Kubernetes** o **Service Fabric**. Si usa **Kubernetes**, debe usar pods para organizar contenedores y servicios para conectarlos en red. También usa implementaciones para organizar la creación y modificación de pods.



## Paso 6. Pruebe su aplicación Docker usando su host Docker local

Este paso variará dependiendo de lo que esté haciendo su aplicación. En una aplicación web .NET simple que se implementa como un único contenedor o servicio, puede acceder al servicio abriendo un navegador en el host de Docker y navegando a ese sitio, como se muestra en la Figura 5-13. (Si la configuración en el Dockerfile asigna el contenedor a un puerto en el host que no sea 80, incluya el puerto del host en la URL).



Figura 5-13. Ejemplo de prueba de su aplicación Docker localmente usando localhost

Si localhost no apunta a la IP del host de Docker (de manera predeterminada, al usar Docker CE, debería), para navegar a su servicio, use la dirección IP de la tarjeta de red de su máquina.

Esta URL en el navegador usa el puerto 80 para el ejemplo de contenedor particular que se está discutiendo. Sin embargo, internamente las solicitudes están siendo redirigidas al puerto 5000, porque así fue como se implementó con el comando docker run, como se explicó en un paso anterior.

También puede probar la aplicación usando curl desde la terminal, como se muestra en la Figura 5-14. En una instalación de Docker en Windows, la IP predeterminada del host de Docker siempre es 10.0.75.1 además de la dirección IP real de su máquina.

```
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 su aplicación Docker localmente usando curl

## Prueba y depuración de contenedores con Visual Studio 2022

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

## Prueba y depuración sin Visual Studio

Si está desarrollando con el enfoque de editor/CLI, la depuración de contenedores es más difícil y probablemente querrá depurar generando seguimientos.

## Recursos adicionales

- Inicio rápido: Docker en Visual Studio. <https://docs.microsoft.com/visualstudio/containers/container-tools>
- Depuración de aplicaciones en un contenedor Docker local <https://docs.microsoft.com/visualstudio/containers/edit-and-refresh>

## Flujo de trabajo simplificado al desarrollar contenedores con Visual Studio

Efectivamente, el flujo de trabajo cuando usa Visual Studio es mucho más simple que si usa el enfoque del editor/CLI. La mayoría de los pasos requeridos por Docker relacionados con los archivos Dockerfile y docker-compose.yml están ocultos o simplificados por Visual Studio, como se muestra en la Figura 5-15.

# VS development workflow for Docker apps

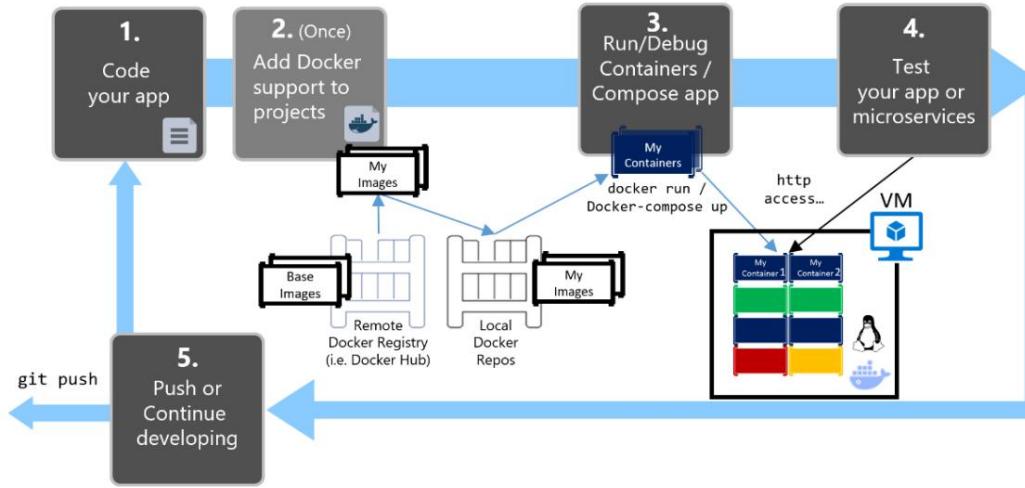


Figura 5-15. Flujo de trabajo simplificado al desarrollar con Visual Studio

Además, debe realizar el paso 2 (agregar compatibilidad con Docker a sus proyectos) solo una vez. Por lo tanto, el flujo de trabajo es similar a sus tareas de desarrollo habituales cuando utiliza .NET para cualquier otro desarrollo.

Necesita saber qué sucede debajo de las sábanas (el proceso de creación de imágenes, qué imágenes base está usando, implementación de contenedores, etc.) y, a veces, también necesitará editar el archivo Dockerfile o docker-compose.yml para personalizar comportamientos. Pero la mayor parte del trabajo se simplifica enormemente con el uso de Visual Studio, lo que lo hace mucho más productivo.

## Recursos adicionales

- Depurar aplicaciones en un contenedor Docker local  
<https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T111>

## Uso de comandos de PowerShell en un Dockerfile para configurar Windows Contenedores

[Los contenedores de Windows](#) le permiten convertir sus 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 los comandos de PowerShell en Dockerfile, como se muestra en el siguiente ejemplo:

```

DESDE mcr.microsoft.com/windows/servercore
ETIQUETA Descripción="IIS" Proveedor="Microsoft" Versión="10"
EJECUTAR powershell -Command Add-WindowsFeature Web-Server
CMD [ "ping", "localhost", "-t" ]
  
```

En este caso, usamos una imagen base de Windows Server Core (la configuración DESDE) e instalamos IIS con un comando de PowerShell (la configuración EJECUTAR). De manera similar, también puede usar los comandos de PowerShell para configurar componentes adicionales como ASP.NET 4.x, .NET Framework 4.6 o cualquier otro software de Windows. Por ejemplo, el siguiente comando en un Dockerfile configura ASP.NET 4.5:

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

## Recursos adicionales

- aspnet-docker/Dockerfile. Ejemplos de comandos de PowerShell para ejecutar desde dockerfiles para incluir características de Windows. <https://github.com/Microsoft/aspnet-docker/blob/master/4.7.1-windowsservercore ltsc2016/runtime/Dockerfile>
-

# Diseño y desarrollo de aplicaciones .NET basadas en microservicios y contenedores múltiples

El desarrollo de aplicaciones de microservicios en contenedores significa que está creando aplicaciones de varios contenedores. Sin embargo, una aplicación de varios contenedores también podría ser más simple, por ejemplo, una aplicación de tres niveles, y es posible que no se construya con una arquitectura de microservicio.

Anteriormente planteamos la pregunta "¿Es Docker necesario al construir una arquitectura de microservicio?" La respuesta es un claro no. Docker es un habilitador y puede brindar beneficios significativos, pero los contenedores y Docker no son un requisito estricto para los microservicios. Como ejemplo, podría crear una aplicación basada en microservicios con o sin Docker al usar Azure Service Fabric, que admite microservicios que se ejecutan como procesos simples o como contenedores de Docker.

Sin embargo, si sabe cómo diseñar y desarrollar una aplicación basada en microservicios que también se base en contenedores Docker, podrá diseñar y desarrollar cualquier otro modelo de aplicación más simple. Por ejemplo, puede diseñar una aplicación de tres niveles que también requiera un enfoque de varios contenedores. Por eso, y debido a que las arquitecturas de microservicios son una tendencia importante dentro del mundo de los contenedores, esta sección se centra en la implementación de una arquitectura de microservicios utilizando contenedores Docker.

## Diseñar una aplicación orientada a microservicios

Esta sección se centra en el desarrollo de una aplicación empresarial hipotética del lado del servidor.

### Especificaciones de la aplicación

La aplicación hipotética maneja las solicitudes mediante la ejecución de la lógica empresarial, el acceso a las bases de datos y la devolución de respuestas HTML, JSON o XML. Diremos que la aplicación debe ser compatible con varios clientes, incluidos los navegadores de escritorio que ejecutan aplicaciones de página única (SPA), aplicaciones web tradicionales, aplicaciones web móviles y aplicaciones móviles nativas. La aplicación también podría exponer una API para terceros.

consumir. También debería poder integrar sus microservicios o aplicaciones externas de forma asíncrona, por lo que ese enfoque ayudará a la resiliencia de los microservicios en caso de fallas parciales.

La aplicación constará de este tipo de componentes:

- Componentes de la presentación. Estos componentes son responsables de manejar la interfaz de usuario y consumir servicios remotos.
- Dominio o lógica de negocio. Este componente es la lógica de dominio de la aplicación.
- Lógica de acceso a la base de datos. Este componente consta de componentes de acceso a datos responsables de acceder a bases de datos (SQL o NoSQL).
- Lógica de integración de aplicaciones. Este componente incluye un canal de mensajería, basado en intermediarios de mensajes.

La aplicación requerirá una alta escalabilidad, al tiempo que permitirá que sus subsistemas verticales se escalen de forma autónoma, porque ciertos subsistemas requerirán más escalabilidad que otros.

La aplicación debe poder implementarse en múltiples entornos de infraestructura (múltiples nubes públicas y en las instalaciones) e, idealmente, debe ser multiplataforma, capaz de pasar de Linux a Windows (o viceversa) fácilmente.

## Contexto del equipo de desarrollo

También asumimos lo siguiente sobre el proceso de desarrollo de la aplicación:

- Tiene varios equipos de desarrollo centrados en diferentes áreas comerciales de la aplicación.
- Los nuevos miembros del equipo deben volverse productivos rápidamente y la aplicación debe ser fácil de entender y modificar.
- La aplicación tendrá una evolución a largo plazo y reglas de negocio en constante cambio.
- Necesita una buena capacidad de mantenimiento a largo plazo, lo que significa tener agilidad al implementar nuevos cambios en el futuro y poder actualizar varios subsistemas con un impacto mínimo en los otros subsistemas.
- Desea practicar la integración continua y el despliegue continuo de la aplicación.
- Quiere aprovechar las tecnologías emergentes (marcos, lenguajes de programación, etc.) mientras desarrolla la aplicación. No desea realizar migraciones completas de la aplicación al pasar a nuevas tecnologías, ya que eso generaría costos elevados y afectaría la previsibilidad y la estabilidad de la aplicación.

## Elegir una arquitectura

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

En este enfoque, cada servicio (contenedor) implementa un conjunto de funciones cohesivas y estrechamente relacionadas. Por ejemplo, una aplicación puede constar de servicios como el servicio de catálogo, el servicio de pedidos, el servicio de cesta, el servicio de perfil de usuario, etc.

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

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

Cada microservicio tiene su propia base de datos, lo que le permite estar totalmente desvinculado de otros microservicios. Cuando es necesario, la coherencia entre las bases de datos de diferentes microservicios se logra utilizando eventos de integración de nivel de aplicación (a través de un bus de eventos lógicos), como se maneja en la Segregación de responsabilidad de comandos y consultas (CQRS). Por eso, las restricciones comerciales deben adoptar la eventual coherencia entre los múltiples microservicios y las bases de datos relacionadas.

### [eShopOnContainers: una aplicación de referencia para .NET y microservicios implementados mediante contenedores](#)

Para que pueda concentrarse en la arquitectura y las tecnologías en lugar de pensar en un dominio comercial hipotético que tal vez no conozca, hemos seleccionado un dominio comercial bien conocido, a saber, una aplicación simplificada de comercio electrónico (e-shop) que presenta un catálogo de productos, toma pedidos de los clientes, verifica el inventario y realiza otras funciones comerciales. El código fuente de esta aplicación basada en contenedores está disponible en el repositorio de GitHub de [eShopOnContainers](#).

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

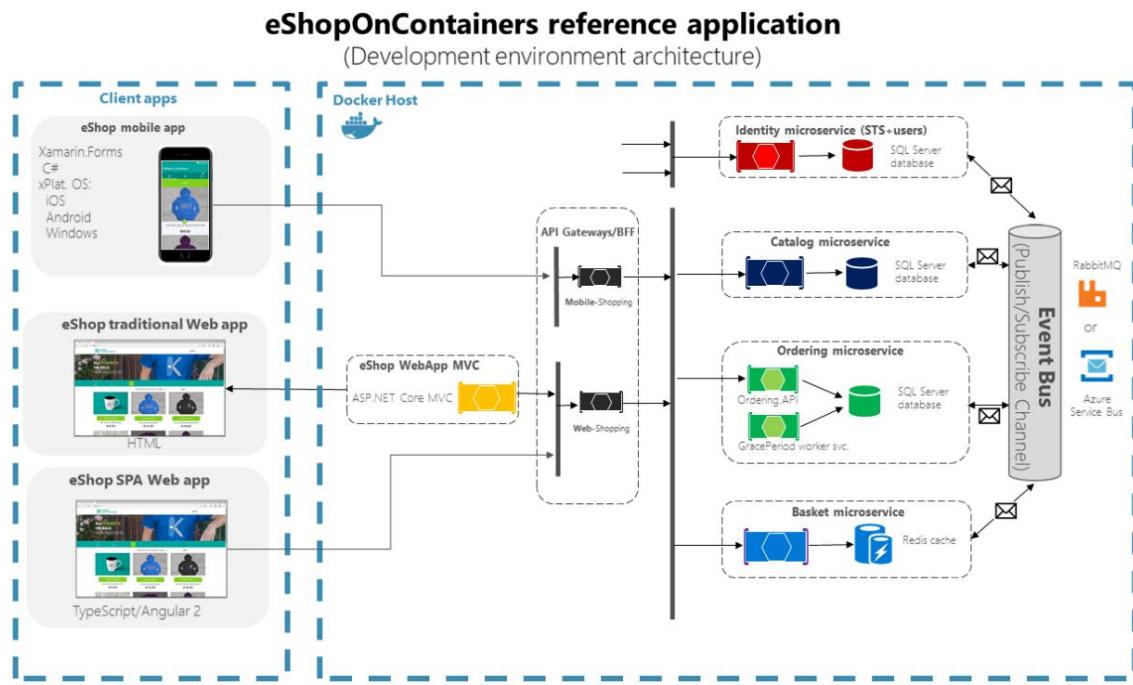


Figura 6-1. La arquitectura de aplicaciones de referencia de eShopOnContainers para el entorno de desarrollo

El diagrama anterior muestra que los clientes móviles y SPA se comunican con puntos finales de puerta de enlace de API únicos, que luego se comunican con microservicios. Los clientes web tradicionales se comunican con el microservicio MVC, que se comunica con los microservicios a través de la puerta de enlace API.

Entorno de acogida. En la Figura 6-1, ve varios contenedores implementados dentro de un solo host de Docker.

Ese sería el caso cuando se implementa en un único host de Docker con el comando docker-compose up. Sin embargo, si usa un orquestador o un clúster de contenedores, cada contenedor podría ejecutarse en un host (nodo) diferente y cualquier nodo podría ejecutar cualquier número de contenedores, como explicamos anteriormente en la sección de arquitectura.

Arquitectura de la comunicación. La aplicación eShopOnContainers utiliza dos tipos de comunicación, según el tipo de acción funcional (consultas versus actualizaciones y transacciones):

- Comunicación Http de cliente a microservicio a través de API Gateways. Este enfoque se usa para consultas y cuando se aceptan comandos de actualización o transaccionales de las aplicaciones cliente. El enfoque que usa API Gateways se explica en detalle en secciones posteriores.
- Comunicación asíncrona basada en eventos. Esta comunicación se produce a través de un bus de eventos para propagar actualizaciones a través de microservicios o para integrarse con aplicaciones externas. El bus de eventos se puede implementar con cualquier tecnología de infraestructura de agente de mensajería como RabbitMQ o mediante buses de servicio de nivel superior (nivel de abstracción) como Azure Service Bus, NServiceBus, MassTransit o Brighter.

La aplicación se despliega como un conjunto de microservicios en forma de contenedores. Las aplicaciones cliente pueden comunicarse con esos microservicios que se ejecutan como contenedores a través de las URL públicas publicadas por API Gateways.

## Soberanía de datos por microservicio

En la aplicación de ejemplo, cada microservicio posee su propia base de datos o fuente de datos, aunque todas las bases de datos de SQL Server se implementan como un único contenedor. Esta decisión de diseño se tomó solo para facilitar que un desarrollador obtenga el código de GitHub, lo clone y lo abra en Visual Studio o Visual Studio Code. O, alternativamente, facilita la compilación de imágenes de Docker personalizadas mediante la CLI de .NET y la CLI de Docker, y luego las implementa y ejecuta en un entorno de desarrollo de Docker. De cualquier manera, el uso de contenedores para fuentes de datos permite a los desarrolladores crear e implementar en cuestión de minutos sin tener que aprovisionar una base de datos externa o cualquier otra fuente de datos con dependencias estrictas de la infraestructura (en la nube o en las instalaciones).

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

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

## Recursos adicionales

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

## Beneficios de una solución basada en microservicios

Una solución basada en microservicios como esta tiene muchos beneficios:

Cada microservicio es relativamente pequeño, fácil de administrar y evolucionar. Específicamente:

- Es fácil de entender para un desarrollador y comenzar rápidamente con una buena productividad.
- Los contenedores se inician rápido, lo que hace que los desarrolladores sean más productivos.
- Un IDE como Visual Studio puede cargar proyectos más pequeños rápidamente, lo que hace que los desarrolladores sean productivos.
- Cada microservicio puede diseñarse, desarrollarse e implementarse independientemente de otros microservicios, lo que brinda agilidad porque es más fácil implementar nuevas versiones de microservicios con frecuencia.

Es posible escalar áreas individuales de la aplicación. Por ejemplo, es posible que sea necesario escalar horizontalmente el servicio de catálogo o el servicio de cesta, pero no el proceso de pedido. Una infraestructura de microservicios será mucho más eficiente con respecto a los recursos utilizados al escalar horizontalmente que una arquitectura monolítica.

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

Los problemas son más aislados. Si hay un problema en un servicio, solo ese servicio se ve afectado inicialmente (excepto cuando se usa un diseño incorrecto, con dependencias directas entre microservicios), y otros servicios pueden continuar manejando solicitudes. Por el contrario, un componente que funciona mal en un monolítico

la arquitectura de implementación puede hacer caer todo el sistema, especialmente cuando involucra recursos, como una fuga de memoria. Además, cuando se resuelve un problema en un microservicio, puede implementar solo el microservicio afectado sin afectar al resto de la aplicación.

Puede utilizar las últimas tecnologías. Debido a que puede comenzar a desarrollar servicios de forma independiente y ejecutarlos en paralelo (gracias a los contenedores y .NET), puede comenzar a usar las últimas tecnologías y marcos de manera conveniente en lugar de quedarse atascado en una pila o marco anterior para toda la aplicación.

## Desventajas de una solución basada en microservicios

Una solución basada en microservicios como esta también tiene algunos inconvenientes:

**Aplicación distribuida.** La distribución de la aplicación agrega complejidad para los desarrolladores cuando están diseñando y construyendo los servicios. Por ejemplo, los desarrolladores deben implementar la comunicación entre servicios mediante protocolos como HTTP o AMPQ, lo que agrega complejidad para las pruebas y el manejo de excepciones. También agrega latencia al sistema.

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

**Transacciones atómicas.** Las transacciones atómicas entre múltiples microservicios generalmente no son posibles. Los requisitos comerciales deben adoptar la coherencia eventual entre múltiples microservicios.

**Mayores necesidades de recursos globales** (memoria total, unidades y recursos de red para todos los servidores o hosts). En muchos casos, cuando reemplaza una aplicación monolítica con un enfoque de microservicios, la cantidad de recursos globales iniciales que necesita la nueva aplicación basada en microservicios será mayor que las necesidades de infraestructura de la aplicación monolítica original. Este enfoque se debe a que el mayor grado de granularidad y los servicios distribuidos requieren más recursos globales. Sin embargo, dado el bajo costo de los recursos en general y el beneficio de poder escalar ciertas áreas de la aplicación en comparación con los costos a largo plazo cuando se desarrollan aplicaciones monolíticas, el aumento en el uso de los recursos suele ser una buena compensación para los grandes proyectos a largo plazo. aplicaciones a plazo.

**Problemas con la comunicación directa entre el cliente y el microservicio.** Cuando la aplicación es grande, con docenas de microservicios, existen desafíos y limitaciones si la aplicación requiere comunicaciones directas entre el cliente y el microservicio. Un problema es una falta de coincidencia potencial entre las necesidades del cliente y las API expuestas por cada uno de los microservicios. En determinados casos, es posible que la aplicación del cliente deba realizar muchas solicitudes independientes para componer la interfaz de usuario, lo que puede ser ineficaz en Internet y poco práctico en una red móvil. Por lo 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 que algunos microservicios pueden estar usando protocolos que no son aptos para la Web. Un servicio puede usar un protocolo binario, mientras que otro servicio puede usar mensajería AMQP. Esos protocolos no son aptos para cortafuegos y es mejor usarlos internamente. Por lo general, una aplicación debe usar protocolos como HTTP y WebSockets para la comunicación fuera del firewall.

Otro inconveniente más con este enfoque directo de cliente a servicio es que dificulta la refactorización de los contratos para esos microservicios. Con el tiempo, es posible que los desarrolladores deseen cambiar la forma en que el sistema se divide en servicios. Por ejemplo, pueden fusionar dos servicios o dividir un servicio en dos o más servicios. Sin embargo, si los clientes se comunican directamente con los servicios, realizar este tipo de refactorización puede romper la compatibilidad con las aplicaciones de los clientes.

Como se mencionó en la sección de arquitectura, cuando diseñe y cree una aplicación compleja basada en microservicios, puede considerar el uso de varias puertas de enlace API detalladas en lugar del enfoque de comunicación directa más simple entre el cliente y el microservicio.

Partición de los microservicios. Por último, independientemente del enfoque que adopte para su arquitectura de microservicios, otro desafío es decidir cómo dividir una aplicación de un extremo a otro en múltiples microservicios. Como se indica en la sección de arquitectura de la guía, hay varias técnicas y enfoques que puede tomar. Básicamente, debe identificar las áreas de la aplicación que están desvinculadas de otras áreas y que tienen una cantidad baja de dependencias estrictas. En muchos casos, este enfoque está alineado con la partición de servicios por caso de uso. Por ejemplo, en nuestra aplicación de tienda electrónica, tenemos un servicio de pedidos que es responsable de toda la lógica comercial relacionada con el proceso de pedido. También contamos con el servicio de catálogo y el servicio de canasta que implementan otras capacidades. Idealmente, cada servicio debería tener solo un pequeño conjunto de responsabilidades. Este enfoque es similar al principio de responsabilidad única (SRP) aplicado a las clases, que establece que una clase solo debe tener una razón para cambiar. Pero en este caso, se trata de microservicios, por lo que el alcance será mayor que una sola clase. Sobre todo, un microservicio debe ser autónomo, de principio a fin, incluida la responsabilidad de sus propias fuentes de datos.

## Arquitectura externa versus interna y patrones de diseño

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

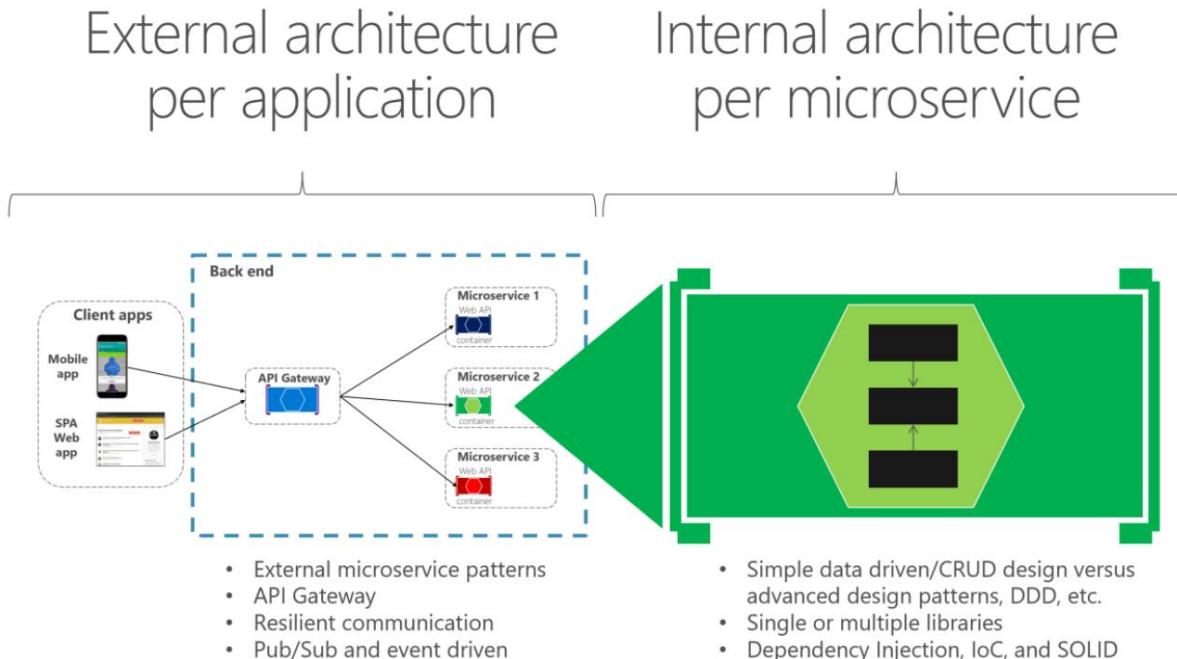


Figura 6-2. Arquitectura y diseño externo versus interno

Por ejemplo, en nuestra muestra de eShopOnContainers , los microservicios de catálogo, cesta y perfil de usuario son simples (básicamente, subsistemas CRUD). Por lo tanto, su arquitectura interna y diseño es sencillo. Sin embargo, es posible que tenga otros microservicios, como el microservicio de pedidos, que es más complejo y representa reglas comerciales en constante cambio con un alto grado de complejidad de dominio. En casos como estos, es posible que desee implementar patrones más avanzados dentro de un microservicio en particular, como los definidos con enfoques de diseño controlado por dominio (DDD), como lo estamos haciendo en el microservicio de pedidos eShopOnContainers . (Revisaremos estos patrones DDD en la sección posterior que explica la implementación del microservicio de pedidos eShopOnContainers ).

Otra razón para una tecnología diferente por microservicio podría ser la naturaleza de cada microservicio.

Por ejemplo, podría ser mejor usar un lenguaje de programación funcional como F#, o incluso un lenguaje como R si se dirige a dominios de inteligencia artificial 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 diferentes patrones de diseño. No todos los microservicios deben implementarse utilizando patrones DDD avanzados, porque eso sería sobrediseñarlos. Del mismo modo, los microservicios complejos con una lógica comercial en constante cambio no deben implementarse como componentes CRUD, o puede terminar con un código de baja calidad.

## El nuevo mundo: múltiples patrones arquitectónicos y microservicios políglotas

Hay muchos patrones arquitectónicos utilizados por los arquitectos y desarrolladores de software. Los siguientes son algunos (mezcla de estilos de arquitectura y patrones de arquitectura):

- CRUD simple, de un solo nivel, de una sola capa.

- [N-capa tradicional.](#)
- [Diseño basado en dominio N-capas.](#)
- [Arquitectura limpia](#) (como se usa con [eShopOnWeb](#))
- [Segregación de responsabilidad de comandos y consultas](#) (CQRS).
- [Arquitectura impulsada por eventos](#) (EDA).

También puede crear microservicios con muchas tecnologías y lenguajes, como ASP.NET Core Web API, NancyFx, ASP.NET Core SignalR (disponible con .NET Core 2 o posterior), F#, Node.js, Python, Java, C++, GoLang, y más.

El punto importante es que ningún patrón o estilo de arquitectura en particular, ni ninguna tecnología en particular, es adecuada para todas las situaciones. La Figura 6-3 muestra algunos enfoques y tecnologías (aunque no en ningún orden en particular) que podrían usarse en diferentes microservicios.

## The Multi-Architectural-Patterns and polyglot microservices world



Figura 6-3. Patrones multiarquitectónicos y el mundo de los microservicios políglotas

Los microservicios políglotas y con patrones multiarquitectónicos significan que puede mezclar y combinar lenguajes y tecnologías según las necesidades de cada microservicio y aún así hacer que se comuniquen entre sí. Como se muestra en la Figura 6-3, en las aplicaciones compuestas por muchos microservicios (Bounded Contexts en la terminología de diseño controlado por dominio, o simplemente "subsistemas" como microservicios autónomos), puede implementar cada microservicio de una manera diferente. Cada uno puede tener un patrón de arquitectura diferente y usar diferentes lenguajes y bases de datos según la naturaleza de la aplicación, los requisitos comerciales y las prioridades. En algunos casos, los microservicios pueden ser similares. Pero ese no suele ser el caso, porque el límite de contexto y los requisitos de cada subsistema suelen ser diferentes.

Por ejemplo, para una aplicación de mantenimiento CRUD simple, puede que no tenga sentido diseñar e implementar patrones DDD. Pero para su dominio principal o negocio principal, es posible que deba aplicar patrones más avanzados para abordar la complejidad comercial con reglas comerciales en constante cambio.

Especialmente cuando maneja aplicaciones grandes compuestas por múltiples subsistemas, no debe aplicar una única arquitectura de nivel superior basada en un único patrón de arquitectura. Por ejemplo, CQRS no debe aplicarse como una arquitectura de nivel superior para una aplicación completa, pero podría ser útil para un conjunto específico de servicios.

No existe una bala de plata o un patrón de arquitectura adecuado para cada caso dado. No puede tener "un patrón de arquitectura para gobernarlos a todos". Dependiendo de las prioridades de cada microservicio, debe elegir un enfoque diferente para cada uno, como se explica en las siguientes secciones.

## Creación de un microservicio CRUD basado en datos simple

Esta sección describe cómo crear un microservicio simple que realice operaciones de creación, lectura, actualización y eliminación (CRUD) en una fuente de datos.

### Diseño de un microservicio CRUD simple

Desde el punto de vista del diseño, este tipo de microservicio en contenedores es muy simple. Quizás el problema a resolver sea simple, o quizás la implementación sea solo una prueba de concepto.

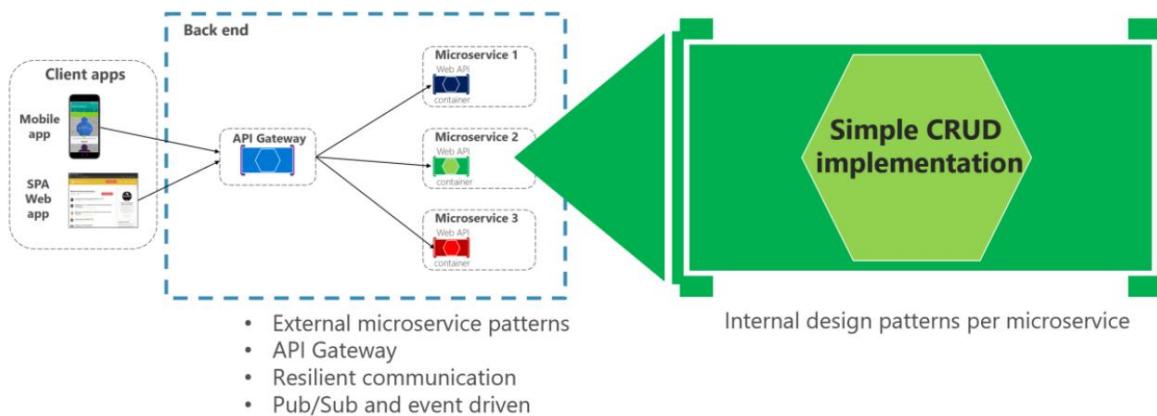


Figura 6-4. Diseño interno para microservicios CRUD simples

Un ejemplo de este tipo de servicio de unidad de datos simple es el microservicio de catálogo de la aplicación de muestra eShopOnContainers. Este tipo de servicio implementa toda su funcionalidad en un solo proyecto ASP.NET Core Web API que incluye clases para su modelo de datos, su lógica de negocios y su código de acceso a datos. También almacena sus datos relacionados en una base de datos que se ejecuta en SQL Server (como otro contenedor para propósitos de desarrollo/prueba), pero también podría ser cualquier host de SQL Server normal, como se muestra en la Figura 6-5.

## Data-Driven/CRUD microservice container

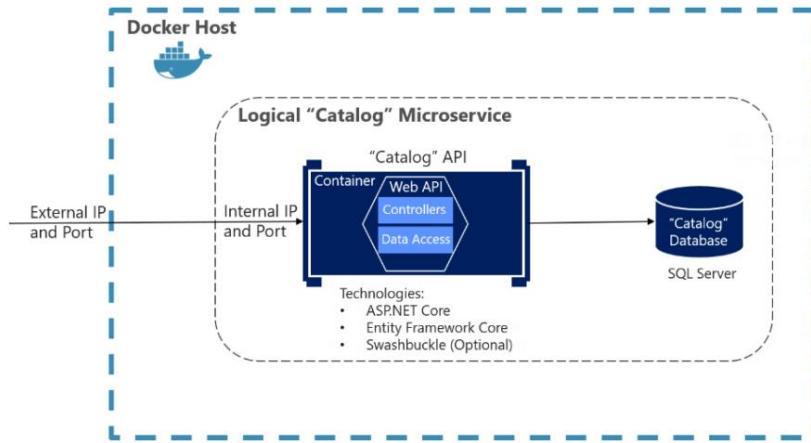


Figura 6-5. Diseño simple de microservicio basado en datos/CRUD

El diagrama anterior muestra el microservicio de catálogo lógico, que incluye su base de datos de catálogo, que puede estar o no en el mismo host de Docker. Tener la base de datos en el mismo host de Docker puede ser bueno para el desarrollo, pero no para la producción. Cuando desarrolla este tipo de servicio, solo necesita [ASP.NET Core](#) y una API de acceso a datos u ORM como [Entity Framework Core](#). También puede generar metadatos de [Swagger](#) automáticamente a través de [Swashbuckle](#) para proporcionar una descripción de lo que ofrece su servicio, como se explica en la siguiente sección.

Tenga en cuenta que ejecutar un servidor de base de datos como SQL Server dentro de un contenedor Docker es excelente para los entornos de desarrollo, ya que puede tener todas sus dependencias en funcionamiento sin necesidad de aprovisionar una base de datos en la nube o en las instalaciones. Este enfoque es conveniente cuando se ejecutan pruebas de integración. Sin embargo, para entornos de producción, no se recomienda ejecutar un servidor de base de datos en un contenedor, ya que normalmente no se obtiene una alta disponibilidad con ese enfoque. Para un entorno de producción en Azure, se recomienda usar Azure SQL DB o cualquier otra tecnología de base de datos que pueda proporcionar alta disponibilidad y alta escalabilidad. Por ejemplo, para un enfoque NoSQL, puede elegir CosmosDB.

Finalmente, al editar los archivos de metadatos Dockerfile y docker-compose.yml, puede configurar cómo se creará la imagen de este contenedor, qué imagen base usará, además de configuraciones de diseño como nombres internos y externos y puertos TCP.

### Implementación de un microservicio CRUD simple con ASP.NET Core

Para implementar un microservicio CRUD simple con .NET y Visual Studio, comience creando un proyecto de API web ASP.NET Core simple (ejecutándose en .NET para que pueda ejecutarse en un host Docker de Linux), como se muestra en la Figura 6-6.

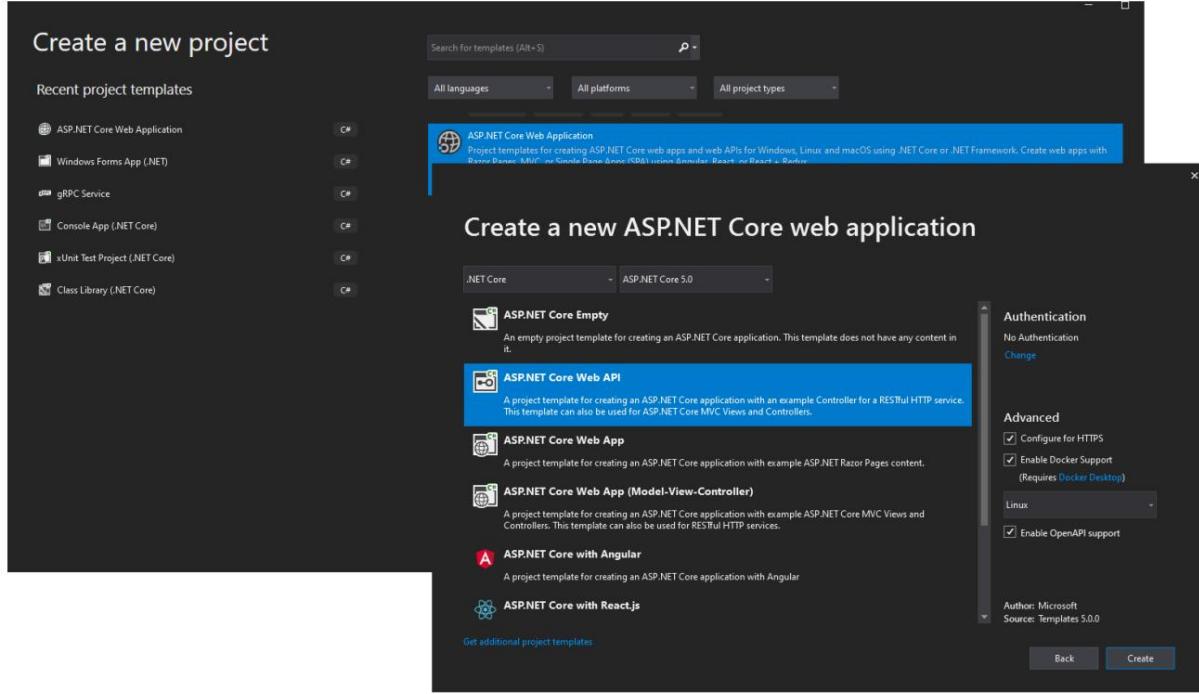


Figura 6-6. Creación de un proyecto de API web de ASP.NET Core en Visual Studio 2019

Para crear un proyecto de API web de ASP.NET Core, primero seleccione una aplicación web de ASP.NET Core y luego seleccione el tipo de API. Después de crear el proyecto, puede implementar sus controladores MVC como lo haría en cualquier otro proyecto de API web, utilizando Entity Framework API u otra API. En un nuevo proyecto de API web, puede ver que la única dependencia que tiene en ese microservicio está en ASP.NET Core. Internamente, dentro de la dependencia `Microsoft.AspNetCore.All`, hace referencia a Entity Framework y muchos otros paquetes .NET NuGet, como se muestra en la figura 6-7.

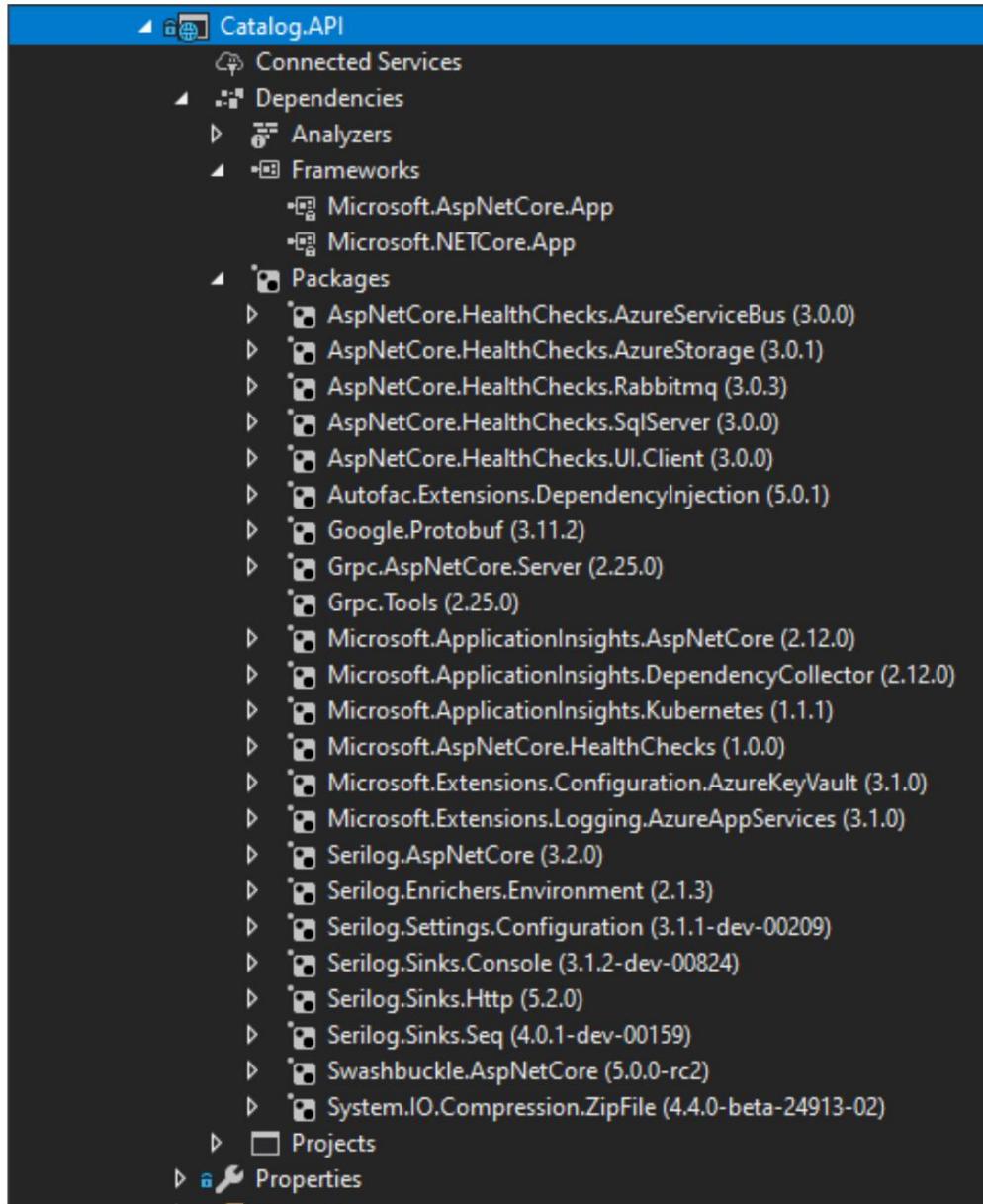


Figura 6-7. Dependencias en un microservicio CRUD Web API simple

El proyecto de API incluye referencias al paquete Microsoft.AspNetCore.App NuGet, que incluye referencias a todos los paquetes esenciales. Podría incluir algunos otros paquetes también.

### Implementación de servicios CRUD Web API con Entity Framework Core

Entity Framework (EF) Core es una versión liviana, extensible y multiplataforma de la popular tecnología de acceso a datos de 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 de .NET.

El microservicio de catálogo usa EF y el proveedor de SQL Server porque su base de datos se ejecuta en un contenedor con la imagen de Docker de SQL Server para Linux. Sin embargo, la base de datos podría desplegarse en

cualquier SQL Server, como Windows local o Azure SQL DB. Lo único que necesitaría cambiar es la cadena de conexión en el microservicio ASP.NET Web API.

### el modelo de datos

Con EF Core, el acceso a los datos se realiza mediante 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 le permite consultar y guardar datos. Puede generar un modelo a partir de una base de datos existente, codificar manualmente un modelo para que coincida con su base de datos o usar la técnica de migraciones de EF para crear una base de datos a partir de su modelo, utilizando el enfoque de código primero (que facilita la evolución de la base de datos como su modelo). cambia con el tiempo). Para el microservicio de catálogo, se ha utilizado el último enfoque. Puede ver un ejemplo de la clase de entidad CatalogItem en el siguiente ejemplo de código, que es una clase de entidad [Simple Old Class Object \(POCO\)](#).

```
elemento de catálogo de clase pública
{
    Id int público { obtener; colocar; }
    cadena pública Nombre { obtener; colocar; }
    public string Descripción { get; colocar; } Precio decimal
    público { obtener; colocar; } public string PictureFileName
    { get; colocar; } cadena pública PictureUri { obtener; colocar; }
    public int CatalogTypeId { obtener; colocar; } public CatalogType
    CatalogType { get; colocar; } public int CatalogBrandId { get;
    colocar; } public CatalogBrand CatalogBrand { get; colocar; }
    public int Existencias disponibles { get; colocar; } public int
    RestockThreshold { get; colocar; } public int MaxStockThreshold
    { get; colocar; }

    public bool OnReorder { get; colocar; } public
    CatalogItem() {}

    // Código adicional...
}
```

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, como se muestra en el siguiente ejemplo:

```
clase pública CatalogContext: DbContext {

    Public CatalogContext(DbContextOptions<CatalogContext> options) : base(options) {} public DbSet<CatalogItem>
    CatalogItems { get; colocar; } public DbSet<CatalogBrand> CatalogBrands { get; colocar; } public DbSet<CatalogType>
    CatalogTypes { get; colocar; }

    // Código adicional...
}
```

Puede tener [implementaciones de DbContext adicionales](#). Por ejemplo, en el microservicio Catalog.API de muestra, hay un segundo DbContext llamado CatalogContextSeed donde rellena automáticamente los datos de muestra la primera vez que intenta acceder a la base de datos. Este método es útil para datos de demostración y también para escenarios de prueba automatizados.

Dentro de DbContext, utiliza el método OnModelCreating para personalizar asignaciones de entidad de objeto/base de datos y otros [puntos de extensibilidad de EF](#).

### Consulta de datos de controladores de API web

Las instancias de sus clases de entidad generalmente se recuperan de la base de datos utilizando Language-Integrated Query (LINQ), como se muestra en el siguiente ejemplo:

```
[Ruta("api/v1/[controlador]")] clase
pública CatalogController : ControllerBase {

    privado de solo lectura CatalogContext _catalogContext;
    configuración privada de solo lectura CatalogSettings _settings;
    privado de solo lectura ICatalogIntegrationEventService _catalogIntegrationEventService;

    Controlador de catálogo público (
        Contexto CatalogContext,
        Configuración de IOptionsSnapshot<CatalogSettings>,
        ICatalogIntegrationEventService catalogIntegrationEventService)
    {
        _catalogContext = contexto ?? lanzar una nueva excepción ArgumentNullException (nombre de (contexto));
        _catalogIntegrationEventService = catalogIntegrationEventService
            ?? throw new ArgumentNullException(nameof(catalogIntegrationEventService));

        _configuraciones =
        configuraciones.Valor; context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
    }

    // OBTENER api/v1/[controlador]/elementos[?pageSize=3&pageIndex=10]
    [HttpGet]
    [Ruta("elementos")]
    [ProducesResponseType(typeof(PaginatedItemsViewModel<CatalogItem>),
    (int) HttpStatusCode.OK)]
    [ProducesResponseType(typeof(IEnumerable<CatalogItem>), (int) HttpStatusCode.OK)]
    [ProducesResponseType((int) HttpStatusCode.BadRequest)] public async
    Task<IActionResult> ItemsAsync([FromQuery]int pageSize = 10,
        [FromQuery]int pageIndex = 0, string ids = null)

    {
        if (!string.IsNullOrEmpty(ids))

            var items = esperar GetItemsByIdsAsync(ids);

            si (!elementos. Cualquiera ())
            {
                return BadRequest(" Valor de ID no válido. Debe ser una lista separada por comas de
números");
            }

            volver Ok(artículos);
    }

    var totalItems = esperar _catalogContext.CatalogItems .LongCountAsync();

    var itemsOnPage = await _catalogContext.CatalogItems .OrderBy(c =>
        c.Name)
        .Saltar(tamaño de página * índice de página)
```

```

    .Take(tamaño de página)
    .ToListAsync();

    itemsOnPage = ChangeUriPlaceholder(itemsOnPage);

    var model = new PaginatedItemsViewModel<CatalogItem>(pageIndex,
        pageSize, totalItems, itemsOnPage);

    volver Ok(modelo);

} //...
}

```

#### Guardando datos

Los datos se crean, eliminan y modifican en la base de datos utilizando instancias de sus clases de entidad. Puede agregar código como el siguiente ejemplo codificado (datos simulados, en este caso) a sus controladores de API web.

```

var catalogItem = new CatalogItem() {CatalogTypeId=2, CatalogBrandId=2,
    Nombre="Camiseta Roslyn", Precio = 12};
_contexto.Catálogo.Agregar(artículo de
catálogo); _contexto.GuardarCambios();

```

#### Inyección de dependencia en ASP.NET Core y controladores Web API

En ASP.NET Core, puede usar Inyección de dependencia (DI) lista para usar. No necesita configurar un contenedor de inversión de control (IoC) de terceros, aunque puede conectar su contenedor de IoC preferido en la infraestructura de ASP.NET Core si lo desea. En este caso, significa que puede inyectar directamente el EF DbContext requerido o repositorios adicionales a través del constructor del controlador.

En la clase CatalogController mencionada anteriormente, el tipo CatalogContext (que hereda de DbContext) se inyecta junto con los demás objetos necesarios en el constructor CatalogController() .

Una configuración importante para establecer en el proyecto de API web es el registro de la clase DbContext en el contenedor IoC del servicio. Por lo general, lo hace en la clase Startup llamando al método services.AddDbContext<CatalogContext>() dentro del método ConfigureServices() , como se muestra en el siguiente ejemplo simplificado :

```

public void ConfigureServices(servicios IServiceCollection) {

    // Código adicional...
    servicios.AddDbContext<CatalogContext>(opciones => {

        opciones.UseSqlServer(Configuración["ConnectionString"]);
        sqlServerOptionsAction: sqlOptions => {

            sqlOptions.MigrationsAssembly(
                typeof(Inicio).GetTypeInfo().Assembly.GetName().Name);

            //Configuración de la resistencia de la conexión:
            sqlOptions.
                EnableRetryOnFailure(maxRetryCount: 5,
                    maxRetryDelay: TimeSpan.FromSeconds(30),
                    errorNumbersToAdd: null);
        }
    });
}

```

```

    });

    // Cambiar el comportamiento predeterminado cuando se produce la evaluación del cliente para lanzar.
    // El valor predeterminado en EFCore sería registrar una advertencia cuando se realiza la
    // evaluación del cliente. opciones.ConfigureWarnings(advertencias =>
        advertencias.Throw( RelationalEventId.QueryClientEvaluationWarning));
    });

    //...
}

```

## Recursos adicionales

- Consulta de datos  
<https://docs.microsoft.com/ef/core/querying/index>
- Guardar datos  
<https://docs.microsoft.com/ef/core/saving/index>

## La cadena de conexión de la base de datos y las variables de entorno utilizadas por los contenedores de Docker

Puede usar la configuración de ASP.NET Core y agregar una propiedad ConnectionString a su archivo settings.json como se muestra en el siguiente ejemplo:

```
{
    "ConnectionString": "Servidor=tcp:127.0.0.1,5433;Catálogo
initial=Microsoft.eShopOnContainers.Services.CatalogDb;ID de usuario=sa;Contraseña=[MARCADOR]",
    "ExternalCatalogBaseUrl": "http://host.docker.internal:5101", "Registro":
    { "IncludeScopes": falso, "LogLevel": { "Predeterminado": "Depuración",
        "Sistema": "Información", "Microsoft": "Información"
    }
}
}
```

El archivo settings.json puede tener valores predeterminados para la propiedad ConnectionString o para cualquier otra propiedad. Sin embargo, esas propiedades serán anuladas por los valores de las variables de entorno que especifique en el archivo docker-compose.override.yml, cuando use Docker.

Desde sus archivos docker-compose.yml o docker-compose.override.yml, puede inicializar esas variables de entorno para que Docker las configure como variables de entorno del sistema operativo para usted, como se muestra en el siguiente archivo docker-compose.override.yml (la cadena de conexión y otras líneas se envuelven en este ejemplo, pero no en su propio archivo).

```
# docker-compose.override.yml

#
catalogo-api:
  entorno:
  -
    ConnectionString=Servidor=sqldata;Base de datos=Microsoft.eShopOnContainers.Services.CatalogDb;Uso
```

```
r Id=sa;Contraseña=[MARCADOR DE POSICIÓN]
# Variables de entorno adicionales para este servicio
puertos:
- "5101:80"
```

Los archivos docker-compose.yml a nivel de solución no solo son más flexibles que los archivos de configuración a nivel de proyecto o microservicio, sino que también son más seguros si anula las variables de entorno declaradas en los archivos docker-compose con valores establecidos desde sus herramientas de implementación. , como las tareas de implementación de Docker de Azure DevOps Services.

Finalmente, puede obtener ese valor de su código usando Configuration["ConnectionString"], como se muestra en el método ConfigureServices en un ejemplo de código anterior.

Sin embargo, para entornos de producción, es posible que desee explorar formas adicionales sobre cómo almacenar secretos como las cadenas de conexión. Una forma excelente de administrar los secretos de las aplicaciones es usar [Azure Key Vault](#).

---

Azure Key Vault ayuda a almacenar y proteger las claves criptográficas y los secretos que utilizan sus aplicaciones y servicios en la nube. Un secreto es cualquier cosa sobre la que desea mantener un control estricto, como claves API, cadenas de conexión, contraseñas, etc. y el control estricto incluye el registro de uso, la configuración de la caducidad, la gestión del acceso, entre otros.

Azure Key Vault permite un nivel de control detallado del uso de los secretos de la aplicación sin necesidad de que nadie los conozca. Los secretos incluso se pueden rotar para mejorar la seguridad sin interrumpir el desarrollo o las operaciones.

Las aplicaciones deben estar registradas en el Active Directory de la organización para que puedan usar Key Vault.

Puede consultar la documentación de Conceptos de Key Vault para obtener más detalles.

### [Implementación de control de versiones en ASP.NET Web API](#)

A medida que cambian los requisitos comerciales, se pueden agregar nuevas colecciones de recursos, las relaciones entre los recursos pueden cambiar y la estructura de los datos en los recursos puede modificarse.

Actualizar una API web para manejar los nuevos requisitos es un proceso relativamente sencillo, pero debe considerar los efectos que tales 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 esa API, el desarrollador no tiene el mismo grado de control sobre las aplicaciones cliente que pueden crear organizaciones de terceros que operan de forma remota.

El control de versiones permite que una API web indique las funciones y los recursos que expone. Luego, una aplicación cliente puede enviar solicitudes a una versión específica de una característica o recurso. Existen varios enfoques para implementar el control de versiones:

- Control de versiones de URI
- Control de versiones de cadenas de consulta
- Versiones de encabezado

La cadena de consulta y el control de versiones de URI son los más simples de implementar. El control de versiones de encabezado es un buen enfoque. Sin embargo, el control de versiones de encabezado no es tan explícito y directo como el control de versiones de URI.

Debido a que el control de versiones de URL es el más simple y explícito, la aplicación de ejemplo eShopOnContainers utiliza el control de versiones de URI.

Con el control de versiones de URI, como en la aplicación de muestra eShopOnContainers, cada vez que modifica la API web o cambia el esquema de los recursos, agrega un número de versión al URI para cada recurso.

Los URI existentes deberían continuar funcionando como antes, devolviendo recursos que se ajusten al esquema que coincida con la versión solicitada.

Como se muestra en el siguiente ejemplo de código, la versión se puede establecer mediante el atributo Route en el controlador de API web, lo que hace que la versión sea explícita en el URI (v1 en este caso).

```
[Ruta("api/v1/[controlador]")] clase
pública CatalogController : ControllerBase {
    // Implementación...
```

Este mecanismo de control de versiones es simple y depende de que el servidor enrute la solicitud al punto final apropiado. Sin embargo, para un control de versiones más sofisticado y el mejor método al usar REST, debe usar hipermedia e implementar [HATEOAS \(Hypertext as the Engine of Application State\)](#).

## Recursos adicionales

- Scott Hanselman. Control de versiones de ASP.NET Core RESTful Web API simplificado <https://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>
- Versión 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, hipermedia y REST <https://www.infoq.com/articles/roy-fielding-on-versioning>

## Generación de metadatos de descripción de Swagger desde su API web ASP.NET Core

[Swagger](#) es un marco de código abierto de uso común respaldado por un gran ecosistema de herramientas que lo ayuda a diseñar, compilar, documentar y consumir sus API RESTful. Se está convirtiendo en el estándar para el dominio de metadatos de descripción de API. Debe incluir metadatos de descripción de Swagger con cualquier tipo de microservicio, ya sea microservicios basados en datos o microservicios basados en dominio más avanzados (como se explica en la siguiente sección).

El corazón de Swagger es la especificación de Swagger, que son metadatos de descripción de API en un archivo JSON o YAML. La especificación crea el contrato RESTful para su API, detallando todos sus recursos y operaciones en un formato legible por humanos y máquinas para facilitar el desarrollo, el descubrimiento y la integración.

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 cómo se puede descubrir un servicio y cómo se entienden 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 de Swagger (<https://swagger.io>).

---

### ¿Por qué usar Swagger?

Las principales razones para generar metadatos de Swagger para sus API son las siguientes.

Capacidad para que otros productos consuman e integren automáticamente sus API. Docenas de productos y [herramientas comerciales](#) y muchas [bibliotecas y marcos](#) admiten Swagger. Microsoft tiene productos y herramientas de alto nivel que pueden consumir automáticamente las API basadas en Swagger, como las siguientes:

- [Descanso automático](#). Puede generar automáticamente clases de cliente .NET para llamar a Swagger. Esta herramienta se puede usar desde la CLI y también se integra con Visual Studio para facilitar su uso a través de la GUI.
- [Flujo de Microsoft](#). Puede [usar e integrar automáticamente su API en un flujo de trabajo](#) de Microsoft Flow de alto nivel, sin necesidad de conocimientos de programación.
- [Microsoft PowerApps](#). Puede consumir automáticamente su API desde [las aplicaciones móviles de PowerApps](#) creadas con [PowerApps Studio](#), sin necesidad de conocimientos de programación.
- [Aplicaciones lógicas de Azure App Service](#). Puede [usar e integrar automáticamente su API en una aplicación lógica de Azure App Service](#), sin necesidad de conocimientos de programación.

Capacidad para generar automáticamente la documentación de la API. Cuando crea API RESTful a gran escala, como aplicaciones complejas basadas en microservicios, necesita manejar muchos puntos finales con diferentes modelos de datos utilizados en las cargas útiles de solicitud y respuesta. Tener la documentación adecuada y tener un explorador de API sólido, como lo obtiene con Swagger, es clave para el éxito de su API y la adopción por parte de los desarrolladores.

Los metadatos de Swagger son los que usan Microsoft Flow, PowerApps y Azure Logic Apps para comprender cómo 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 el más conocido es [Swashbuckle](#), que se usa actualmente en [eShopOnContainers](#) y lo cubriremos con cierto detalle en esta guía, pero también existe la opción de usar [NSwag](#), que puede [generar](#) clientes Typescript y API C#, así como controladores C#, desde un especificación Swagger u OpenAPI e incluso escaneando el .dll que contiene los controladores, utilizando [NSwagStudio](#).

---

### Cómo automatizar la generación de metadatos API Swagger con el paquete Swashbuckle NuGet

La generación manual de metadatos de Swagger (en un archivo JSON o YAML) puede ser un trabajo tedioso. Sin embargo, puede automatizar el descubrimiento de API de los servicios de API web de ASP.NET mediante el paquete [Swashbuckle NuGet](#) para generar dinámicamente metadatos de API de Swagger.

Swashbuckle genera automáticamente metadatos de Swagger para sus proyectos de API web ASP.NET. Admite proyectos de API web de ASP.NET Core y la API web de ASP.NET tradicional y cualquier otro tipo, como la aplicación de API de Azure, la aplicación móvil de Azure, los microservicios de Azure Service Fabric basados en ASP.NET. También es compatible con la API web simple implementada en contenedores, como en la aplicación de referencia.

Swashbuckle combina API Explorer y Swagger o [swagger-ui](#) para proporcionar una rica experiencia de descubrimiento y documentación para sus consumidores de API. Además de su motor generador de metadatos Swagger, Swashbuckle también contiene una versión integrada de swagger-ui, que servirá automáticamente una vez que se instale Swashbuckle.

Esto significa que puede complementar su API con una buena interfaz de usuario de descubrimiento para ayudar a los desarrolladores a usar su API. Requiere una pequeña cantidad de código y mantenimiento porque se genera automáticamente, lo que le permite concentrarse en crear su API. El resultado de API Explorer se parece a la Figura 6-8.

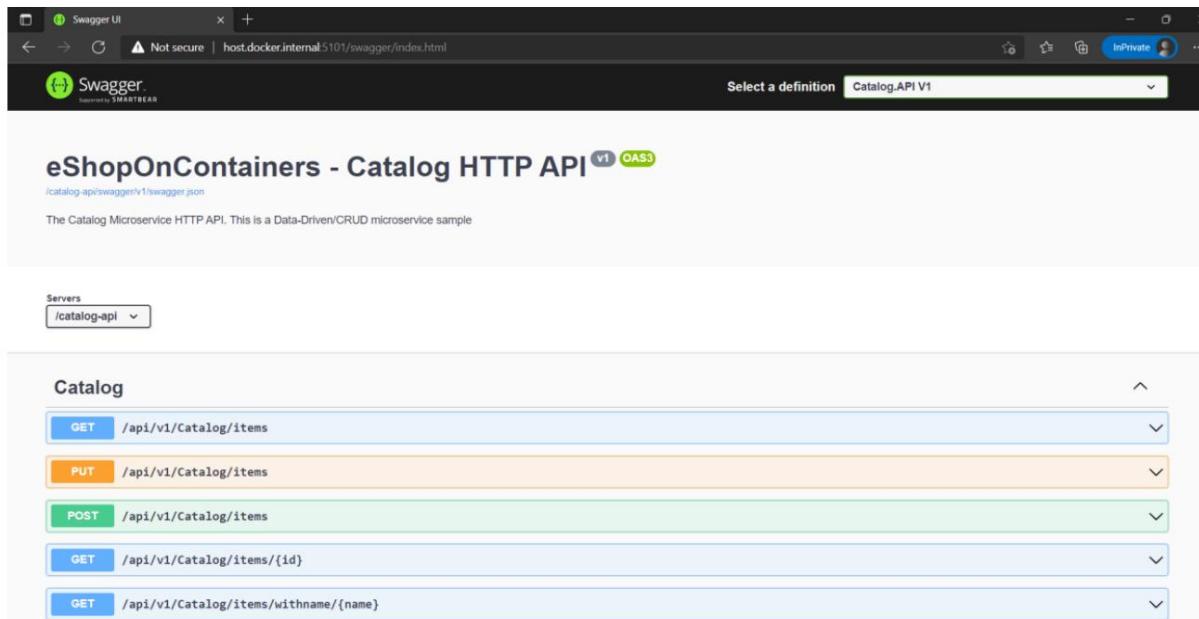


Figura 6-8. Swashbuckle API Explorer basado en metadatos de Swagger: microservicio de catálogo eShopOnContainers

La documentación de la API de la interfaz de usuario de Swagger generada por Swashbuckle incluye todas las acciones publicadas. El explorador de API no es lo más importante aquí. Una vez que tenga una API web que pueda describirse a sí misma en los metadatos de Swagger, su API se puede usar sin problemas desde las herramientas basadas en Swagger, incluidos los generadores de código de clase proxy de cliente que pueden dirigirse a muchas plataformas. Por ejemplo, como se mencionó, [AutoRest](#) genera automáticamente clases de cliente .NET. Pero también [hay disponibles herramientas adicionales](#) como [swagger-codegen](#), que permiten la generación de código de bibliotecas de clientes API, stubs de servidor y documentación automáticamente.

Actualmente, Swashbuckle consta de cinco paquetes NuGet internos bajo el metapquete de alto nivel [Swashbuckle.AspNetCore](#) para aplicaciones ASP.NET Core.

Después de haber instalado estos paquetes NuGet en su proyecto de API web, debe configurar Swagger en la clase Startup, como en el siguiente código simplificado :

```
Inicio de clase pública {
```

```

Configuración IConfigurationRoot pública { get; }

// Otro código de inicio...

public void ConfigureServices(IServiceCollection servicios) {

    // Otro código ConfigureServices()...

    // Agregar servicios de marco.
    servicios.AddSwaggerGen(opciones => {

        opciones.DescribeAllEnumsAsStrings();
        opciones.SwaggerDoc("v1", nuevo OpenApiInfo {

            Título = "eShopOnContainers - API HTTP de catálogo",
            Versión = "v1", Descripción = "API HTTP de microservicio de
            catálogo. Este es un ejemplo de microservicio basado en datos/CRUD" });

    });

    // Otro código ConfigureServices()...
}

configurar vacío público (aplicación IApplicationBuilder,
    IHostingEnvironment env,
    fábrica de registradores de ILoggerFactory)
{
    // Otro código Configure()... // ...
    app.UseSwagger()

        .UseSwaggerUI(c => {

            c.SwaggerEndpoint("/swagger/v1/swagger.json", "Mi API V1");
        });
}
}

```

Una vez hecho esto, puede iniciar su aplicación y explorar los siguientes puntos finales de Swagger JSON y UI utilizando direcciones URL como estas:

`http://<su-url-raíz>/swagger/v1/swagger.json`

`http://<su-url-raíz>/swagger/`

Anteriormente vio la interfaz de usuario generada creada por Swashbuckle para una URL como `http://<your-root url>/swagger`. En la Figura 6-9, también puede ver cómo puede probar cualquier método API.

The screenshot shows the Swagger UI interface for a .NET application. The top navigation bar indicates the URL as `localhost:5101/swagger/`. The main content area is titled "Catalog" and shows a "GET /api/v1/Catalog/items" operation. Under "Parameters", three query parameters are defined: `pageSize` (integer, value 12), `pageIndex` (integer, value 0), and `ids` (string, value `ids`). Below the parameters are "Responses" sections for "Curl" (containing a command to run the API call) and "Request URL" (showing the full URL `http://localhost:5101/api/v1/Catalog/items?pageSize=12&pageIndex=0`). The "Server response" section shows a successful 200 status code response with a JSON payload:

```
{
  "pageIndex": 0,
  "pageSize": 12,
  "count": 12,
  "data": [
    {
      "id": 2,
      "name": ".NET Black & White Mug",
      "description": ".NET Black & White Mug",
      "price": 8.5,
      "pictureFileName": "21.png",
      "pictureUrl": "http://localhost:5202/api/v1/c/catalog/items/2/pic/",
      "catalogTypeId": 1,
      "catalogBrandId": null,
      "catalogType": null,
      "catalogBrand": null,
      "availableStock": 100
    }
  ]
}
```

Figura 6-9. Interfaz de usuario de Swashbuckle probando el método API de catálogo/elementos

El detalle de la API de la interfaz de usuario de Swagger muestra una muestra de la respuesta y se puede usar para ejecutar la API real, lo cual es excelente para el descubrimiento del desarrollador. La figura 6-10 muestra los metadatos Swagger JSON generados a partir del microservicio eShopOnContainers (que es lo que usan las herramientas debajo) cuando solicita `http://<your-root-url>/swagger/v1/swagger.json` usando [Postman](#).

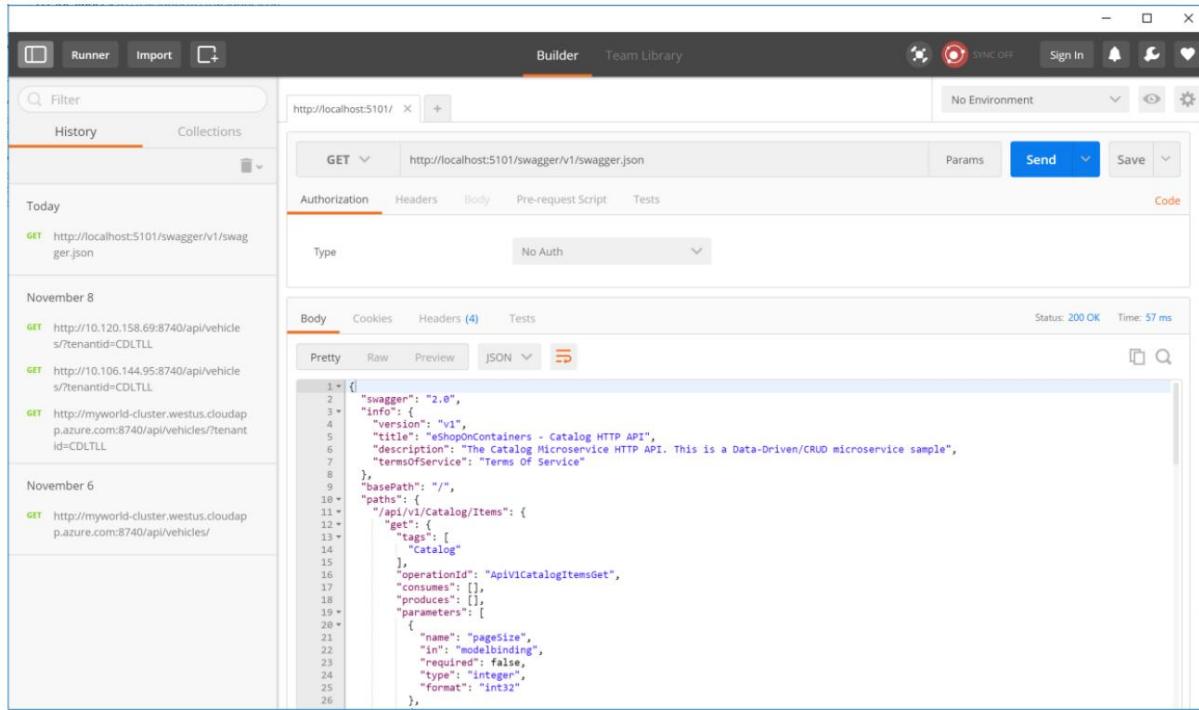


Figura 6-10. Metadatos Swagger JSON

Es así de simple. Y debido a que se genera automáticamente, los metadatos de Swagger crecerán cuando agregue más funcionalidad a su API.

## Recursos adicionales

- Páginas de ayuda de la API web de ASP.NET con Swagger  
<https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger>
- Comience con Swashbuckle y ASP.NET Core  
<https://docs.microsoft.com/aspnet/core/tutorials/getting-started-with-swashbuckle>
- Comience con NSwag y ASP.NET Core <https://docs.microsoft.com/aspnet/core/tutorials/getting-started-with-nswag>

## Definición de su aplicación multicontenedor con docker-compose.yml

En esta guía, el [archivo docker-compose.yml](#) se introdujo en la sección [Paso 4. Defina sus servicios en docker-compose.yml al crear una aplicación Docker de varios contenedores](#). Sin embargo, hay formas adicionales de usar los archivos docker-compose que vale la pena explorar con más detalle.

Por ejemplo, puede describir explícitamente cómo desea implementar su aplicación de varios contenedores en el archivo docker-compose.yml. Opcionalmente, también puede describir cómo va a crear sus imágenes de Docker personalizadas. (Las imágenes de Docker personalizadas también se pueden crear con la CLI de Docker).

Básicamente, define cada uno de los contenedores que desea implementar más ciertas características para cada implementación de contenedor. Una vez que tenga un archivo de descripción de implementación de varios contenedores, puede implementar la solución completa en una sola acción orquestada por el comando CLI docker-compose up, o puede implementarla de forma transparente desde Visual Studio. De lo contrario, deberá usar la CLI de Docker para implementar contenedor por contenedor en varios pasos mediante el comando docker run desde la línea de comandos. Por lo tanto, cada servicio definido en docker-compose.yml debe especificar exactamente una imagen o compilación. Otras teclas son opcionales y son análogas a sus contrapartes de línea de comandos de ejecución de ventana acopiable.

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

```
versión: '3.4'

servicios:
  webmvc:
    imagen: eshop/webmvc
    entorno:
      - CatalogUrl=http://catalog-api
      - OrderingUrl=http://ordering-api
      - BasketUrl=http://basket-api
    puertos:
      - "5100:80" depende_de: - catalog-api
        - ordering-api - basket-api

  catalog-api:
    imagen: eshop/catalog-api
    entorno:
      - ConnectionString=Server=sqldata;Initial Catalog=CatalogData;User
        Id=sa;Contraseña=[MARCADOR DE
        POSICIÓN] exponer: - "80"

    puertos:
      - "5101:80"
    #se pueden usar hosts adicionales para SQL Server independiente o servicios en la PC dev extra_hosts:
    - "CESARDLSURFBOOK:10.0.75.1" depende_de: - sqldata

  ordering-api:
    imagen: eshop/ordering-api
    entorno:
      - ConnectionString=Server=sqldata;Database=Services.OrderingDb;User
        Id=sa;Password=[PLACEHOLDER]
    puertos:
      - "5102:80"

    #se pueden usar hosts adicionales para SQL Server independiente o servicios en la PC dev extra_hosts:
    - "CESARDLSURFBOOK:10.0.75.1"

    depende_de:
      - sqldata
```

```
basket-api:
  imagen: eshop/basket-api entorno:
    - ConnectionString=sqldata puertos: -
      "5103:80" depende de: - sqldata
```

```
sqldata:
  entorno:
    - SA CONTRASEÑA=[MARCADOR]
    - ACCEPT_EULA=
  Puertos Y: - "5434:1433"
```

```
basketdata:
  imagen: redis
```

La clave raíz de este archivo es services. Bajo esa clave, define los servicios que desea implementar y ejecutar cuando ejecuta el comando docker-compose up o cuando implementa desde Visual Studio usando este archivo docker-compose.yml. En este caso, el archivo docker-compose.yml tiene varios servicios definidos, como se describe en la siguiente tabla.

Servicio nombre	Descripción
webmvc	Contenedor que incluye la aplicación ASP.NET Core MVC que consume los microservicios del lado del servidor C#
catálogo-api	Contenedor que incluye el microservicio Catalog ASP.NET Core Web API
ordering-api	Contenedor que incluye el microservicio Ordering ASP.NET Core Web API
sqldata	Contenedor que ejecuta SQL Server para Linux, que contiene las bases de datos de microservicios
cesta-api	Contenedor con el microservicio Basket ASP.NET Core Web API
cesta de datos	Contenedor que ejecuta el servicio de caché de REDIS, con la base de datos de cesta como caché de REDIS

## Un contenedor API de servicio web simple

Centrándose en un solo contenedor, catalog-api container-microservice tiene una definición sencilla:

```
catalog-api:
  imagen: eshop/catalog-api entorno:
    - ConnectionString=Server=sqldata;Initial Catalog=CatalogData;User
      Id=sa;Contraseña=[MARCADOR DE
      POSICIÓN] exponer: - "80"

  puertos:
    - "5101:80"
  #se pueden usar hosts adicionales para SQL Server independiente o servicios en la PC dev extra_hosts: -
  "CESARDLSURFBOOK:10.0.75.1"
```

**depende\_de:**  
- sqldata

Este servicio en contenedores tiene la siguiente configuración básica:

- Se basa en la imagen personalizada eshop/catalog-api . En aras de la simplicidad, no hay configuración de clave build: en el archivo. Esto significa que la imagen debe haber sido previamente construida (con docker build) o descargada (con el comando docker pull) desde cualquier registro de Docker.
- Define una variable de entorno denominada ConnectionString con la cadena de conexión que utilizará Entity Framework para acceder 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, necesita menos memoria en su máquina de desarrollo para Docker. Sin embargo, también podría implementar un contenedor de SQL Server para cada base de datos de microservicios.
- El nombre de SQL Server es sqldata, que es el mismo nombre que se usa para el contenedor que ejecuta la instancia de SQL Server para Linux. Esto es conveniente; poder usar esta resolución de nombres (interna en el host de Docker) resolverá la dirección de red, por lo que no necesita conocer la IP interna de los contenedores a los que accede desde otros contenedores.

Debido a que la cadena de conexión está definida por una variable de entorno, puede configurar esa variable a través de un mecanismo diferente y en un momento diferente. Por ejemplo, podría establecer una cadena de conexión diferente al implementar en producción en los hosts finales, o hacerlo desde sus canalizaciones de CI/CD en Azure DevOps Services o su sistema DevOps preferido.

- Expone el puerto 80 para el acceso interno al servicio catalog-api dentro del host Docker. Actualmente, el host es una máquina virtual de Linux porque se basa en una imagen de Docker para Linux, pero puede configurar el contenedor para que se ejecute en una imagen de Windows.
- Reenvía el puerto 80 expuesto en el contenedor al puerto 5101 en la máquina host de Docker (la VM de Linux).
- Vincula el servicio web al servicio sqldata (la instancia de SQL Server para la base de datos de Linux que se ejecuta en un contenedor). Cuando especifica esta dependencia, el contenedor catalog-api no se iniciará hasta que el contenedor sqldata ya se haya iniciado; este aspecto es importante porque catalog-api necesita tener la base de datos de SQL Server en funcionamiento primero. Sin embargo, este tipo de dependencia del contenedor no es suficiente en muchos casos, porque Docker solo verifica a nivel del contenedor.  
A veces, es posible que el servicio (en este caso, SQL Server) aún no esté listo, por lo que es recomendable implementar una lógica de reinicio con retroceso exponencial en los microservicios de su cliente. De esa forma, si un contenedor de dependencias no está listo por un tiempo breve, la aplicación seguirá siendo resistente.
- Está configurado para permitir el acceso a servidores externos: la configuración extra\_hosts le permite acceder a máquinas o servidores externos fuera del host Docker (es decir, fuera de la máquina virtual Linux predeterminada, que es un host Docker de desarrollo), como un servidor SQL local. instancia en su PC de desarrollo.

También hay otras configuraciones más avanzadas de docker-compose.yml que analizaremos en las siguientes secciones.

## Uso de archivos docker-compose para apuntar a múltiples entornos

Los archivos docker-compose.\*.yml son archivos de definición y pueden ser utilizados por múltiples infraestructuras que entienden ese formato. La herramienta más sencilla es el comando docker-compose.

Por lo tanto, al usar el comando docker-compose puede apuntar a los siguientes escenarios principales.

### Entornos de desarrollo

Cuando desarrolla aplicaciones, es importante poder ejecutar una aplicación en un entorno de desarrollo aislado.

Puede usar el comando CLI docker-compose para crear ese entorno o 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 su aplicación (otros servicios, caché, bases de datos, colas, etc.). Con el comando CLI docker-compose, puede crear e iniciar uno o más contenedores para cada dependencia con un solo comando (docker-compose up).

Los archivos docker-compose.yml son archivos de configuración interpretados por el motor de Docker, pero también sirven como archivos de documentación convenientes sobre la composición de su aplicación de varios contenedores.

### Entornos de prueba

Una parte importante de cualquier proceso de implementación continua (CD) o integración continua (CI) son las pruebas unitarias y las pruebas de integración. Estas pruebas automatizadas requieren un entorno aislado para que no se vean afectadas por los usuarios ni por ningún otro cambio en los datos de la aplicación.

Con Docker Compose, puede crear y destruir ese entorno aislado muy fácilmente con unos pocos comandos desde su símbolo del sistema o scripts, como los siguientes comandos:

```
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
```

### Despliegues de producción

También puede usar Compose para implementar en un Docker Engine remoto. Un caso típico es implementar en una sola instancia de host de Docker (como una VM de producción o un servidor aprovisionado con [Docker Machine](#)).

Si usa cualquier otro orquestador (Azure Service Fabric, Kubernetes, etc.), es posible que deba agregar opciones de configuración de configuración y metadatos como las de docker-compose.yml, pero en el formato requerido por el otro orquestador.

En cualquier caso, docker-compose es una herramienta conveniente y un formato de metadatos para flujos de trabajo de desarrollo, prueba y producción, aunque el flujo de trabajo de producción puede variar según el orquestador que esté utilizando.

## Usar múltiples archivos docker-compose para manejar varios entornos

Al apuntar a diferentes entornos, debe usar varios archivos de composición. Este enfoque le permite crear múltiples variantes de configuración según el entorno.

## Anulando el archivo base docker-compose

Podría usar un solo archivo docker-compose.yml como en los ejemplos simplificados que se muestran en las secciones anteriores. Sin embargo, eso 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 utiliza Visual Studio y habilita la compatibilidad con Docker, Visual Studio también crea un archivo adicional docker-compose.vs.debug.g.yml para depurar la aplicación. Puede consultar este archivo. en la carpeta obj\ Docker\ en la carpeta principal de la solución.

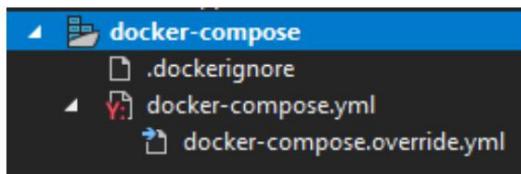


Figura 6-11. docker-compose archivos en Visual Studio 2019

Estructura del archivo del proyecto docker-compose :

- .dockerignore : se usa para ignorar archivos
- docker-compose.yml : se usa para componer microservicios docker-
- compose.override.yml : se usa para configurar el entorno de microservicios

Puede editar los archivos docker-compose con cualquier editor, como Visual Studio Code o Sublime, y ejecutar la aplicación con el comando docker-compose up.

Por convención, el archivo docker-compose.yml contiene su configuración básica y otras configuraciones estáticas. Eso significa que la configuración del servicio no debe cambiar según el entorno de implementación al que se dirige.

El archivo docker-compose.override.yml, como sugiere su nombre, contiene ajustes de configuración que anulan la configuración base, como la configuración que depende del entorno de implementación.

También puede tener varios archivos de anulación con diferentes nombres. Los archivos de invalidación suelen contener información adicional que necesita la aplicación, pero que es específica de un entorno o una implementación.

## Orientación a múltiples entornos

Un caso de uso típico es cuando define varios archivos de composición para que pueda dirigirse a varios entornos, como producción, ensayo, CI o desarrollo. Para admitir estas diferencias, puede dividir su configuración de Compose en varios archivos, como se muestra en la Figura 6-12.

## Multiple docker-compose files

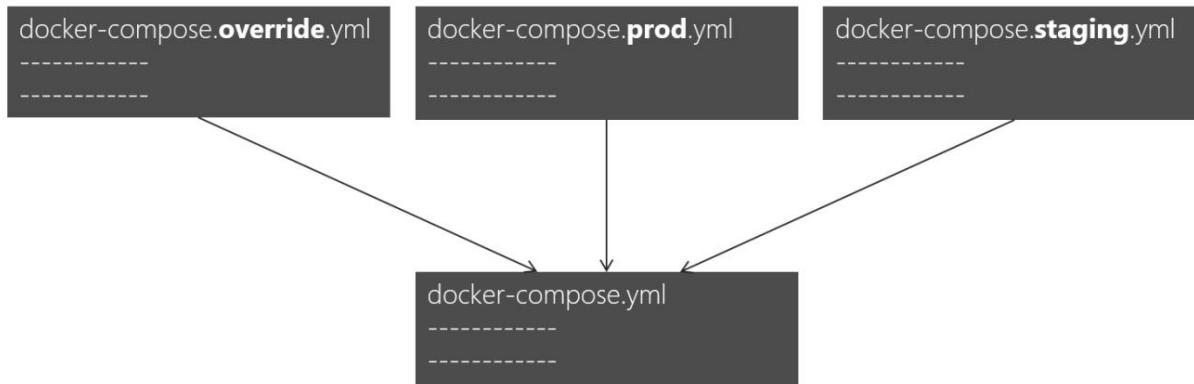


Figura 6-12. Múltiples archivos docker-compose que anulan los valores en el archivo base docker-compose.yml

Puede combinar varios archivos `docker-compose*.yml` para manejar diferentes entornos. Comienza con el archivo base `docker-compose.yml`. Este archivo base contiene los valores de configuración básicos o estáticos que no cambian según el entorno. Por ejemplo, la aplicación eShopOnContainers tiene el siguiente archivo `docker-compose.yml` (simplificado con menos servicios) como archivo base.

```

#docker-compose.yml (Base)
versión: '3.4' servicios: basket-api:
imagen: eshop/basket-api:${TAG:-latest} build: contexto: . dockerfile:
src/Services/Basket/Basket.API/Dockerfile depend_on: - basketdata - identidad-api - rabbitmq

catalog-api:
imagen: eshop/catalog-api:${TAG:-latest} build:
contexto: .
dockerfile: src/Services/Catalog/Catalog.API/Dockerfile depend_on: - sqldata
- rabbitmq

marketing-api:
imagen: eshop/marketing-api:${TAG:-latest} compilación:
contexto: .
dockerfile: src/Services/Marketing/Marketing.API/Dockerfile depende_en: - sqldata
- nosqldata - identidad-api - rabbitmq

webmvc:
  
```

```

imagen: eshop/webmvc:${TAG:-latest}
compilación:
  contexto: .
  dockerfile: src/Web/WebMVC/Dockerfile depend_on:
  - catalog-api - ordering-api - Identity-api - basket-api -
    marketing-api
  
```

```

sqldata:
  imagen: mcr.microsoft.com/mssql/server:2019-latest

nosqldata:
  imagen: mongo

basketdata:
  imagen: redis

rabbitmq:
  imagen: gestión de rabbitmq:3
  
```

Los valores en el archivo base docker-compose.yml no deberían cambiar debido a los diferentes entornos de implementación de destino.

Si se enfoca en la definición del servicio webmvc, por ejemplo, puede ver cómo esa información es muy similar sin importar el entorno al que se dirija. Tienes la siguiente información:

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

Puede tener una configuración adicional, pero el punto importante es que en el archivo base docker compose.yml, solo desea configurar la información que es común en todos los entornos. Luego, en docker-compose.override.yml o archivos similares para producción o ensayo, debe colocar la configuración que sea específica para cada entorno.

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

```

#docker-compose.override.yml (Configuración extendida para el entorno DE DESARROLLO)
versión: '3.4'

servicios:
# Número simplificado de servicios aquí:

basket-api:
  entorno: -
    ASPNETCORE_ENVIRONMENT=Desarrollo -
    ASPNETCORE_URLS=http://0.0.0.0:80 -
    ConnectionString=${ESHOP_AZURE_REDIS_BASKET_DB:-basketdata}
  
```

```

- identidadUrl=http://identidad-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=${INSTRUMENTATION_KEY}
- Tipo de orquestador=${ORCHESTRATOR_TYPE}
- UseLoadTest=${USE_LOADTEST:-False}

```

puertos:  
- "5103:80"

catálogo-api:  
medioambiente:

```

- ASPNETCORE_ENVIRONMENT=Desarrollo -
ASPNETCORE_URLS=http://0.0.0.0:80 -
ConnectionString=${ESHOP_AZURE_CATALOG_DB:-}
Servidor=sqldata;Base de datos=Microsoft.eShopOnContainers.Services.CatalogDb;Id. de
usuario=sa;Contraseña=[LUGAR]
- PicBaseUrl=${ESHOP_AZURE_STORAGE_CATALOG_URL}-
http://host.docker.internal:5202/api/v1/catalog/items/[0]/pic/
- EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
- EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
- EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
- AzureStorageAccountName=${ESHOP_AZURE_STORAGE_CATALOG_NAME}
- AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_CATALOG_KEY}
- UseCustomizationData=True
- AzureServiceBusEnabled=False
- AzureStorageEnabled=False
- ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
- OrchestratorType=${ORCHESTRATOR_TYPE}

```

puertos: - "5101:80"

marketing-api:  
entorno:

```

- ASPNETCORE_ENVIRONMENT=Desarrollo -
ASPNETCORE_URLS=http://0.0.0.0:80 -
ConnectionString=${ESHOP_AZURE_MARKETING_DB:-}
Servidor=sqldata;Base de datos=Microsoft.eShopOnContainers.Services.MarketingDb;Id. de
usuario=sa;Contraseña=[LUGAR]
- MongoConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}
- MongoDBDatabase=MarketingDb -
EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
- EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
- EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD} -
IdentityUrl=http://identity-api - IdentityUrlExternal=http://${
{ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105 - CampaignDetailFunctionUri=${
{ESHOP_AZUREFUNC_CAMPAIGN_DETAILS_URI}
- PicBaseUrl=${ESHOP_AZURE_STORAGE_MARKETING_URL}-
http://host.docker.internal:5110/api/v1/campaigns/[0]/pic/
- AzureStorageAccountName=${ESHOP_AZURE_STORAGE_MARKETING_NAME}
- AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_MARKETING_KEY}
- AzureServiceBusEnabled=False
- AzureStorageEnabled=False
- ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
- Tipo de orquestador=${ORCHESTRATOR_TYPE}
- UseLoadTest=${USE_LOADTEST:-False}

```

puertos: - "5110:80"

```

webmvc:
  entorno: -
    ASPNETCORE_ENVIRONMENT=Desarrollo -
    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://pago-api/hc -
    SignalrHubUrl=http://${
    {ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5202 -
    UseCustomizationData=True - ApplicationInsights__InstrumentationKey=$
    {INSTRUMENTATION_KEY}

    - Tipo de orquestador=${ORCHESTRATOR_TYPE}
    - UseLoadTest=${USE_LOADTEST:-False}
puertos: - "5100:80" sqldata: entorno: -
  SA_PASSWORD=[PLACEHOLDER]

    - ACCEPT_EULA=Y
  puertos: - "5433:1433"
    nosqldata: puertos:
    - "27017:27017" basketdata:

  puertos:
    - "6379:6379"
  rabbitmq: puertos: -
    "15672:15672" -
    "5672:5672"

```

En este ejemplo, la configuración de anulación de desarrollo expone algunos puertos al host, define variables de entorno con direcciones URL de redirección y especifica cadenas de conexión para el entorno de desarrollo. Estas configuraciones son solo para el entorno de desarrollo.

Cuando ejecuta docker-compose up (o lo inicia desde Visual Studio), el comando lee las anulaciones automáticamente como si estuviera fusionando ambos archivos.

Suponga que desea otro archivo Compose para el entorno de producción, con diferentes valores de configuración, puertos o cadenas de conexión. Puede crear otro archivo de anulación, como un archivo llamado docker-compose.prod.yml con diferentes configuraciones y variables de entorno. Ese archivo puede estar almacenado en un repositorio de Git diferente o administrado y protegido por un equipo diferente.

#### Cómo implementar con un archivo de anulación específico

Para usar varios archivos de anulación o un archivo de anulación con un nombre diferente, puede usar la opción -f con el comando docker-compose y especificar los archivos. Componer fusiona archivos en el orden en que se especifican en la línea de comandos. El siguiente ejemplo muestra cómo implementar con archivos de anulación.

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

## Uso de variables de entorno en archivos docker-compose

Es conveniente, sobre todo en entornos de producción, poder obtener información de configuración de las variables de entorno, como hemos mostrado en ejemplos anteriores. Puede hacer referencia a una variable de entorno en sus archivos docker-compose usando la sintaxis \${MY\_VAR}. La siguiente línea de un archivo docker-compose.prod.yml muestra cómo hacer referencia al valor de una variable de entorno.

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

Las variables de entorno se crean e inicializan de diferentes maneras, según el entorno de su host (Linux, Windows, clúster en la nube, etc.). Sin embargo, un enfoque conveniente es usar un archivo .env. Los archivos docker-compose admiten la declaración de variables de entorno predeterminadas en el archivo .env. Estos valores para las variables de entorno son los valores predeterminados. Pero pueden ser anulados por los valores que podría haber definido en cada uno de sus entornos (sistema operativo host o variables de entorno de su clúster). Coloque este archivo .env en la carpeta desde donde se ejecuta el comando docker-compose.

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

```
# archivo .env

ESHOP_EXTERNAL_DNS_NAME_OR_IP=host.docker.internal

ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=10.121.122.92
```

Docker-compose espera que cada línea de un archivo .env tenga el formato <variable>=<valor>.

Los valores establecidos en el entorno de tiempo de ejecución siempre anulan los valores definidos dentro del archivo .env. De manera similar, los valores pasados mediante argumentos de la línea de comandos también anulan los valores predeterminados establecidos en el archivo .env.

## Recursos adicionales

- Descripción general de Docker  
[Compose https://docs.docker.com/compose/overview/](https://docs.docker.com/compose/overview/)
- Múltiples archivos de composición <https://docs.docker.com/compose/extends/#multiple-compose-files>

## Creación de imágenes optimizadas de ASP.NET Core Docker

Si está explorando Docker y .NET en fuentes en Internet, encontrará Dockerfiles que demuestran la simplicidad de crear una imagen de Docker copiando su fuente en un contenedor. Estos ejemplos sugieren que al usar una configuración simple, puede tener una imagen de Docker con el entorno empaquetado con su aplicación. El siguiente ejemplo muestra un Dockerfile simple en este sentido.

```
DESDE mcr.microsoft.com/dotnet/sdk:6.0 WORKDIR /app
ENV ASPNETCORE_URLS http://+:80 EXPOSE 80

COPIAR _.
EJECUTAR dotnet restaurar
PUNTO DE ENTRADA ["dotnet", "ejecutar"]
```

Un Dockerfile como este funcionará. Sin embargo, puede optimizar sustancialmente sus imágenes, especialmente sus imágenes de producción.

En el modelo de contenedores y microservicios, constantemente inicia contenedores. La forma típica de usar contenedores no reinicia un contenedor durmiente, porque el contenedor es desecharable.

Los orquestadores (como Kubernetes y Azure Service Fabric) crean nuevas instancias de imágenes. Lo que esto significa es que necesitaría optimizar precompilando la aplicación cuando se construya para que el proceso de creación de instancias sea más rápido. Cuando se inicia el contenedor, debe estar listo para ejecutarse. No restaure ni compile en tiempo de ejecución con los comandos de la CLI `dotnet restore` y `dotnet build` como puede ver en las publicaciones de blog sobre .NET y Docker.

El equipo de .NET ha estado realizando un trabajo importante para hacer de .NET y ASP.NET Core un marco optimizado para contenedores. .NET no solo es un marco liviano con una huella de memoria pequeña; el equipo se centró en imágenes de Docker optimizadas para tres escenarios principales y las publicó en el registro de Docker Hub en `dotnet/`, comenzando con la versión 2.1:

1. Desarrollo: la prioridad es la capacidad de iterar y depurar cambios rápidamente, y dónde está el tamaño secundario.
2. Compilación : la prioridad es compilar la aplicación y la imagen incluye archivos binarios y otras dependencias para optimizar los archivos binarios.
3. Producción: el enfoque es la implementación y el inicio rápidos de los contenedores, por lo que estas imágenes se limitan a los binarios y el contenido necesarios para ejecutar la aplicación.

El equipo de .NET proporciona cuatro variantes básicas [en dotnet/](#) (en Docker Hub):

1. `sdk`: para escenarios de desarrollo y compilación 2.

`aspnet`: para escenarios de producción de ASP.NET 3.

`runtime`: para escenarios de producción de .NET 4.

`runtime-deps`: para escenarios de producción de [aplicaciones independientes](#)

Para un inicio más rápido, las imágenes en tiempo de ejecución también configuran automáticamente `aspnetcore_urls` en el puerto 80 y usan Ngen para crear una caché de imagen nativa de ensamblajes.

## Recursos adicionales

- Creación de imágenes de Docker optimizadas con ASP.NET Core  
<https://docs.microsoft.com/archive/blogs/stevelasker/building-optimized-docker-images-with-asp-net-core>
- Creación de imágenes de Docker para aplicaciones .NET  
<https://docs.microsoft.com/dotnet/core/docker/building-net-docker-images>

## Utilice un servidor de base de datos que se ejecuta como un contenedor

Puede tener sus bases de datos (SQL Server, PostgreSQL, MySQL, etc.) en servidores independientes regulares, en clústeres locales o en servicios PaaS en la nube como Azure SQL DB. Sin embargo, para entornos de desarrollo y prueba, es conveniente que sus bases de datos se ejecuten como contenedores, porque no

tiene alguna dependencia externa y simplemente ejecutar el comando docker-compose up inicia toda la aplicación. Tener esas bases de datos como contenedores también es excelente para las pruebas de integración, porque la base de datos se inicia en el contenedor y siempre se completa con los mismos datos de muestra, por lo que las pruebas pueden ser más predecibles.

## SQL Server ejecutándose como un contenedor con una base de datos relacionada con microservicios

En eShopOnContainers, hay un contenedor llamado sqldata, como se define en el archivo [docker-compose.yml](#), que ejecuta una instancia de SQL Server para Linux con las bases de datos SQL para todos los microservicios que lo necesitan.

Un punto clave en los microservicios es que cada microservicio posee sus datos relacionados, por lo que debe tener su propia base de datos. Sin embargo, las bases de datos pueden estar en cualquier lugar. En este caso, todos están en el mismo contenedor para mantener los requisitos de memoria de Docker lo más bajos posible. Tenga en cuenta que esta es una solución lo suficientemente buena para el desarrollo y, quizás, para las pruebas, pero no para la producción.

El contenedor de SQL Server en la aplicación de muestra está configurado con el siguiente código YAML en el archivo docker-compose.yml, que se ejecuta cuando ejecuta docker-compose up. Tenga en cuenta que el código YAML tiene información de configuración consolidada del archivo docker-compose.yml genérico y del archivo docker-compose.override.yml. (Por lo general, separaría la configuración del entorno de la base o la información estática relacionada con la imagen de SQL Server).

```
sqldata:
  imagen: mcr.microsoft.com/mssql/server:2017-último entorno:
    - SA_PASSWORD=Contraseña@palabra
    - ACCEPT_EULA=
  Puertos Y: - "5434:1433"
```

De manera similar, en lugar de usar docker-compose, el siguiente comando de ejecución de docker puede ejecutar ese contenedor:

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

Sin embargo, si está implementando una aplicación de múltiples contenedores como eShopOnContainers, es más conveniente usar el comando docker-compose up para que implemente todos los contenedores necesarios para la aplicación.

Cuando inicia este contenedor de SQL Server por primera vez, el contenedor inicializa SQL Server con la contraseña que proporcione. Una vez que SQL Server se ejecuta como un contenedor, puede actualizar la base de datos conectándose a través de cualquier conexión SQL normal, como SQL Server Management Studio, Visual Studio o código C#.

La aplicación eShopOnContainers inicializa cada base de datos de microservicios con datos de muestra sembrando datos en el inicio, como se explica en la siguiente sección.

Tener SQL Server ejecutándose como un contenedor no solo es útil para una demostración en la que es posible que no tenga acceso a una instancia de SQL Server. Como se señaló, también es excelente para el desarrollo y las pruebas.

entornos para que pueda ejecutar fácilmente pruebas de integración a partir de una imagen limpia de SQL Server y datos conocidos mediante la inicialización de nuevos datos de muestra.

## Recursos adicionales

- Ejecute la imagen de SQL Server Docker en Linux, Mac o Windows <https://docs.microsoft.com/sql/linux/sql-server-linux-setup-docker>
- Conecte y consulte SQL Server en Linux con sqlcmd <https://docs.microsoft.com/sql/linux/sql-server-linux-connect-and-query-sqlcmd>

## Siembra con datos de prueba en el inicio de la aplicación web

Para agregar datos a la base de datos cuando se inicia la aplicación, puede agregar un código como el siguiente al método Main en la clase Program del proyecto Web API:

```
public static int Principal(cadena[] argumentos) {
    var configuración = GetConfiguration();
    Log.Logger = CreateSerilogLogger(configuración);
    prueba
    {
        Log.Information("Configurando host web ({ApplicationContext}...)", AppName); var host =
        CreateHostBuilder(configuración, argumentos);

        Log.Information("Aplicando migraciones ({ApplicationContext}...)", AppName);
        host.MigrateDbContext<CatalogContext>((contexto, servicios) => {

            var env = servicios.GetService<IWebHostEnvironment>(); var settings =
            servicios.GetService<IOptions<CatalogSettings>>(); var logger =
            servicios.GetService<ILogger<CatalogContextSeed>>();

            nuevo CatalogContextSeed()
                .SeedAsync(contexto, entorno, configuración, registrador)
                .Esperar();
        })
        .MigrateDbContext<IntegrationEventLogContext>((_, __) => { });

        Log.Information("Iniciando host web ({ApplicationContext}...)", AppName); host.Ejecutar();
    }

    devolver 0;
} catch (excepción ex) {
    Log.Fatal(por ejemplo, "¡El programa finalizó inesperadamente ({ApplicationContext})!", AppName); devolver 1;
}

} finalmente
{
    Log.CloseAndFlush();
}
```

Hay una advertencia importante al aplicar migraciones y propagar una base de datos durante el inicio del contenedor. Dado que es posible que el servidor de la base de datos no esté disponible por cualquier motivo, debe manejar los reintentos mientras espera que el servidor esté disponible. Esta lógica de reinicio es manejada por el método de extensión `MigrateDbContext()`, como se muestra en el siguiente código:

```
public static IWebHost MigrateDbContext<TContext>( este host
    IWebHost,
    Acción<TContexto,
    IServiceProvider> sembradora)
    donde TContext: DbContext
{
    var underK8s = host.IsInKubernetes();

    usando (var alcance = host.Services.CreateScope()) {

        var servicios = scope.ServiceProvider;

        var logger = services.GetRequiredService<ILogger<TContext>>();

        var context = services.GetService<TContext>();

        prueba
        {
            logger.LogInformation("Migrando la base de datos asociada con el contexto
{DbContextName}", typeof(TContext).Nombre);

            si (bajoK8s) {

                InvokeSeeder(sembrador, contexto, servicios);

            } más
            {
                var reintentar = Policy.Handle<SqlException>()
                    .WaitAndRetry(new TimeSpan[]
                    { TimeSpan.FromSeconds(3),
                    TimeSpan.FromSeconds(5),
                    TimeSpan.FromSeconds(8), });

                //si el contenedor del servidor sql no se crea al ejecutar la ventana acopiable, esta //migración no
                //puede fallar debido a una excepción relacionada con la red. Las opciones de reinicio para
                Solo contexto de base de datos
                //aplicar a excepciones transitorias
                // Tenga en cuenta que esto NO se aplica cuando se ejecutan algunos orquestadores (deje que el
                orquestador vuelva a crear el servicio fallido)
                retry.Execute(() => InvokeSeeder(sembrador, contexto, servicios));
            }
            logger.LogInformation(" Base de datos migrada asociada con el contexto
{DbContextName}", typeof(TContext).Nombre);

        } catch (excepción ex) {

            logger.LogError(por ejemplo, "Se produjo un error al migrar la base de datos utilizada en el contexto
{DbContextName}", typeof(TContext).Name); si (bajoK8s) {

                lanzar;           // Vuelva a lanzar bajo k8s porque confiamos en k8s para volver a ejecutar el
            }
        }
    }
}
```

```

    vaina
    }
}
anfitrón de retorno ;
}

```

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

```

clase pública CatalogContextSeed {

    tarea asincrónica estática pública SeedAsync (IApplicationBuilder applicationBuilder) {

        var context = (CatalogContext) constructor de aplicaciones
            .ApplicationServices.GetService(typeof(CatalogContext)); usando
        (contexto) {

            contexto.Base de datos.Migrar(); if
            (!context.CatalogBrands.Any()) {

                contexto.CatalogBrands.AddRange(
                    GetPreconfiguredCatalogBrands());
                espera context.SaveChangesAsync();

            } if (!context.CatalogTypes.Any()) {

                contexto.CatalogTypes.AddRange(
                    GetPreconfiguredCatalogTypes());
                espera context.SaveChangesAsync();
            }
        }
    }

    IEnumerable estático <CatalogBrand> GetPreconfiguredCatalogBrands() {

        devuelve nueva Lista<MarcaCatálogo>()
        {
            new CatalogBrand() { Marca = "Azure"}, new
            CatalogBrand() { Brand = ".NET" }, new
            CatalogBrand() { Brand = "Visual Studio" }, new CatalogBrand()
            { Brand = "SQL Server" }
        };
    }

    IEnumerable estático <CatalogType> GetPreconfiguredCatalogTypes() {

        devuelve nueva Lista<TipoCatálogo>()
        {
            new CatalogType() { Tipo = "Mug"}, new
            CatalogType() { Tipo = "T-Shirt" }, new CatalogType()
            { Tipo = "Mochila" }, new CatalogType() { Tipo =
            "USB Memory Stick" }
        };
    }
}

```

Cuando ejecuta pruebas de integración, es útil tener una forma de generar datos consistentes con sus pruebas de integración. Ser capaz de crear todo desde cero, incluida una instancia de SQL Server ejecutándose en un contenedor, es excelente para entornos de prueba.

## Base de datos EF Core InMemory frente a SQL Server ejecutándose como contenedor

Otra buena opción cuando se ejecutan pruebas es usar el proveedor de base de datos InMemory de Entity Framework. Puede especificar esa configuración en el método ConfigureServices de la clase Startup en su proyecto Web API:

```
Inicio de clase pública {
    // Otro código de inicio... public
    void ConfigureServices(IServiceCollection services) {
        services.AddSingleton< IConfiguration>(Configuración); // DbContext
        utilizando un proveedor de base de datos InMemory
        services.AddDbContext< CatalogContext >(opt => opt.UseInMemoryDatabase()); // (Alternativa:
        DbContext usando un proveedor de SQL Server //services.AddDbContext< CatalogContext >(c
        => // { // c.UseSqlServer(Configuration["ConnectionString"]); // });
    }
    // Otro código de inicio...
}
```

Sin embargo, hay una captura importante. La base de datos en memoria no admite muchas restricciones que son específicas de una base de datos en particular. Por ejemplo, puede agregar un índice único en una columna en su modelo EF Core y escribir una prueba en su base de datos en memoria para verificar que no le permita agregar un valor duplicado. Pero cuando usa la base de datos en memoria, no puede manejar índices únicos en una columna. Por lo tanto, la base de datos en memoria no se comporta exactamente igual que una base de datos de SQL Server real: no emula las restricciones específicas de la base de datos.

Aun así, una base de datos en memoria sigue siendo útil para realizar pruebas y prototipos. Pero si desea crear pruebas de integración precisas que tengan en cuenta el comportamiento de una implementación de base de datos específica, debe usar una base de datos real como SQL Server. Para ese propósito, ejecutar SQL Server en un contenedor es una excelente opción y más precisa que el proveedor de base de datos 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 para escenarios de prueba de concepto. Este escenario es conveniente, porque puede tener todas sus dependencias ejecutándose en contenedores, no solo para sus máquinas de desarrollo locales, sino también para sus entornos de prueba en sus canalizaciones de CI/CD.

Sin embargo, cuando ejecuta Redis en producción, es mejor buscar una solución de alta disponibilidad como Redis Microsoft Azure, que se ejecuta como PaaS (Platform as a Service). En su código, solo necesita cambiar sus cadenas de conexión.

Redis proporciona una imagen de Docker con Redis. Esa imagen está disponible en Docker Hub en esta URL:

[https://hub.docker.com/\\_/redis/](https://hub.docker.com/_/redis/)

Puede ejecutar directamente un contenedor de Docker Redis ejecutando el siguiente comando de la CLI de Docker en su símbolo del sistema:

```
ventana acopiable ejecutar --nombre some-redis -d redis
```

La imagen de Redis incluye la exposición: 6379 (el puerto utilizado por Redis), por lo que la vinculación de contenedores estándar hará que esté disponible automáticamente para los contenedores vinculados.

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

```
#docker-compose.yml archivo #...
basketdata: imagen: exposición
redis : - "6379"
```

Este código en docker-compose.yml define un contenedor llamado basketdata basado en la imagen redis y publica el puerto 6379 internamente. Esta configuración significa que solo se podrá acceder a ella desde otros contenedores que se ejecuten dentro del host de Docker.

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

```
basket-api:
  entorno: # Otros
  datos...
  - Cadena de conexión=datos de la cesta
  - EventBusConnection=rabbitmq
```

Como se mencionó anteriormente, el DNS de la red interna de Docker resuelve el nombre de los datos de la cesta del microservicio.

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

Como se describió anteriormente, cuando utiliza la comunicación basada en eventos, un microservicio publica un evento cuando sucede algo notable, como cuando actualiza una entidad comercial. Otros microservicios se suscriben a esos eventos. Cuando un microservicio recibe un evento, puede actualizar sus propias entidades comerciales, lo que podría generar la publicación de más eventos. Esta es la esencia del concepto de consistencia eventual. Este sistema de publicación/suscripción generalmente se realiza utilizando una implementación de un bus de eventos. El bus de eventos se puede diseñar como una interfaz con la API necesaria para suscribirse y cancelar la suscripción a eventos y para publicar eventos. También puede tener una o más implementaciones basadas en cualquier comunicación entre procesos o mensajería, como una cola de mensajería o un bus de servicio que admite comunicación asíncrona y un modelo de publicación/suscripción.

Puede usar eventos para implementar transacciones comerciales que abarquen múltiples servicios, lo que le brinda coherencia eventual entre esos servicios. Una transacción eventualmente consistente consiste en una serie

de acciones distribuidas. En cada acción, el microservicio actualiza una entidad empresarial y publica un evento que desencadena la siguiente acción. La Figura 6-18 a continuación muestra un evento PriceUpdated publicado a través de un bus de eventos, por lo que la actualización del precio se propaga a la cesta y a otros microservicios.

## Implementing asynchronous event-driven communication with an event bus

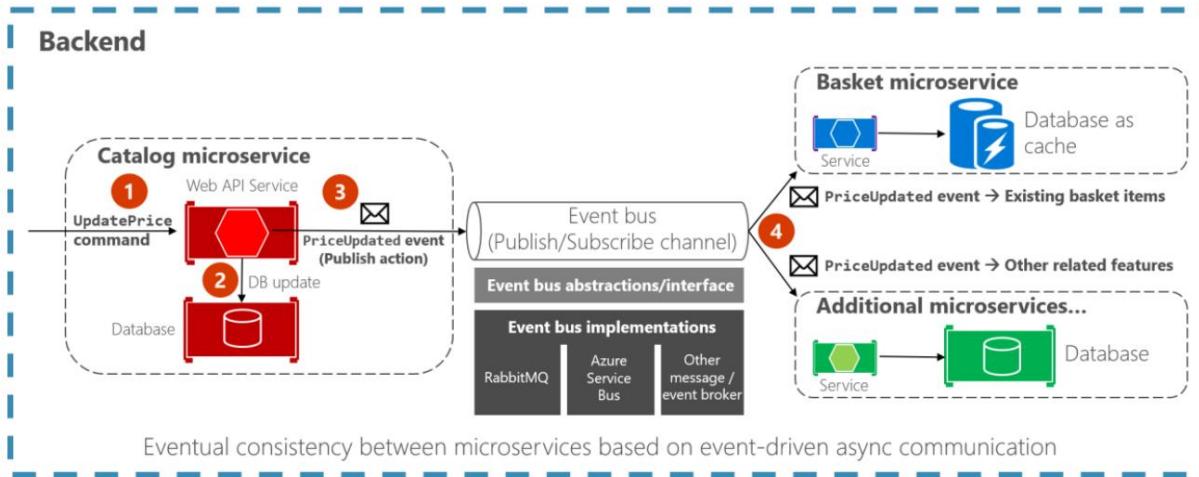


Figura 6-18. Comunicación basada en eventos basada en un bus de eventos

Esta sección describe cómo puede implementar este tipo de comunicación con .NET mediante una interfaz de bus de eventos genérica, como se muestra en la Figura 6-18. Existen múltiples implementaciones potenciales, cada una de las cuales utiliza una tecnología o infraestructura diferente, como RabbitMQ, Azure Service Bus o cualquier otro bus de servicio comercial o de código abierto de terceros.

### Uso de intermediarios de mensajes y buses de servicios para sistemas de producción

Como se indicó en la sección de arquitectura, puede elegir entre múltiples tecnologías de mensajería para implementar su bus de eventos abstractos. Pero estas tecnologías están en diferentes niveles. Por ejemplo, RabbitMQ, un agente de transporte de mensajería, se encuentra en un nivel más bajo que los productos comerciales como Azure Service Bus, NServiceBus, MassTransit o Brighter. La mayoría de estos productos pueden funcionar sobre RabbitMQ o Azure Service Bus. Su elección de producto depende de cuántas características y cuánta escalabilidad lista para usar necesita para su 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 simple sobre RabbitMQ ejecutándose como un contenedor podría ser suficiente. Pero para los sistemas de producción y de misión crítica que necesitan una alta escalabilidad, es posible que desee evaluar y usar Azure Service Bus.

Si necesita abstracciones de alto nivel y características más ricas como [Sagas](#) para procesos de ejecución prolongada que facilitan el desarrollo distribuido, vale la pena evaluar otros buses de servicio comerciales y de código abierto como NServiceBus, MassTransit y Brighter. En este caso, las abstracciones y la API que se usarán normalmente serán directamente las proporcionadas por esos buses de servicio de alto nivel en lugar de sus propias abstracciones (como las [abstracciones simples del bus de eventos proporcionadas en eShopOnContainers](#)). Para esa materia,

puede investigar los [eShopOnContainers bifurcados utilizando NServiceBus](#) (muestra derivada adicional implementada por Particular Software).

Por supuesto, siempre puede crear sus propias funciones de bus de servicio sobre tecnologías de nivel inferior como RabbitMQ y Docker, pero el trabajo necesario para "reinventar la rueda" puede ser demasiado costoso para una aplicación empresarial personalizada.

Para reiterar: las abstracciones y la implementación del bus de eventos de muestra que se muestran en la muestra de eShopOnContainers están destinadas a usarse solo como prueba de concepto. Una vez que haya decidido que desea tener una comunicación asíncrona y basada en 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 del dominio entre múltiples microservicios o sistemas externos. Esta funcionalidad se realiza mediante la publicación de eventos de integración fuera del microservicio. Cuando se publica un evento en varios microservicios receptores (en tantos microservicios como estén suscritos al evento de integración), el controlador de eventos apropiado en cada microservicio receptor maneja el evento.

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

```
clase pública ProductPriceChangedIntegrationEvent : IntegrationEvent {

    public int ProductId { obtener; conjunto privado; } public
    decimal NewPrice { get; conjunto privado; } public decimal
    OldPrice { get; conjunto privado; }

    public ProductPriceChangedIntegrationEvent(int productId, decimal newPrice,
        precio anterior decimal )
    {
        IdProducto = IdProducto;
        NuevoPrecio = nuevoPrecio;
        PrecioAntiguo = PrecioAntiguo;
    }
}
```

Los eventos de integración se pueden definir a nivel de aplicación de cada microservicio, por lo que se desacoplan de otros microservicios, de forma comparable a cómo se definen los ViewModels en el servidor y el cliente. Lo que no se recomienda es compartir una biblioteca de eventos de integración común entre múltiples microservicios; hacerlo sería acoplar esos microservicios con una única biblioteca de datos de definición de eventos. No desea hacer eso por las mismas razones por las que no desea compartir un modelo de dominio común entre múltiples microservicios: los microservicios deben ser completamente autónomos.

Solo hay unos pocos tipos de bibliotecas que debe compartir entre microservicios. Una son las bibliotecas que son bloques de aplicaciones finales, como la [API de cliente de Event Bus](#), como en eShopOnContainers. Otro son las bibliotecas que constituyen herramientas que también podrían compartirse como componentes NuGet, como los serializadores JSON.

## el autobús del evento

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

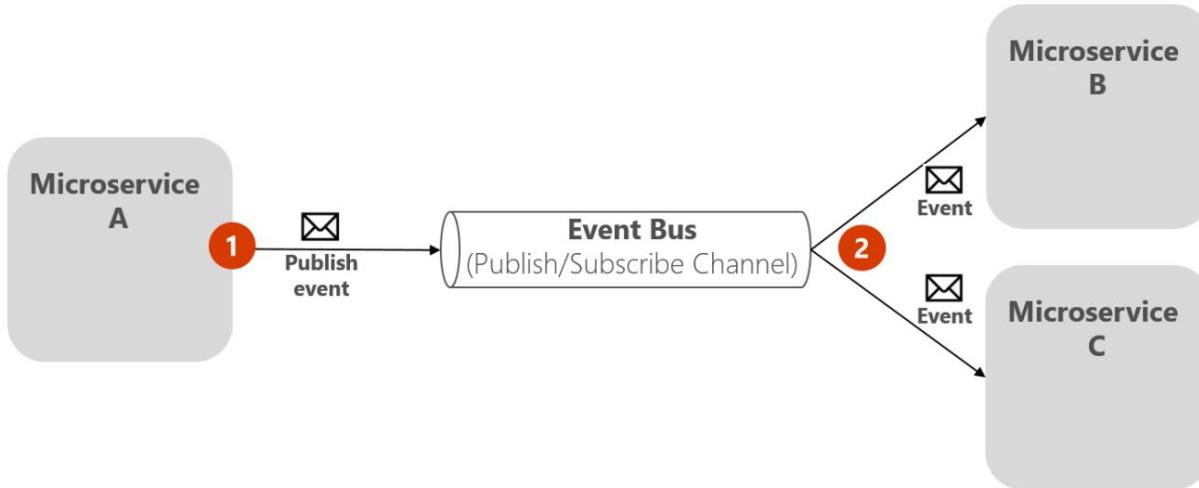


Figura 6-19. Conceptos básicos de publicación/suscripción con un bus de eventos

El diagrama anterior muestra que el microservicio A publica en Event Bus, que distribuye a los microservicios B y C suscritos, sin que el editor necesite conocer a los suscriptores. El bus de eventos está relacionado con el patrón Observer y el patrón de publicación-suscripción.

### Patrón de observador

En el [patrón Observer](#), su objeto principal (conocido como Observable) notifica a otros objetos interesados (conocidos como Observers) con información relevante (eventos).

### Patrón de publicación/suscripción (Pub/Sub)

El propósito del [patrón Publicar/Suscribir](#) es el mismo que el patrón Observador: desea notificar a otros servicios cuando se produzcan ciertos eventos. Pero hay una diferencia importante entre los patrones Observer y Pub/Sub. En el patrón de observador, la transmisión se realiza directamente desde el observable a los observadores, para que se “conozcan” entre sí. Pero cuando se usa un patrón Pub/Sub, hay un tercer componente, llamado intermediario, intermediario de mensajes o bus de eventos, que es conocido tanto por el editor como por el suscriptor. Por lo tanto, al usar el patrón Pub/Sub, el editor y los suscriptores se desacoplan precisamente gracias al mencionado bus de eventos o intermediario de mensajes.

### El intermediario o bus de eventos

¿Cómo se logra el anonimato entre el editor y el suscriptor? Una manera fácil es dejar que un intermediario se encargue de toda la comunicación. Un autobús de eventos es uno de esos intermediarios.

Un bus de eventos normalmente se compone de dos partes:

- La abstracción o interfaz.
- Una o más 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 Pub/Sub. La forma de implementar esta comunicación asíncrona puede variar. Puede tener múltiples implementaciones para que pueda cambiar entre ellas, según 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 múltiples implementaciones basadas en tecnologías de mensajería de infraestructura como RabbitMQ, Azure Service Bus u otro agente de eventos/mensajes.

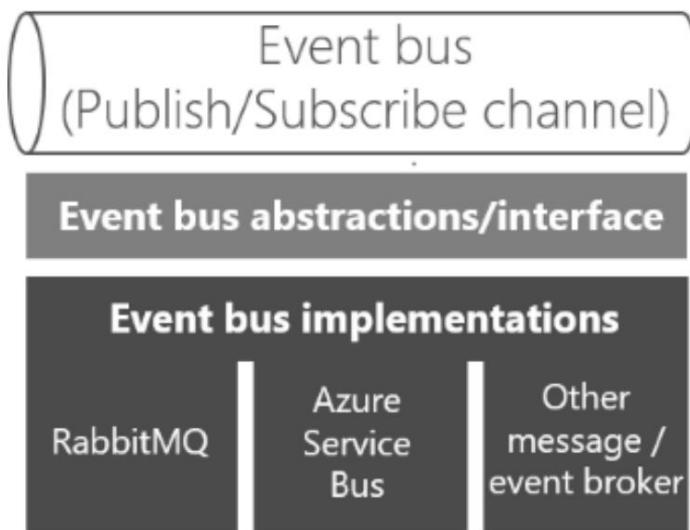


Figura 6- 20. Múltiples implementaciones de un bus de eventos

Es bueno tener el bus de eventos definido a través de una interfaz para que pueda implementarse con varias tecnologías, como RabbitMQ, Azure Service bus u otras. Sin embargo, y como se mencionó anteriormente, usar sus propias abstracciones (la interfaz del bus de eventos) es bueno solo si necesita funciones básicas del bus de eventos compatibles con sus abstracciones. Si necesita funciones de bus de servicio más ricas, probablemente debería usar la API y las abstracciones proporcionadas por su bus de servicio comercial preferido en lugar de sus propias abstracciones.

#### Definición de una interfaz de bus de eventos

Comencemos con un poco de código de implementación para la interfaz del bus de eventos y posibles implementaciones con fines de exploración. La interfaz debe ser genérica y directa, como en la siguiente interfaz.

```

interfaz pública IEventBus {
    void Publicar(IntegrationEvent @event);

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

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

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

    void Cancelar suscripción<T,
        TH>() donde TH: IIntegrationEventHandler<T>
  
```

```

    }
    donde T : EventoIntegración;
}

```

El método de publicación es sencillo. El bus de eventos transmitirá el evento de integración que se le pasó a cualquier microservicio, o incluso a una aplicación externa, suscrita a ese evento. Este método lo utiliza el microservicio que publica el evento.

Los **métodos Subscribe** (puedes tener varias implementaciones dependiendo de los argumentos) son usados por los microservicios que quieren recibir eventos. Este método tiene dos argumentos. El primero es el evento de integración al que suscribirse (`IntegrationEvent`). El segundo argumento es el controlador de eventos de integración (o método de devolución de llamada), llamado `IIntegrationEventHandler<T>`, que se ejecutará cuando el microservicio receptor reciba ese mensaje de evento de integración.

## Recursos adicionales

Algunas soluciones de mensajería listas para producción:

- Autobús de servicio de Azure  
<https://docs.microsoft.com/azure/service-bus-messaging/>
- NServiceBus  
<https://particular.net/nservicebus>
- Tránsito masivo  
<https://masstransit-project.com/>

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

Deberíamos comenzar diciendo que si crea su bus de eventos personalizado basado en RabbitMQ que se ejecuta en un contenedor, como lo hace la aplicación eShopOnContainers, debe usarse solo para sus entornos de desarrollo y prueba. No lo use para su entorno de producción, a menos que lo esté construyendo como parte de un bus de servicio listo para producción. Es posible que a un bus de eventos personalizado simple le falten muchas características críticas listas para producción que tiene un bus de servicio comercial.

Una de las implementaciones personalizadas del bus de eventos en eShopOnContainers es básicamente una biblioteca que utiliza la API 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, publiquen eventos y reciban eventos, como se muestra en la Figura 6-21.

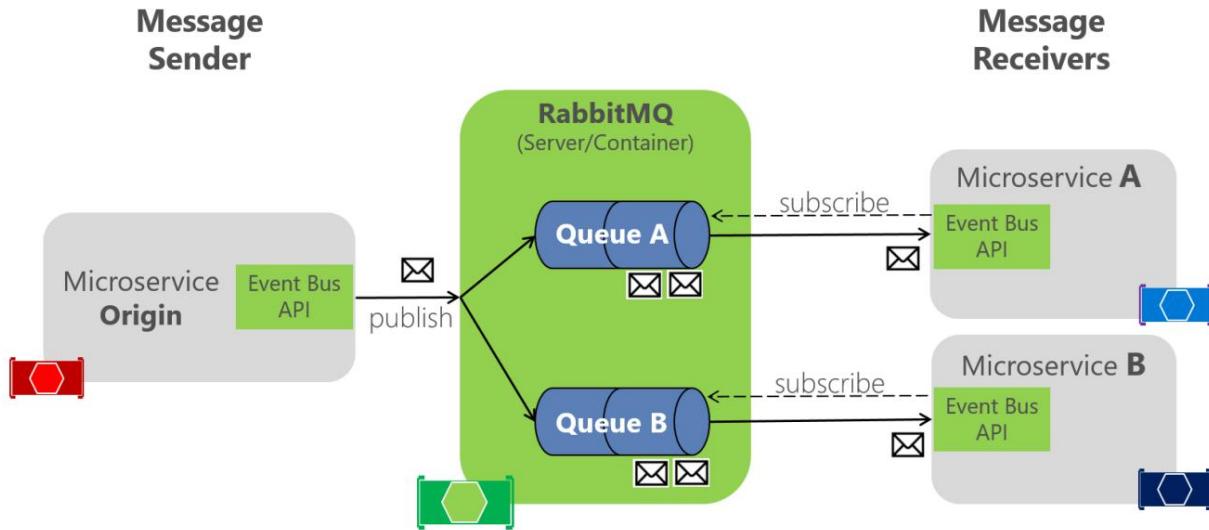


Figura 6-21. Implementación RabbitMQ de un bus de eventos

RabbitMQ funciona como intermediario entre el editor de mensajes y los suscriptores, para manejar la distribución.

En el código, la clase EventBusRabbitMQ implementa la interfaz genérica IEventBus.

Esta implementación se basa en la inyección de dependencia para que pueda cambiar de esta versión de desarrollo/prueba a una versión de producción.

```
clase pública EventBusRabbitMQ: IEventBus, IDisposable {
    // Implementación usando la API de RabbitMQ //...
}
```

La implementación de RabbitMQ de un bus de eventos de prueba/desarrollo muestra es un código repetitivo. Tiene que manejar la conexión con el servidor RabbitMQ y proporcionar código para publicar un evento de mensaje en las colas.

También tiene que implementar un diccionario de colecciones de controladores de eventos de integración para cada tipo de evento; estos tipos de eventos pueden tener una instancia diferente y diferentes suscripciones para cada microservicio receptor, como se muestra en la Figura 6-21.

## Implementando un método de publicación simple con RabbitMQ

El siguiente código es una versión simplificada de una implementación de bus de eventos para RabbitMQ, para mostrar todo el escenario. Realmente no manejas la conexión de esta manera. Para ver la implementación completa, consulte el código real en el repositorio [dotnet-architecture/eShopOnContainers](#).

```
clase pública EventBusRabbitMQ: IEventBus, IDisposable {
    // Objetos miembros y otros métodos... // ...

    public void Publish(IntegrationEvent @event) {
        var eventName = @event.GetType().Name; var
        factory = new ConnectionFactory() { HostName = _connectionString }; usando (var conexión =
        factory.CreateConnection()) usando (var canal = conexión.CreateModel())
```

```

    {
        channel.ExchangeDeclare(intercambio: _brokerName, tipo:
            "directo");
        mensaje de cadena = JsonConvert.SerializeObject(@event); var
        body = Codificación.UTF8.GetBytes(mensaje);
        channel.BasicPublish(intercambio: _brokerName,
            clave de enrutamiento:
            eventName, basicProperties:
            nulo, cuerpo: cuerpo);
    }
}
}

```

El [código real](#) del método Publish en la aplicación eShopOnContainers se mejora mediante el uso de una política de reintento de [Polly](#), que vuelve a intentar la tarea algunas veces en caso de que el contenedor RabbitMQ no esté listo. Este escenario puede ocurrir cuando docker-compose inicia los contenedores; por ejemplo, el contenedor RabbitMQ podría iniciarse más lentamente que los demás contenedores.

Como se mencionó anteriormente, hay muchas configuraciones posibles en RabbitMQ, por lo que este código debe usarse solo para entornos de desarrollo/prueba.

## Implementando el código de suscripción con la API de RabbitMQ

Al igual que con el código de publicación, el siguiente código es una simplificación de parte de la implementación del bus de eventos para RabbitMQ. Una vez más, normalmente no es necesario cambiarlo a menos que lo esté mejorando.

```

clase pública EventBusRabbitMQ: IEventBus, IDisposable {

    // Objetos miembros y otros métodos... // ...

    public void Subscribe<T, TH>() where
        T : IntegrationEvent where TH :
        IIIntegrationEventHandler<T>
    {
        var eventName = _subsManager.GetEventKey<T>();

        var containsKey = _subsManager.HasSubscriptionsForEvent(eventName); si (! contiene
        clave) {

            if (! _persistentConnection.IsConnected) {

                _persistentConnection.TryConnect();
            }

            usando (var canal = _persistentConnection.CreateModel()) {

                channel.QueueBind(cola: _queueName,
                    intercambio: BROKER_NAME,
                    routingKey: eventName);
            }
        }

        _subsManager.AddSubscription<T, TH>();
    }
}

```

Cada tipo de evento tiene un canal relacionado para obtener eventos de RabbitMQ. Luego puede tener tantos controladores de eventos por canal y tipo de evento como sea necesario.

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

Como se mencionó anteriormente, el bus de eventos implementado en `eShopOnContainers` solo tiene un propósito educativo, ya que solo maneja los escenarios principales, por lo que no está listo para la producción.

Para escenarios de producción, consulte los recursos adicionales a continuación, específicos para RabbitMQ, y la sección [Implementación de comunicación basada en eventos entre microservicios](#).

## Recursos adicionales

Una solución lista para producción con soporte para RabbitMQ.

- EasyNetQ : cliente API .NET de código abierto para RabbitMQ  
<https://easynetq.com/>
- Tránsito masivo  
<https://masstransit-project.com/>
- Rebus - Bus de servicio .NET de código abierto <https://github.com/rebus-org/Rebus>

## Suscripción a eventos

El primer paso para usar el bus de eventos es suscribir los microservicios a los eventos que desean recibir. Esta funcionalidad debe hacerse en los microservicios del receptor.

El siguiente código simple muestra lo que necesita implementar cada microservicio receptor 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` debe suscribirse a los mensajes `ProductPriceChangedIntegrationEvent` y `OrderStartedIntegrationEvent`.

Por ejemplo, cuando se suscribe al evento `ProductPriceChangedIntegrationEvent`, eso hace que el microservicio de la cesta sea consciente de cualquier cambio en el precio del producto y le permite advertir al usuario sobre el cambio si ese producto está en la cesta del usuario.

```
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();

eventBus.Subscribe<ProductPriceChangedIntegrationEvent,
    ProductPriceChangedIntegrationEventHandler>();

eventBus.Subscribe<OrderStartedIntegrationEvent,
    OrderStartedIntegrationEventHandler>();
```

Después de que se ejecute este código, el microservicio del suscriptor escuchará a través de los canales de RabbitMQ. Cuando llega cualquier 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

Finalmente, el remitente del mensaje (microservicio de origen) publica los eventos de integración con un código similar al siguiente ejemplo. (Este enfoque es un ejemplo simplificado que no tiene en cuenta la atomicidad). Implementaría un código similar cada vez que un evento deba propagarse a través de múltiples microservicios, generalmente justo después de confirmar datos o transacciones desde el microservicio de origen.

Primero, el objeto de implementación del bus de eventos (basado en RabbitMQ o basado en un bus de servicio) se inyectaría en el constructor del controlador, como en el siguiente código:

```
[Ruta("api/v1/[controlador]")] clase
pública CatalogController : ControllerBase {

    privado de solo lectura CatalogContext _context; privado
    de solo lectura IOptionsSnapshot<Configuración> _configuración; privado
    de solo lectura IEventBus _eventBus;

    CatalogController público ( contexto CatalogContext ,
        Configuración de IOptionsSnapshot<Configuración>,
        IEventBus eventBus)
    {
        _contexto = contexto;
        _configuraciones =
            configuraciones; _eventBus = eventoBus;

    } // ...
}
```

Luego lo usa desde los métodos de su controlador, como en el método UpdateProduct:

```
[Ruta("elementos")]
[HttpPost]
tarea asíncrona pública <ActionResult> UpdateProduct ([FromBody]CatalogItem producto) {

    var item = await _context.CatalogItems.SingleOrDefaultAsync( i => i.Id ==
        producto.Id); // ... if (artículo.Precio != producto.Precio) {

        var PrecioAntiguo = artículo.Precio;
        artículo.Precio = producto.Precio;
        _context.CatalogItems.Update(elemento); var
        @event = new ProductPriceChangedIntegrationEvent(item.Id, item.Price, oldPrice); //
        Confirmar cambios en la transacción original aguardar
        _context.SaveChangesAsync(); // Publicar evento de integración en el bus de
        eventos // (RabbitMQ o un bus de servicio debajo) _eventBus.Publish(@event); //

    }

}
```

```
// ...
```

En este caso, dado que el microservicio de origen es un microservicio CRUD simple, ese código se coloca directamente en un controlador de API web.

En microservicios más avanzados, como cuando se usan enfoques CQRS, se puede implementar en la clase `CommandHandler`, dentro del método `Handle()`.

## Diseño de atomicidad y resiliencia al publicar en el bus de eventos

Cuando publica eventos de integración a través de un sistema de mensajería distribuida como su bus de eventos, tiene el problema de actualizar atómicamente la base de datos original y publicar un evento (es decir, ambas operaciones se completan o ninguna). Por ejemplo, en el ejemplo simplificado que se mostró anteriormente, el código envía datos a la base de datos cuando se cambia el precio del producto y luego publica un mensaje `ProductPriceChangedIntegrationEvent`. Inicialmente, podría parecer esencial que estas dos operaciones se realicen atómicamente. Sin embargo, si usa una transacción distribuida que involucra la base de datos y el intermediario de mensajes, como lo hace en sistemas más antiguos como [Microsoft Message Queuing \(MSMQ\)](#), no se recomienda este enfoque por las razones descritas por el [teorema CAP](#).

---

Básicamente, utiliza microservicios para crear sistemas escalables y de alta disponibilidad. Simplificando un poco, el teorema CAP dice que no se puede construir una base de datos (distribuida) (o un microservicio que posea su modelo) que esté continuamente disponible, fuertemente consistente y 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 debe restar importancia a la consistencia sólida. Por lo tanto, en la mayoría de las aplicaciones modernas basadas en microservicios, normalmente no desea usar transacciones distribuidas en la mensajería, como lo hace cuando implementa [transacciones distribuidas](#) basadas en el [Coordinador de transacciones distribuidas \(DTC\)](#) de Windows con [MSMQ](#).

---

Volvamos al tema inicial y su ejemplo. Si el servicio falla después de actualizar la base de datos (en este caso, justo después de la línea de código con `_context.SaveChangesAsync()`), pero antes de que se publique el evento de integración, el sistema general podría volverse inconsistente. Este enfoque puede ser crítico para el negocio, dependiendo de la operación comercial específica con la que esté tratando.

Como se mencionó anteriormente en la sección de arquitectura, puede tener varios enfoques para tratar este problema:

- Usando el [patrón de origen de eventos completo](#).
- Uso de la minería de registros de transacciones.
- Usando el [patrón de Bandeja de salida](#). Esta es una tabla transaccional para almacenar los eventos de integración (extendiendo la transacción local).

Para este escenario, usar el patrón de origen de eventos (ES) completo es uno de los mejores enfoques, si no el mejor. Sin embargo, en muchos escenarios de aplicación, es posible que no pueda implementar un sistema ES completo. ES significa almacenar solo eventos de dominio en su base de datos transaccional, en lugar de almacenar datos de estado actual. Almacenar solo eventos de dominio puede tener grandes beneficios, como tener disponible el historial de su sistema y poder determinar el estado de su sistema en cualquier momento en el pasado. Sin embargo,

implementar un sistema ES completo requiere que usted rediseñe la mayor parte de su sistema e introduce muchas otras complejidades y requisitos. Por ejemplo, le gustaría usar una base de datos creada específicamente para el abastecimiento de eventos, como [Event Store](#), o una base de datos orientada a documentos como Azure Cosmos DB, MongoDB, Cassandra, CouchDB o RavenDB. ES es un excelente enfoque para este problema, pero no es la solución más fácil a menos que ya esté familiarizado con el abastecimiento de eventos.

La opción de utilizar la minería de registros de transacciones inicialmente parece transparente. Sin embargo, para usar este enfoque, el microservicio debe estar acoplado a su registro de transacciones RDBMS, como el registro de transacciones de SQL Server. Este enfoque probablemente no sea deseable. Otro inconveniente es que las actualizaciones de bajo nivel registradas en el registro de transacciones pueden no estar al mismo nivel que los eventos de integración de alto nivel. Si es así, el proceso de ingeniería inversa de esas operaciones de registro de transacciones puede ser difícil.

Un enfoque equilibrado es una combinación de una tabla de base de datos transaccional y un patrón ES simplificado. Puede usar un estado como "listo para publicar el evento", que establece en el evento original cuando lo confirma en la tabla de eventos de integración. A continuación, intenta publicar el evento en el bus de eventos. Si la acción de publicación del evento tiene éxito, inicia otra transacción en el servicio de origen y cambia el estado de "listo para publicar el evento" a "evento ya publicado".

Si falla la acción de publicación de eventos en el bus de eventos, los datos seguirán sin ser incoherentes dentro del microservicio de origen; seguirán marcados como "listos para publicar el evento" y, con respecto al resto de los servicios, eventualmente se consistente. Siempre puede tener trabajos en segundo plano que verifiquen 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 publicar ese evento en el bus de eventos.

Tenga en cuenta que con este enfoque, solo conserva los eventos de integración para cada microservicio de origen y solo los eventos que desea comunicar a otros microservicios o sistemas externos. Por el contrario, en un sistema ES completo, también almacena todos los eventos del dominio.

Por lo tanto, este enfoque equilibrado es un sistema ES simplificado. Necesita una lista de eventos de integración con su estado actual ("listo para publicar" frente a "publicado"). Pero solo necesita implementar estos estados para los eventos de integración. Y en este enfoque, no necesita almacenar todos los datos de su dominio como eventos en la base de datos transaccional, como lo haría en un sistema ES completo.

Si ya está usando una base de datos relacional, puede usar una tabla transaccional para almacenar eventos de integración. Para lograr atomicidad en su aplicación, utiliza un proceso de dos pasos basado en transacciones locales. Básicamente, tiene una tabla `IntegrationEvent` en la misma base de datos donde tiene sus entidades de dominio. Esa tabla funciona como un seguro para lograr la atomicidad, de modo que incluya eventos de integración persistentes en las mismas transacciones que comprometen los datos de su dominio.

Paso a paso, el proceso es así:

1. La aplicación inicia una transacción de base de datos local.
2. Luego actualiza el estado de las entidades de su dominio e inserta un evento en la tabla de eventos de integración.
3. Finalmente, confirma la transacción, por lo que obtiene la atomicidad deseada y luego
4. Publicas el evento de alguna manera (siguiente).

Al implementar los pasos de publicación de eventos, tiene estas opciones:

- Publique el evento de integración justo después de confirmar la transacción y use otra transacción local para marcar los eventos en la tabla como publicados. Luego, use la tabla solo como un artefacto para rastrear los eventos de integración en caso de problemas en los microservicios remotos y realice acciones compensatorias basadas en los eventos de integración almacenados.
- Usa la tabla como una especie de cola. Un subproceso o proceso de aplicación independiente consulta la tabla de eventos de integración, publica los eventos en el bus de eventos y luego usa una transacción local para marcar los eventos como publicados.

La figura 6-22 muestra la arquitectura del primero de estos enfoques.

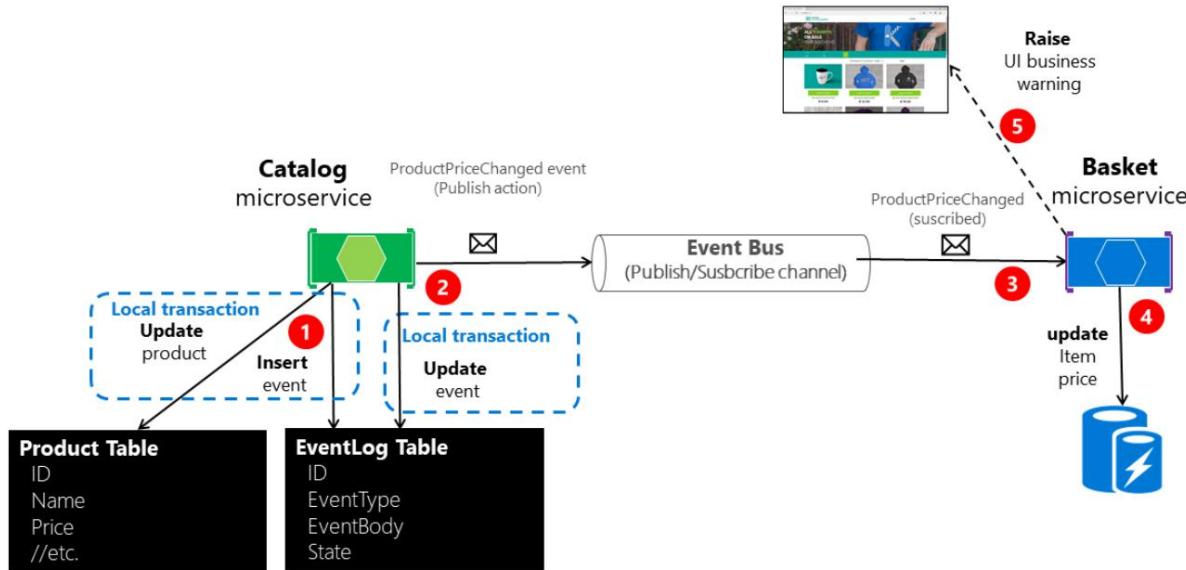


Figura 6-22. Atomicidad al publicar eventos en el bus de eventos

Al enfoque ilustrado en la Figura 6-22 le falta un microservicio de trabajador adicional que se encarga de verificar y confirmar el éxito de los eventos de integración publicados. En caso de falla, ese microservicio de trabajador de verificador adicional puede leer eventos de la tabla y volver a publicarlos, es decir, repetir el paso número 2.

Acerca del segundo enfoque: usa la tabla EventLog como una cola y siempre usa un microservicio de trabajador para publicar los mensajes. En ese caso, el proceso es como el que se muestra en la Figura 6-23. Esto muestra un microservicio adicional y la tabla es la única fuente cuando se publican eventos.

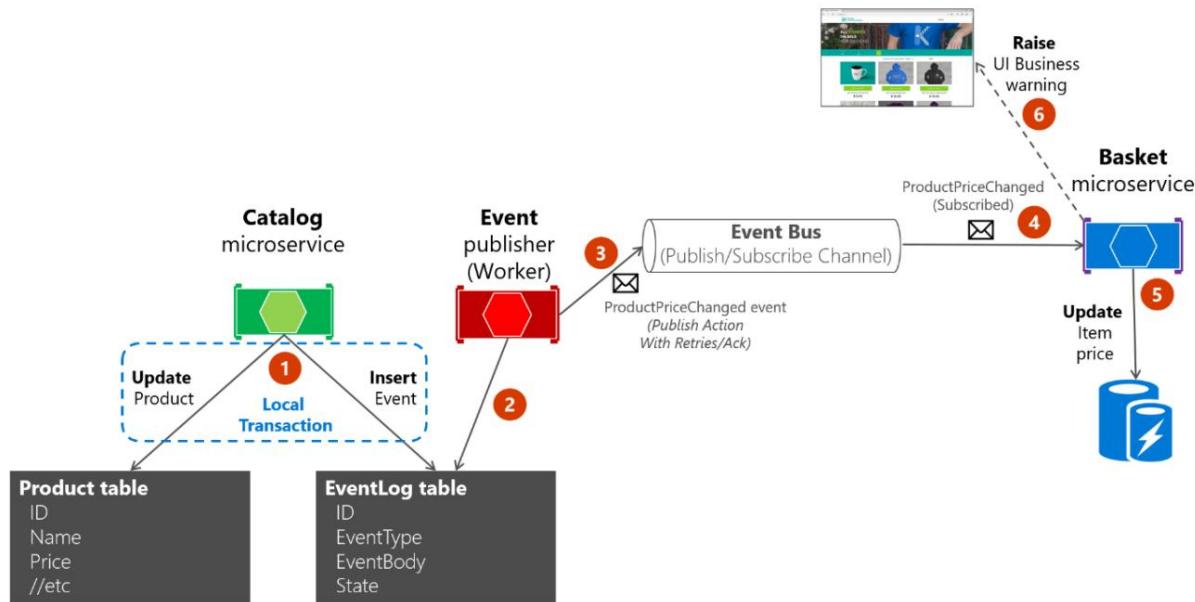


Figura 6-23. Atomicidad al publicar eventos en el bus de eventos con un microservicio de trabajo

Para simplificar, el ejemplo de eShopOnContainers utiliza el primer enfoque (sin procesos adicionales ni microservicios de verificación) más el bus de eventos. Sin embargo, el ejemplo de eShopOnContainers no maneja todos los posibles casos de error. En una aplicación real implementada en la nube, debe aceptar el hecho de que eventualmente surgirán problemas, y debe implementar esa lógica de verificación y reenvío. Usar la tabla como una cola puede ser más efectivo que el primer enfoque si tiene esa tabla como fuente única de eventos al publicarlos (con el trabajador) a través del bus de eventos.

### Implementación de atomicidad al publicar eventos de integración a través del bus de eventos

El siguiente código muestra cómo puede crear una sola transacción que involucre varios objetos DbContext: un contexto relacionado con los datos originales que se actualizan y el segundo contexto relacionado con la tabla IntegrationEventLog.

La transacción en el código de ejemplo a continuación no será resistente si las conexiones a la base de datos tienen algún problema en el momento en que se ejecuta el código. Esto puede suceder en sistemas basados en la nube como Azure SQL DB, que pueden mover bases de datos entre servidores. Para implementar transacciones resilientes en varios contextos, consulte la sección [Implementación de conexiones resilientes de Entity Framework Core SQL](#) más adelante en esta guía.

Para mayor claridad, el siguiente ejemplo muestra todo el proceso en una sola pieza de código. Sin embargo, la implementación de eShopOnContainers se refactoriza y divide esta lógica en varias clases para que sea más fácil de mantener.

```
// Actualizar producto desde el microservicio de catálogo // Public
async Task<IActionResult> UpdateProduct([FromBody]CatalogItem
productToUpdate) { var catalogItem = await _catalogContext.CatalogItems.SingleOrDefaultAsync(i => i.Id ==
```

```

        producto a actualizar.Id);

if (catalogItem == null) return NotFound();

bool aumentarProductoPrecioCambiadoEvento = falso;
Eventointegración precioCambiadoEvento = nulo;

if (catalogItem.Price != productToUpdate.Price)
    raiseProductPriceChangedEvent = true;

if (raiseProductPriceChangedEvent) // Crear evento si el precio ha cambiado {

    var precioantiguo = catalogItem.Precio;
    priceChangedEvent = new ProductPriceChangedIntegrationEvent(catalogItem.Id,
                                                                productToUpdate.Price ,
                                                                oldPrice);

}

// Actualizar producto catalogItem
actual = productToUpdate;

// Simplemente guarde el producto actualizado si el precio del producto no ha cambiado. if (!
raiseProductPriceChangedEvent) {

    esperar _catalogContext.SaveChangesAsync();

} else // Publicar en event bus solo si el precio del producto cambió {

    // Logrando la atomicidad entre la base de datos original y el IntegrationEventLog // con una
    transacción local usando (var transacción = _catalogContext.Database.BeginTransaction()) {

        _catalogContext.CatalogItems.Update(catalogItem); esperar
        _catalogContext.SaveChangesAsync();

        esperar _integrationEventLogService.SaveEventAsync(priceChangedEvent);

        transacción.Commit();
    }

    // Publicar el evento de integración a través del bus de eventos
    _eventBus.Publish(priceChangedEvent);

    _integrationEventLogService.MarkEventAsPublishedAsync(
        precioCambiadoEvento);
}

volver bien();
}

```

Después de que se crea el evento de integración ProductPriceChangedIntegrationEvent, la transacción que almacena la operación de dominio original (actualizar el elemento del catálogo) también incluye la persistencia del evento en la tabla EventLog. Esto lo convierte en una sola transacción y siempre podrá verificar si se enviaron mensajes de eventos.

La tabla de registro de eventos se actualiza atómicamente con la operación de la base de datos original, utilizando una transacción local en la misma base de datos. Si alguna de las operaciones falla, se lanza una excepción y la transacción revierte cualquier operación completada, manteniendo así la coherencia entre las operaciones del dominio y los mensajes de eventos guardados en la tabla.

## Recepción de mensajes de suscripciones: controladores de eventos en microservicios receptores

Además de la lógica de suscripción de eventos, debe implementar el código interno para los controladores de eventos de integración (como un método de devolución de llamada). El controlador de eventos es donde especifica dónde se recibirán y procesarán los mensajes de eventos de un determinado tipo.

Un controlador de eventos primero recibe una instancia de evento del bus de eventos. Luego localiza el componente a procesar relacionado con ese evento de integración, propagando y persistiendo el evento como un cambio de estado en el microservicio receptor. Por ejemplo, si un evento ProductPriceChanged se origina en el microservicio del catálogo, se maneja en el microservicio de la cesta y también cambia el estado en este microservicio de la cesta del receptor, como se muestra en el siguiente código.

```
espacio de nombres Microsoft.eShopOnContainers.Services.Basket.API.IntegrationEvents.EventHandling {

    clase pública ProductPriceChangedIntegrationEventHandler :
        IIntegrationEventHandler<ProductPriceChangedIntegrationEvent>
    {
        IBasketRepository privado de solo lectura _repository;

        Public ProductPriceChangedIntegrationEventHandler(
            repositorio IBasketRepository)
        {
            _repositorio = repositorio;
        }

        Manejador de tareas asíncronas públicas (ProductPriceChangedIntegrationEvent @event)
        {
            var userIds = esperar _repository.GetUsers(); foreach
            (id de var en ID de usuario)

                var cesta = esperar _repository.GetBasket(id); await
                UpdatePricelnBasketItems(@event.ProductId, @event.NewPrice, basket);
            }
        }

        Tarea asincrónica privada UpdatePricelnBasketItems (int productId, decimal newPrice,
        Cesta de la cesta del cliente)
        {
            var itemsToUpdate = basket?.Items?.Where(x => int.Parse(x.ProductId) == productId).ToList();
            if (itemsToUpdate != null)

                foreach (elemento var en itemsToUpdate)

                    if(artículo.PrecioUnitario != nuevoPrecio)
                    {
                        var PrecioOriginal = artículo.PrecioUnitario;
                        item.UnitPrice = nuevoPrecio;
                        artículo.PrecioUnitarioAntiguo = PrecioOriginal ;
                    }

                    } await _repository.UpdateBasket(cesta);
            }
        }
    }
}
```

El controlador de eventos debe verificar si el producto existe en alguna de las instancias de la cesta. También actualiza el precio del artículo para cada artículo de la cesta relacionado. Finalmente, crea una alerta para mostrar al usuario sobre el cambio de precio, como se muestra en la Figura 6-24.

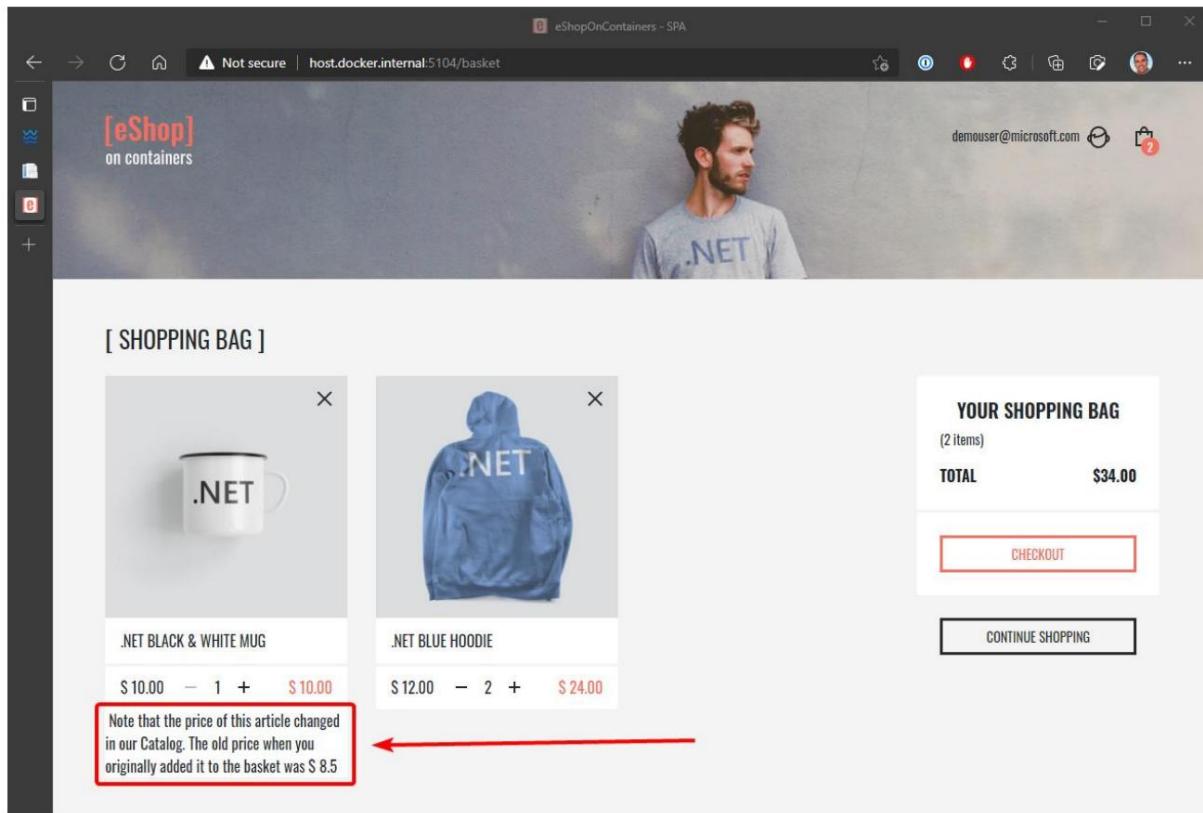


Figura 6-24. Mostrar el cambio de precio de un artículo en una cesta, tal como lo comunican los eventos de integración

### Idempotencia en eventos de mensajes de actualización

Un aspecto importante de los eventos de mensajes de actualización es que una falla en cualquier punto de la comunicación debe hacer que se vuelva a intentar enviar el mensaje. De lo contrario, una tarea en segundo plano podría intentar publicar un evento que ya se ha publicado, creando una condición de carrera. Asegúrese de que las actualizaciones sean idempotentes o que brinden suficiente información para garantizar que pueda detectar un duplicado, descartarlo y enviar solo una respuesta.

Como se señaló anteriormente, la idempotencia significa que una operación se puede realizar varias veces sin cambiar el resultado. En un entorno de mensajería, como cuando se comunican eventos, un evento es idempotente si se puede entregar varias veces sin cambiar el resultado para el microservicio del receptor. Esto puede ser necesario debido a la naturaleza del evento mismo o a la forma en que el sistema maneja el evento. La idempotencia de mensajes es importante en cualquier aplicación que use mensajería, no solo en 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 aún no están en la tabla. No importa cuántas veces ejecute esa instrucción SQL de inserción; el resultado será el mismo: la tabla contendrá esos datos. Una idempotencia como esta también puede ser necesaria cuando se trata de mensajes si los mensajes podrían enviarse potencialmente y, por lo tanto,

procesado más de una vez. Por ejemplo, si la lógica de reintento hace que un remitente envíe exactamente el mismo mensaje más de una vez, debe asegurarse de que sea idempotente.

Es posible diseñar mensajes idempotentes. Por ejemplo, puede crear un evento que diga "establecer el precio del producto en \$25" en lugar de "agregar \$5 al precio del producto". Puede procesar con seguridad el primer mensaje cualquier número de veces y el resultado será el mismo. Eso no es cierto para el segundo mensaje. Pero incluso en el primer caso, es posible que no desee procesar el primer evento, porque el sistema también podría haber enviado un evento de cambio de precio más nuevo y estaría sobrescribiendo el nuevo precio.

Otro ejemplo podría ser un evento de pedido completado que se propaga a varios suscriptores. La aplicación debe asegurarse de que la información del pedido se actualice en otros sistemas solo una vez, incluso si hay eventos de mensajes duplicados para el mismo evento de pedido completado.

Es conveniente tener algún tipo de identidad por evento para poder crear una lógica que obligue a que cada evento se procese solo una vez por receptor.

Parte del procesamiento de mensajes es inherentemente idempotente. Por ejemplo, si un sistema genera miniaturas de imágenes, puede que no importe cuántas veces se procese el mensaje sobre la miniatura generada; el resultado es que se generan las miniaturas y son las mismas cada vez. Por otro lado, operaciones como llamar a una pasarela de pago para cargar una tarjeta de crédito pueden no ser en absoluto idempotentes. En estos casos, debe asegurarse de que procesar un mensaje varias veces tenga el efecto esperado.

## Recursos adicionales

- Respetar la idempotencia [de mensajes](https://docs.microsoft.com/previous-versions/msp-np/jj591565(v=pandp.10)#honoring message-idempotency)  
[https://docs.microsoft.com/previous-versions/msp-np/jj591565\(v=pandp.10\)#honoring message-idempotency](https://docs.microsoft.com/previous-versions/msp-np/jj591565(v=pandp.10)#honoring message-idempotency)

## Deduplicación de mensajes de eventos de integración

Puede asegurarse de que los eventos de mensajes se envíen y procesen solo una vez por suscriptor en diferentes niveles. Una forma es utilizar una función de deduplicación que ofrece la infraestructura de mensajería que está utilizando. Otra es implementar una lógica personalizada en su microservicio de destino. Tener validaciones tanto a nivel de transporte como a nivel de aplicación es su mejor opción.

## Deduplicación de eventos de mensajes en el nivel EventHandler

Una forma de asegurarse de que cualquier receptor procese un evento solo una vez es implementar cierta lógica al procesar los eventos del mensaje en los controladores de eventos. Por ejemplo, ese es el enfoque utilizado en la aplicación eShopOnContainers, como puede ver en el [código fuente de la clase UserCheckoutAcceptedIntegrationEventHandler](#) cuando recibe un evento de integración

UserCheckoutAcceptedIntegrationEvent. (En este caso, CreateOrderCommand se envuelve con IdentifiedCommand, utilizando eventMsg.RequestId como identificador, antes de enviarlo al controlador de comandos).

## Deduplicación de mensajes al usar RabbitMQ

Cuando ocurren fallas intermitentes en la red, los mensajes se pueden duplicar y el receptor del mensaje debe estar listo para manejar estos mensajes duplicados. Si es posible, los receptores deben manejar los mensajes de manera idempotente, lo cual es mejor que manejarlos explícitamente con deduplicación.

De acuerdo con la [documentación de RabbitMQ](#), "Si un mensaje se entrega a un consumidor y luego se vuelve a poner en cola (porque no se reconoció antes de que se interrumpiera la conexión del consumidor, por ejemplo), RabbitMQ establecerá el indicador de reenvío cuando se entregue nuevamente (ya sea para el mismo consumidor o uno diferente).

Si se establece el indicador "reenviado", el receptor debe tenerlo en cuenta, porque es posible que el mensaje ya se haya procesado. Pero eso no está garantizado; Es posible que el mensaje nunca haya llegado al receptor después de dejar el intermediario de mensajes, quizás debido a problemas de red. Por otro lado, si no se establece el indicador de "reenviado", se garantiza que el mensaje no se ha enviado más de una vez.

Por lo tanto, el receptor necesita deduplicar mensajes o procesar mensajes de manera idempotente solo si el indicador "reenviado" está establecido en el mensaje.

## Recursos adicionales

- eShopOnContainers bifurcados usando NServiceBus (software particular) <https://go.particular.net/eShopOnContainers>
- Mensajería basada en eventos [https://patterns.arcitura.com/soa-patterns/design\\_patterns/event\\_driven\\_messaging](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-suscripción <https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- 
- Comunicación entre contextos acotados [https://docs.microsoft.com/versiones-anteriores/msp-np/j591572\(v=pandp.10\)](https://docs.microsoft.com/versiones-anteriores/msp-np/j591572(v=pandp.10))
- Coherencia eventual [https://en.wikipedia.org/wiki/Eventual\\_consistency](https://en.wikipedia.org/wiki/Eventual_consistency)
- Felipe Brown. Estrategias para integrar contextos acotados <https://www.culttt.com/2014/11/26/strategies-integrating-bounded-contexts/>
- Chris Richardson. Desarrollo de microservicios transaccionales mediante agregados, abastecimiento de eventos y CQRS - Parte 2 <https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson>
- Chris Richardson. Patrón de abastecimiento de eventos <https://microservices.io/patterns/data/event-sourcing.html>
- Presentamos el abastecimiento de eventos [https://docs.microsoft.com/previous-versions/msp-np/j591559\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-np/j591559(v=pandp.10))

- Base de datos de la tienda de eventos. Sitio oficial.  
<https://geteventstore.com/>
- Patricio Nommensen. Gestión de datos basada en eventos para microservicios <https://dzone.com/articles/event-driven-data-management-for-microservices-1>
- El teorema de la PAC  
[https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem)
- ¿Qué es el Teorema CAP? <https://www.quora.com/What-Is-CAP-Theorem-1>
- Introducción a la coherencia de datos [https://docs.microsoft.com/previous-versions/msp-np/dn589800\(v=pandp.10\)](https://docs.microsoft.com/previous-versions/msp-np/dn589800(v=pandp.10))
- Rick Saling. El teorema CAP: por qué "todo es diferente" con la nube e Internet <https://docs.microsoft.com/archive/blogs/rickatmicrosoft/the-cap-theorem-why-everything-is-different-with-the-cloud- e-internet/>
- Eric Brewer. CAP Doce años después: cómo han cambiado las "reglas" <https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>
- Bus de servicio de Azure. Mensajería intermediada: Detección de duplicados  
[https://github.com/microsoftarchive/msdn-code-gallery\\_microsoft/tree/master/Windows%20Azure%20Product%20Team/Brokered%20Messaging%20Duplicate%20Detection](https://github.com/microsoftarchive/msdn-code-gallery_microsoft/tree/master/Windows%20Azure%20Product%20Team/Brokered%20Messaging%20Duplicate%20Detection)
- Guía de confiabilidad (documentación de RabbitMQ) <https://www.rabbitmq.com/reliability.html#consumer>

## Prueba de servicios ASP.NET Core y aplicaciones web

Los controladores son una parte central de cualquier servicio ASP.NET Core API y aplicación web ASP.NET MVC. Como tal, debe estar seguro de que se comportan según lo previsto para su aplicación. Las pruebas automatizadas pueden brindarle esta confianza y pueden detectar errores antes de que lleguen a la producción.

Debe probar cómo se comporta el controlador en función de las entradas válidas o no válidas, y probar las respuestas del controlador en función del resultado de la operación comercial que realiza. Sin embargo, debe tener estos tipos de pruebas para sus microservicios:

- Pruebas unitarias. Estas pruebas aseguran que los componentes individuales de la aplicación funcionen como se espera. Las aserciones prueban la API del componente.
- Pruebas de integración. Estas pruebas aseguran que las interacciones de los componentes funcionen según lo esperado frente a artefactos externos como bases de datos. Las aserciones pueden probar la API del componente, la interfaz de usuario o los efectos secundarios de acciones como la E/S de la base de datos, el registro, etc.
- Pruebas funcionales para cada microservicio. Estas pruebas aseguran que la aplicación funcione como se espera desde la perspectiva del usuario.

- Pruebas de servicio. Estas pruebas aseguran que se prueben los casos de uso de servicios de extremo a extremo, incluida la prueba de múltiples servicios al mismo tiempo. Para este tipo de prueba, primero debe preparar el entorno. En este caso, significa iniciar los servicios (por ejemplo, usando docker compose up).

## Implementación de pruebas unitarias para ASP.NET Core Web API

Las pruebas unitarias implican probar una parte de una aplicación de forma aislada de su infraestructura y dependencias.

Cuando realiza una prueba unitaria de la lógica del controlador, solo se prueba el contenido de una sola acción o método, no el comportamiento de sus dependencias o del propio marco. Las pruebas unitarias no detectan problemas en la interacción entre componentes, ese es el propósito de las pruebas de integración.

Mientras prueba las acciones de su controlador, asegúrese de concentrarse solo en su comportamiento. Una prueba de unidad de controlador evita cosas como filtros, enrutamiento o vinculación de modelos (la asignación de datos de solicitud a un ViewModel o DTO). Debido a que se enfocan en probar solo una cosa, las pruebas unitarias generalmente son simples de escribir y rápidas de ejecutar. Un conjunto bien escrito de pruebas unitarias se puede ejecutar con frecuencia sin mucha sobrecarga.

Las pruebas unitarias se implementan en base a marcos de prueba como xUnit.net, MSTest, Moq o NUnit. Para la aplicación de muestra eShopOnContainers, estamos usando xUnit.

Cuando escribe una prueba de unidad para un controlador de API web, crea una instancia de la clase de controlador directamente usando la nueva palabra clave en C#, de modo que la prueba se ejecute lo más rápido posible. El siguiente ejemplo muestra cómo hacer esto cuando se usa [xUnit](#) como marco de prueba.

```
[Hecho]
Tarea asíncrona pública Get_order_detail_success() {

    //Organizar
    var fakeOrderId = "12"; var
    pedidofalso = ObtenerPedidoFalso();

    //...
    //Act
    var orderController = new
        OrderController( _orderServiceMock.Object,
        _basketServiceMock.Object,
        _identityParserMock.Object);

    orderController.ControllerContext.HttpContext = _contextMock.Object; var actionResult =
    esperar orderController.Detail(fakeOrderId);

    // Afirmar
    var viewResult = Assert.IsType<ViewResult>(actionResult);
    Assert.IsAssignableFrom<Order>(viewResult.ViewData.Model);
}
```

## Implementación de pruebas de integración y funcionales para cada microservicio.

Como se señaló, las pruebas de integración y las pruebas funcionales tienen diferentes propósitos y objetivos. Sin embargo, la forma en que implementa ambos cuando prueba los controladores ASP.NET Core es similar, por lo que en esta sección nos concentraremos en las pruebas de integración.

Las pruebas de integración aseguran que los componentes de una aplicación funcionen correctamente cuando se ensamblan.

ASP.NET Core es compatible con las pruebas de integración mediante marcos de pruebas unitarias y un host web de prueba integrado que se puede usar para manejar solicitudes sin sobrecarga de la red.

A diferencia de las pruebas unitarias, las pruebas de integración con frecuencia involucran problemas de infraestructura de aplicaciones, como una base de datos, un sistema de archivos, recursos de red o solicitudes y respuestas web. Las pruebas unitarias utilizan objetos falsos o simulados en lugar de estas preocupaciones. Pero el propósito de las pruebas de integración es confirmar que el sistema funciona como se espera con estos sistemas, por lo que para las pruebas de integración no utiliza objetos falsos ni ficticios. En su lugar, incluye la infraestructura, como el acceso a la base de datos o la invocación de servicios desde otros servicios.

Debido a que las pruebas de integración utilizan segmentos de código más grandes que las pruebas unitarias y debido a que las pruebas de integración se basan en elementos de infraestructura, tienden a ser mucho más lentas que las pruebas unitarias. Por lo tanto, es una buena idea limitar la cantidad de pruebas de integración que escribe y ejecuta.

ASP.NET Core incluye un servidor web de prueba integrado que se puede usar para manejar solicitudes HTTP sin sobrecarga de la red, lo que significa que puede ejecutar esas pruebas más rápido que cuando usa un servidor web real. El host web de prueba (TestServer) está disponible en un componente NuGet como Microsoft.AspNetCore.TestHost. Puede agregarse a proyectos de prueba de integración y usarse para hospedar aplicaciones ASP.NET Core.

Como puede ver en el siguiente código, cuando crea pruebas de integración para controladores ASP.NET Core, crea una instancia de los controladores a través del host de prueba. Esta funcionalidad es comparable a una solicitud HTTP, pero se ejecuta más rápido.

```
clase pública PrimeWebDefaultRequestDebe {

    TestServer privado de solo lectura _server;
    privado de solo lectura HttpClient _client;

    Público PrimeWebDefaultRequestDebería () {

        // Organizar
        _server = nuevo TestServer(nuevo WebHostBuilder()
            .UseStartup<Inicio>()); _cliente
        = _servidor.CrearCliente();
    }

    [Hecho]
    tarea asíncrona pública ReturnHelloWorld() {

        // Actuar
        respuesta var = esperar _client.GetAsync("/");
        respuesta.EnsureSuccessStatusCode(); var responseString
        = esperar respuesta.Content.ReadAsStringAsync(); // Afirmar Assert.Equal("Hola
        mundo!", cadena de respuesta);

    }
}
```

## Recursos adicionales

- Steve Smith. Controladores de prueba (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 usando dotnet test  
<https://docs.microsoft.com/dotnet/core/testing/unit-testing-with-dotnet-test>
- xUnit.net. Sitio oficial.  
<https://xunit.net/>
- Fundamentos de las pruebas unitarias.  
<https://docs.microsoft.com/visualstudio/test/unit-test-basics>
- Moq. repositorio de GitHub.  
<https://github.com/moq/moq>
- NUnidad. Sitio oficial.  
<https://nunit.org/>

### Implementación de pruebas de servicio en una aplicación de varios contenedores

Como se señaló anteriormente, cuando prueba aplicaciones de varios contenedores, todos los microservicios deben ejecutarse dentro del host de Docker o del clúster de contenedores. Las pruebas de servicio de un extremo a otro que incluyen múltiples operaciones que involucran varios microservicios requieren que implemente e inicie toda la aplicación en el host de Docker ejecutando docker-compose up (o un mecanismo comparable si está usando un orquestador).

Una vez que se ejecuta la aplicación completa y todos sus servicios, puede ejecutar pruebas funcionales y de integración de extremo a extremo.

Hay algunos enfoques que puede utilizar. En el archivo docker-compose.yml que 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 redacción que ejecutaría sus pruebas en la imagen a la que se dirige. Al usar otro archivo de redacción para las pruebas de integración que incluye sus microservicios y bases de datos en contenedores, puede asegurarse de que los datos relacionados siempre se restablezcan a su estado original antes de ejecutar las pruebas.

Una vez que la aplicación de redacción está en funcionamiento, puede aprovechar los puntos de interrupción y las excepciones si está ejecutando Visual Studio. O puede ejecutar las pruebas de integración automáticamente en su canalización de CI en Azure DevOps Services o cualquier otro sistema de CI/CD que admita contenedores Docker.

## Pruebas en eShopOnContainers

Las pruebas de la aplicación de referencia (eShopOnContainers) se reestructuraron recientemente y ahora hay cuatro categorías:

1. Pruebas unitarias , simplemente viejas pruebas unitarias regulares, contenidas en {MicroserviceName}.UnitTests proyectos
2. Pruebas funcionales/de integración de microservicios, con casos de prueba que involucren la infraestructura para cada uno microservicio pero aislado de los demás y están contenidos en el Proyectos {MicroserviceName}.FunctionalTests .
3. Pruebas funcionales/de integración de aplicaciones, que se centran en la integración de microservicios, con prueba casos que ejercen varios microservicios. Estas pruebas se encuentran en el proyecto Application.FunctionalTests.

Mientras que las pruebas unitarias y de integración se organizan en una carpeta de prueba dentro del proyecto de microservicio, las pruebas de aplicación y de carga se administran por separado en la carpeta raíz, como se muestra en la Figura 6-25.

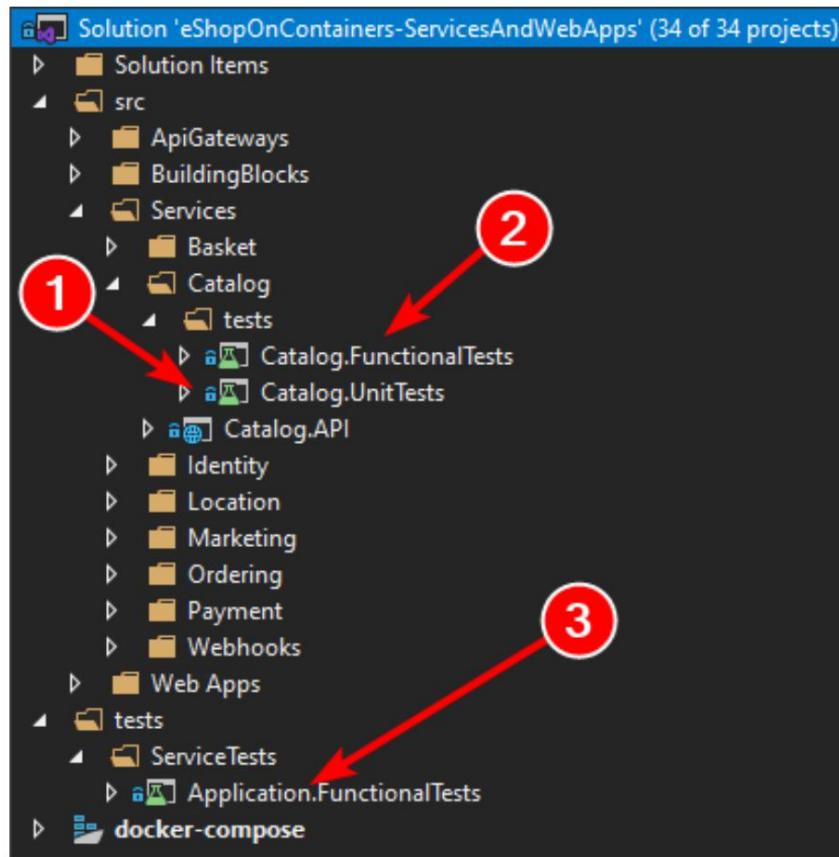


Figura 6-25. Pruebe la estructura de carpetas en eShopOnContainers

Las pruebas funcionales/de integración de microservicios y aplicaciones se ejecutan desde Visual Studio, utilizando el ejecutor de pruebas normal, pero primero debe iniciar los servicios de infraestructura requeridos, con un conjunto de archivos de composición de docker contenidos en la carpeta de prueba de la solución:

`docker-compose-prueba.yml`

```

versión: '3.4'

servicios:
  redis.datos:
    imagen: redis:alpine
  rabbitmq: imagen: rabbitmq:3-
    management-alpine
  sqldata: imagen: mcr.microsoft.com/mssql/server:2017-latest
  nosqldata: imagen: mongo
  
```

`docker-compose-test.override.yml`

```

versión: '3.4'

servicios:
  
```

```

redis.datos:
puertos:
- "6379:6379"
rabbitmq: puertos: -
"15672:15672" -
"5672:5672" sqldata:
entorno:

- SA_PASSWORD=Contraseña@palabra
- ACCEPT_EULA=
Puertos Y: - "5433:1433"

nosqldata:
puertos: -
"27017:27017"

```

Entonces, para ejecutar las pruebas funcionales/de integración, 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 arriba
```

Como puede ver, estos archivos docker-compose solo inician los microservicios Redis, RabbitMQ, SQL Server y MongoDB.

## Recursos adicionales

- Pruebas de unidad e integración en eShopOnContainers <https://github.com/dotnet-architecture/eShopOnContainers/wiki/Unit-and-integration-testing>
- Pruebas de carga en eShopOnContainers <https://github.com/dotnet-architecture/eShopOnContainers/wiki/Load-testing>

## Implementar tareas en segundo plano en microservicios con IHostedService y la clase BackgroundService

Las tareas en segundo plano y los trabajos programados son algo que podría necesitar usar en cualquier aplicación, ya sea que siga o no el patrón de arquitectura de microservicios. La diferencia cuando se usa una arquitectura de microservicios es que puede implementar la tarea en segundo plano en un proceso/contenedor separado para el alojamiento, de modo que pueda reducirlo o aumentarlo según sus necesidades.

Desde un punto de vista genérico, en .NET llamamos a este tipo de tareas Servicios alojados, porque son servicios lógica que aloja dentro de su host/aplicación/microservicio. Tenga en cuenta que, en este caso, el servicio alojado simplemente significa una clase con la lógica de la tarea en segundo plano.

Desde .NET Core 2.0, el marco proporciona una nueva interfaz llamada [IHostedService](#) que lo ayuda a implementar fácilmente los servicios alojados. La idea básica es que puede registrar varias tareas en segundo plano (servicios alojados) que se ejecutan en segundo plano mientras se ejecuta su host o host web, como se muestra en la imagen 6-26.

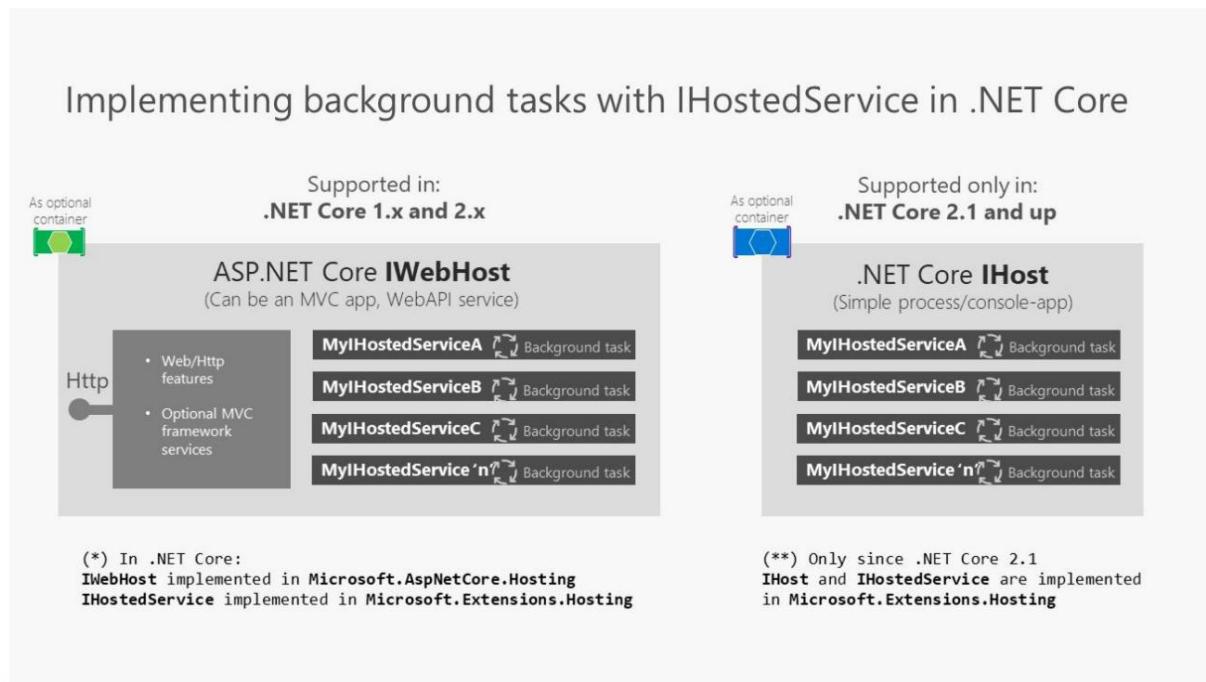


Figura 6-26. Uso de `IHostedService` en un `WebHost` frente a un `Host`

ASP.NET Core 1.x y 2.x admiten `IWebHost` para procesos en segundo plano en aplicaciones web. .NET Core 2.1 y versiones posteriores admiten `IHost` para procesos en segundo plano con aplicaciones de consola simples. Tenga en cuenta la diferencia entre `WebHost` y `Host`.

Un `WebHost` (clase base que implementa `IWebHost`) en ASP.NET Core 2.0 es el artefacto de infraestructura que usa para proporcionar funciones de servidor HTTP a su proceso, como cuando implementa una aplicación web MVC o un servicio API web. Proporciona todas las bondades de la nueva infraestructura en ASP.NET Core, lo que le permite usar la inserción de dependencias, insertar middlewares en la canalización de solicitudes y similares. `WebHost` utiliza estos mismos `IHostedServices` para tareas en segundo plano.

Se introdujo un `Host` (clase base que implementa `IHost`) en .NET Core 2.1. Básicamente, un `Host` le permite tener una infraestructura similar a la que tiene con `WebHost` (inyección de dependencia, servicios alojados, etc.), pero en este caso, solo desea tener un proceso simple y más liviano como el host, sin nada relacionado a funciones de servidor MVC, Web API o HTTP.

Por lo tanto, puede elegir y crear un proceso de host especializado con `IHost` para manejar los servicios alojados y nada más, como un microservicio hecho solo para alojar `IHostedServices`, o alternativamente puede extender un ASP.NET Core `WebHost` existente, como un Aplicación ASP.NET Core Web API o MVC existente.

Cada enfoque tiene ventajas y desventajas según su negocio y sus necesidades de escalabilidad. La conclusión es básicamente que si sus tareas en segundo plano no tienen nada que ver con HTTP (`IWebHost`), debe usar Puedo alojar.

## Registro de servicios alojados en su `WebHost` o `Host`

Profundicemos más en la interfaz `IHostedService`, ya que su uso es bastante similar en un `WebHost` o en un `Host`.

SignalR es un ejemplo de un artefacto que usa servicios alojados, pero también puede usarlo para cosas mucho más simples como:

- Una tarea en segundo plano sondeando una base de datos en busca de cambios.
- Una tarea programada que actualiza algún caché periódicamente.
- Una implementación de QueueBackgroundWorkItem que permite ejecutar una tarea en un subproceso en segundo plano.
- Procesar mensajes de una cola de mensajes en el fondo 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 en una tarea en segundo plano que implemente IHostedService.

La forma en que agrega uno o varios IHostedServices a suWebHost o Host es registrándolos a través del método de extensión [AddHostedService](#) en un ASP.NET CoreWebHost (o en un Host en .NET Core 2.1 y superior). Básicamente, debe registrar los servicios alojados dentro del método familiar ConfigureServices() de la clase Startup, como en el siguiente código de un ASP.NET WebHost típico.

```
public IServiceProvider ConfigureServices(IServiceCollection servicios) {
    //Otros registros DI;
    // Registrar servicios alojados
    servicios.AddHostedService<GracePeriodManagerService>();
    servicios.AddHostedService<MyHostedServiceB>();
    servicios.AddHostedService<MyHostedServiceC>(); //...
}
```

En ese código, el servicio alojado GracePeriodManagerService es un código real del microservicio comercial Pedidos en eShopOnContainers, mientras que los otros dos son solo dos ejemplos adicionales.

La ejecución de la tarea en segundo plano de IHostedService se coordina con la vida útil de la aplicación (host o microservicio, para el caso). Registra tareas cuando se inicia la aplicación y tiene la oportunidad de realizar alguna acción elegante o limpieza cuando la aplicación se está cerrando.

Sin usar IHostedService, siempre puede iniciar un subproceso en segundo plano para ejecutar cualquier tarea. La diferencia está precisamente en el momento del cierre de la aplicación, cuando ese hilo simplemente se eliminaría sin tener la oportunidad de ejecutar acciones de limpieza elegantes.

## La interfaz IHostedService

Cuando registra un IHostedService, .NET llamará a los métodos StartAsync() y StopAsync() de su tipo IHostedService durante el inicio y la detención de la aplicación, respectivamente. Para obtener más detalles, consulte la interfaz [IHostedService](#)

Como puede imaginar, puede crear múltiples implementaciones de IHostedService y registrarlas en el método ConfigureServices() en el contenedor DI, como se mostró anteriormente. Todos esos servicios alojados se iniciarán y detendrán junto con la aplicación/microservicio.

Como desarrollador, usted es responsable de manejar la acción de detención de sus servicios cuando el host activa el método `StopAsync()`.

## Implementación de `IHostedService` con una clase de servicio alojada personalizada derivada de la clase base `BackgroundService`

Puede seguir adelante y crear su clase de servicio alojada personalizada desde cero e implementar `IHostedService`, como debe hacer cuando usa .NET Core 2.0 y versiones posteriores.

Sin embargo, dado que la mayoría de las tareas en segundo plano tendrán necesidades similares con respecto a la administración de tokens de cancelación y otras operaciones típicas, existe una clase base abstracta conveniente de la que puede derivar, llamada `BackgroundService` (disponible desde .NET Core 2.1).

Esa clase proporciona el trabajo principal necesario para configurar la tarea en segundo plano.

El siguiente código es la clase base abstracta `BackgroundService` tal como se implementa en .NET.

```
// Derechos de autor (c) Fundación .NET. Con licencia de Apache License, versión 2.0. /// <summary> /// Clase
base para implementar un <see cref="IHostedService"/> de ejecución prolongada. /// </summary> clase
abstract pública BackgroundService: IHostedService, IDisposable {

    tarea privada _executingTask; privado
    de solo lectura CancellationSource _stoppingCts = new
        CancellationSource();

    Tarea abstracta protegida ExecuteAsync (CancellationToken detención de Token);

    Tarea virtual pública StartAsync (CancellationToken cancelationToken) {

        // Almacenar la tarea que estamos ejecutando
        _executingTask = ExecuteAsync(_stoppingCts.Token);

        // Si la tarea se completó, devuélvala, // esto generará una
        cancelación y falla para la persona que llama if (_executingTask.IsCompleted) {

            volver _ejecuciónTarea;
        }

        // De lo contrario, se está
        ejecutando return Task.CompletedTask;
    }

    pública virtual asincrónica Tarea StopAsync(CancellationToken cancelationToken) {

        // Detener llamada sin iniciar if
        (_executingTask == null) {

            retorno;
        }

        prueba
        {
            // Señal de cancelación al método de ejecución
            _stoppingCts.Cancel();
        }
    }
}
```

```

    } finalmente
    {
        // Espere hasta que se complete la tarea o hasta que se active el token de
        // detención await Task.WhenAny(_executingTask, Task.Delay(Timeout.Infinite,
        // cancellationToken));
    }

}

vacío virtual público Dispose () {

    _stoppingCts.Cancel();
}
}

```

Al derivar de la clase base abstracta anterior, gracias a esa implementación heredada, solo necesita implementar el método `ExecuteAsync()` en su propia clase de servicio alojada personalizada, como en el siguiente código simplificado de `eShopOnContainers` que sondea una base de datos y publica eventos de integración en el Event Bus cuando sea necesario.

```

clase pública GracePeriodManagerService: BackgroundService {

    privado de solo lectura ILogger<GracePeriodManagerService> _logger; privado de
    solo lectura OrderingBackgroundSettings _settings;

    privado de solo lectura IEventBus _eventBus;

    GracePeriodManagerService público (configuración de IOptions<OrderingBackgroundSettings>,
        bus de eventos IEventBus,
        registrador ILogger<GracePeriodManagerService>)

    {
        // Validaciones de parámetros del constructor...
    }

    protegido anular asíncrono Tarea ExecuteAsync (CancellationToken detención de Token) {

        _logger.LogDebug($"GracePeriodManagerService se está iniciando");

        deteniendoToken.Register(() =>
            _logger.LogDebug($" La tarea en segundo plano de GracePeriod se está deteniendo"));

        while (!tokenToken.IsCancellationRequested) {

            _logger.LogDebug($" Tarea GracePeriod haciendo trabajo en segundo plano");

            // Este método eShopOnContainers consulta una tabla de base de datos // y publica
            // eventos en el Bus de eventos (RabbitMQ / ServiceBus)
            VerificarPedidosConfirmadosPeríodoGracia();

            esperar Task.Delay(_settings.CheckUpdateTime, parandoToken );
        }

        _logger.LogDebug($"La tarea en segundo plano de GracePeriod se está deteniendo");
    }

    .../...
}

```

En este caso específico para eShopOnContainers, ejecuta un método de aplicación que consulta una tabla de base de datos en busca de pedidos con un estado específico y, al aplicar cambios, publica eventos de integración a través del bus de eventos (debajo puede estar usando RabbitMQ o Azure Service Bus) .

Por supuesto, podría ejecutar cualquier otra tarea de antecedentes comerciales en su lugar.

De forma predeterminada, el token de cancelación se establece con un tiempo de espera de 5 segundos, aunque puede cambiar ese valor al crear su WebHost utilizando la extensión `UseShutdownTimeout` de `IWebHostBuilder`. Esto significa que se espera que nuestro servicio se cancele dentro de 5 segundos, de lo contrario, se cancelará más abruptamente.

El siguiente código estaría cambiando ese tiempo a 10 segundos.

```
WebHost.CreateDefaultBuilder(argumentos)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...
```

### Diagrama de clase de resumen

La siguiente imagen muestra un resumen visual de las clases e interfaces involucradas al implementar `IHostedServices`.

Class diagram with a custom `IHostedService` and related classes and interfaces

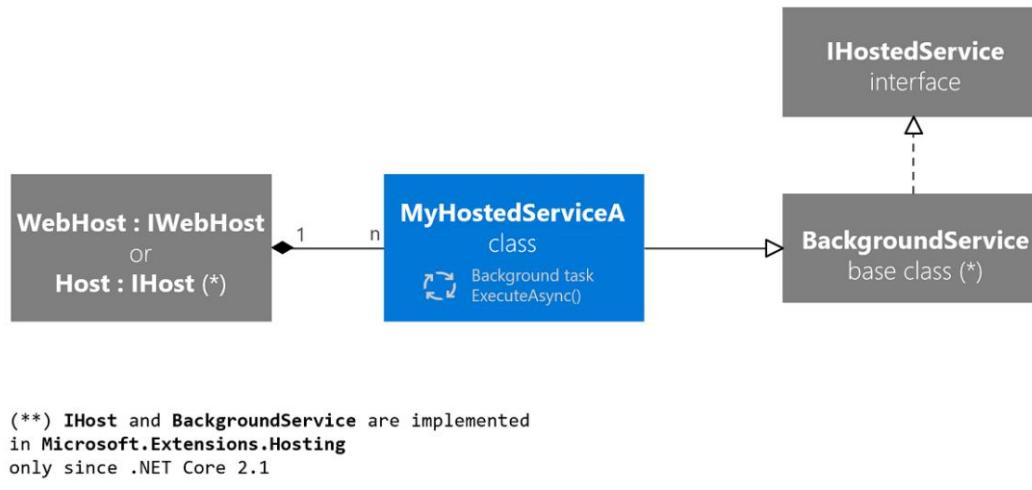


Figura 6-27. Diagrama de clases que muestra las múltiples clases e interfaces relacionadas con `IHostedService`

Diagrama de clases: `IWebHost` e `IHost` pueden alojar muchos servicios, que se heredan de `BackgroundService`, que implementa `IHostedService`.

### Consideraciones de implementación y conclusiones

Es importante tener en cuenta que la forma en que implementa su ASP.NET Core WebHost o .NET Host puede afectar la solución final. Por ejemplo, si implementa su WebHost en IIS o un Azure App Service normal, su host puede cerrarse debido a los reciclajes del grupo de aplicaciones. Pero si está implementando su host como un contenedor en un orquestador como Kubernetes, puede controlar la cantidad segura de instancias activas de su host. Además, podrías considerar otros enfoques en la nube especialmente hechos

para estos escenarios, como Azure Functions. Finalmente, si necesita que el servicio se ejecute todo el tiempo y se está implementando en un servidor de Windows, puede usar un servicio de Windows.

Pero incluso para un WebHost implementado en un grupo de aplicaciones, existen escenarios como repoblar o vaciar la caché en memoria de la aplicación que aún serían aplicables.

La interfaz `IHostedService` proporciona una manera conveniente de iniciar tareas en segundo plano en una aplicación web ASP.NET Core (en .NET Core 2.0 y versiones posteriores) o en cualquier proceso/host (a partir de .NET Core 2.1 con `IHost`). Su principal beneficio es la oportunidad que obtiene con la cancelación elegante para limpiar el código de sus tareas en segundo plano cuando el host mismo se está cerrando.

## Recursos adicionales

- Creación de una tarea programada en ASP.NET Core/Standard 2.0  
<https://blog.maartenballiauw.be/post/2017/08/01/building-a-scheduled-cache-updater-in-aspnet-core-2.html>
- Implementación de `IHostedService` en ASP.NET Core 2.0  
<https://www.stevejgordon.co.uk/asp-net-core-2-ihostedservice>
- Ejemplo de `GenericHost` con ASP.NET Core 2.1 <https://github.com/aspnet/Hosting/tree/release/2.1/samples/GenericHostSample>

## Implementar puertas de enlace API con Ocelot

### Importante

La aplicación de microservicio de referencia [eShopOnContainers](#) actualmente utiliza funciones proporcionadas por [Envoy](#) para implementar API Gateway en lugar del [Ocelot mencionado anteriormente](#). Hicimos esta elección de diseño debido a la compatibilidad integrada de Envoy con el protocolo WebSocket, requerido por las nuevas comunicaciones entre servicios gRPC implementadas en eShopOnContainers. Sin embargo, conservamos esta sección en la guía para que pueda considerar a Ocelot como una API Gateway simple, capaz y liviana adecuada para escenarios de nivel de producción. Además, la última versión de Ocelot contiene un cambio importante en su esquema json. Considere usar Ocelot < v16.0.0, o use la clave `Routes` en lugar de `ReRoutes`.

## Construya y diseñe sus API Gateways

El siguiente diagrama de arquitectura muestra cómo se implementaron API Gateways con Ocelot en eShopOnContainers.

## eShopOnContainers reference application

(Development environment architecture)

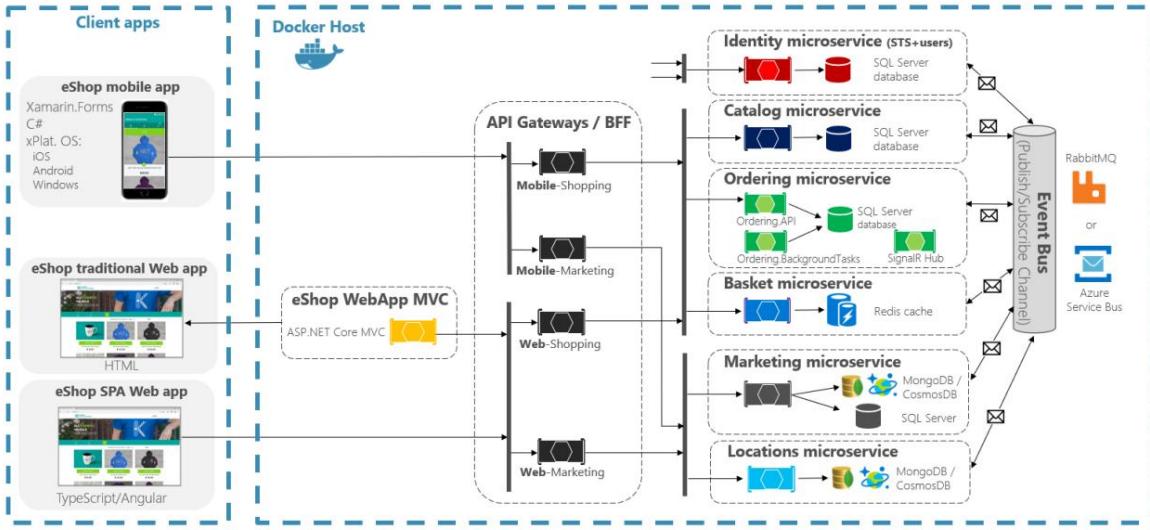


Figura 6-28. Arquitectura eShopOnContainers con API Gateways

Ese diagrama muestra cómo se implementa toda la aplicación en un único host Docker o PC de desarrollo con "Docker para Windows" o "Docker para Mac". Sin embargo, la implementación en cualquier orquestador sería similar, pero cualquier contenedor del diagrama se podría escalar horizontalmente en el orquestador.

Además, los activos de infraestructura, como bases de datos, caché y agentes de mensajes, deben descargarse del orquestador e implementarse en sistemas de alta disponibilidad para la infraestructura, como Azure SQL Database, Azure Cosmos DB, Azure Redis, Azure Service Bus o cualquier clúster de alta disponibilidad. solución en las instalaciones.

Como también puede observar en el diagrama, tener varias puertas de enlace API permite que varios equipos de desarrollo sean autónomos (en este caso, funciones de marketing frente a funciones de compras) al desarrollar e implementar sus microservicios, además de sus propias puertas de enlace API relacionadas.

Si tuviera una puerta de enlace de API monolítica única, eso significaría un punto único que varios equipos de desarrollo actualizarían, lo que podría unir todos los microservicios con una sola parte de la aplicación.

Yendo mucho más allá en el diseño, a veces una puerta de enlace API detallada también se puede limitar a un solo microservicio comercial según la arquitectura elegida. Tener los límites de API Gateway dictados por el negocio o el dominio lo ayudará a obtener un mejor diseño.

Por ejemplo, la granularidad fina en el nivel de API Gateway puede ser especialmente útil para aplicaciones de IU compuestas más avanzadas que se basan en microservicios, porque el concepto de API Gateway detallado es similar a un servicio de composición de IU.

Profundizamos en más detalles en la sección anterior [Creación de una interfaz de usuario compuesta basada en microservicios](#).

Como conclusión clave, para muchas aplicaciones medianas y grandes, el uso de un producto API Gateway personalizado suele ser un buen enfoque, pero no como un único agregador monolítico o API Gateway personalizado central único, a menos que API Gateway permita múltiples configuraciones independientes. áreas para los diversos equipos de desarrollo que crean microservicios autónomos.

## Ejemplos de microservicios/contenedores para redirigir a través de API Gateways

Como ejemplo, eShopOnContainers tiene alrededor de seis tipos de microservicios internos que deben publicarse a través de API Gateways, como se muestra en la siguiente imagen.

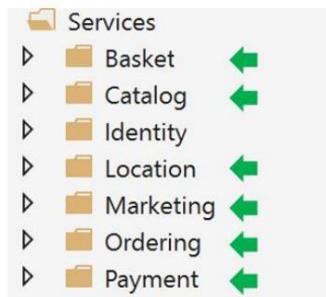


Figura 6-29. Carpetas de microservicios en la solución eShopOnContainers en Visual Studio

Sobre el servicio de Identidad, en el diseño se deja fuera del enruteamiento de API Gateway porque es la única preocupación transversal en el sistema, aunque con Ocelot también es posible incluirlo como parte de las listas de reenrutamiento.

Todos esos servicios se implementan actualmente como servicios API web de ASP.NET Core, como puede ver en el código. Centrémonos en uno de los microservicios, como el código de microservicio Catalog.

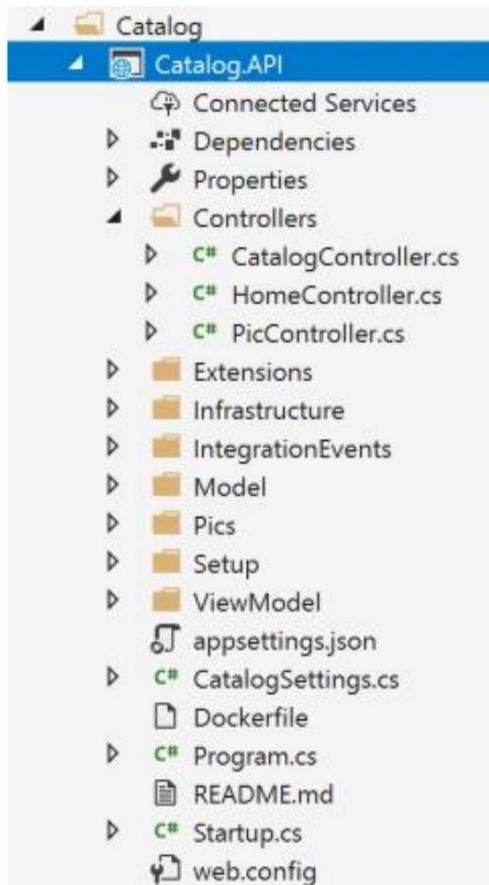


Figura 6-30. Ejemplo de microservicio de API web (microservicio de catálogo)

Puede ver que el microservicio Catalog es un proyecto típico de API web de ASP.NET Core con varios controladores y métodos, como en el siguiente código.

```
[HttpGet]
[Ruta("elementos/{id:int}")]
[ProduceResponseType((int) HttpStatusCode.BadRequest)]
[ProduceResponseType((int) HttpStatusCode.NotFound)]
[ProducesResponseType(typeof(CatalogItem), (int) HttpStatusCode.OK)] public async
Task<ActionResult> GetItemById(int id) {
    si (id <= 0) {
        devuelve BadRequest();
    } var elemento = esperar _catalogContext.CatalogItems.
        SingleOrDefaultAsync(ci => ci.Id == id);
    //...
    si (elemento! = nulo) {
        volver Ok(elemento);
    } devuelve No Encontrado();
}
```

La solicitud HTTP terminará ejecutando ese tipo de código C# accediendo a la base de datos de microservicios y cualquier acción adicional requerida.

Con respecto a la URL del microservicio, cuando los contenedores se implementan en su PC de desarrollo local (host Docker local), el contenedor de cada microservicio siempre tiene un puerto interno (generalmente el puerto 80) especificado en su archivo docker, como en el siguiente archivo docker:

```
DESDE mcr.microsoft.com/dotnet/aspnet:3.1 COMO base
WORKDIR /app EXPOSE 80
```

El puerto 80 que se muestra en el código es interno dentro del host Docker, por lo que las aplicaciones cliente no pueden acceder a él.

Las aplicaciones cliente solo pueden acceder a los puertos externos (si los hay) publicados al implementar con Docker Compose.

Esos puertos externos no deben publicarse al implementar en un entorno de producción. Por esta razón específica, por qué desea utilizar API Gateway, para evitar la comunicación directa entre las aplicaciones cliente y los microservicios.

Sin embargo, al desarrollar, desea acceder directamente al microservicio/contenedor y ejecutarlo a través de Swagger. Es por eso que en eShopOnContainers, los puertos externos aún se especifican incluso cuando API Gateway o las aplicaciones cliente no los utilizarán.

Este es un ejemplo del archivo docker-compose.override.yml para el microservicio Catalog:

```
catálogo-api:
  medioambiente:
    - ASPNETCORE_ENVIRONMENT=Desarrollo -
    - ASPNETCORE_URLS=http://0.0.0.0:80 -
    ConnectionString=SU_VALOR
    - ... Otras Variables de Entorno
```

puertos:

```
- "5101:80" # Importante: en un entorno de producción, debe eliminar el externo  
puerto (5101) guardado aquí para fines de depuración de microservicios.  
# El API Gateway redirige y accede a través del puerto interno (80).
```

Puede ver cómo en la configuración de docker-compose.override.yml, el puerto interno para el contenedor del catálogo es el puerto 80, pero el puerto para el acceso externo es el 5101. Pero la aplicación no debe usar este puerto cuando usa una API Gateway. , solo para depurar, ejecutar y probar solo el microservicio Catalog.

Normalmente, no implementará con docker-compose en un entorno de producción porque el entorno de implementación de producción adecuado para microservicios es un orquestador como Kubernetes o Service Fabric. Al implementar en esos entornos, usa diferentes archivos de configuración en los que no publicará directamente ningún puerto externo para los microservicios, pero siempre usará el proxy inverso de API Gateway.

Ejecute el microservicio de catálogo en su host Docker local. Ejecute la solución eShopOnContainers completa desde Visual Studio (ejecuta todos los servicios en los archivos docker-compose) o inicie el microservicio Catalog con el siguiente comando docker-compose en CMD o PowerShell ubicado en la carpeta donde se encuentra docker-compose.yml y se colocan docker-compose.override.yml .

```
docker-compose ejecutar --service-ports catalog-api
```

Este comando solo ejecuta el contenedor de servicios catalog-api más las dependencias que se especifican en docker-compose.yml. En este caso, el contenedor de SQL Server y el contenedor de RabbitMQ.

Luego, puede acceder directamente al microservicio Catalog y ver sus métodos a través de la interfaz de usuario de Swagger accediendo directamente a través de ese puerto "externo", en este caso <http://host.docker.internal:5101/swagger>:

Figura 6-31. Prueba del microservicio Catalog con su interfaz de usuario de Swagger

En este punto, podría establecer un punto de interrupción en el código C# en Visual Studio, probar el microservicio con los métodos expuestos en la interfaz de usuario de Swagger y finalmente limpiar todo con el comando docker-compose down .

Sin embargo, la comunicación de acceso directo al microservicio, en este caso a través del puerto externo 5101, es precisamente lo que quieras evitar en tu aplicación. Y puede evitarlo configurando el nivel adicional de direccionamiento indirecto de API Gateway (Ocelot, en este caso). De esa forma, la aplicación cliente no accederá directamente al microservicio.

## Implementación de sus API Gateways con Ocelot

Ocelot es básicamente un conjunto de middleware que puede aplicar en un orden específico.

Ocelot está diseñado para funcionar solo con ASP.NET Core. La última versión del paquete tiene como objetivo .NETCoreApp 3.1 y, por lo tanto, no es adecuado para las aplicaciones de .NET Framework.

Instala Ocelot y sus dependencias en su proyecto ASP.NET Core con el paquete NuGet de Ocelot, de Visual Studio.

#### Paquete de instalación Ocelot

En eShopOnContainers, su implementación de API Gateway es un proyecto ASP.NET Core WebHost simple, y el middleware de Ocelot maneja todas las funciones de API Gateway, como se muestra en la siguiente imagen:

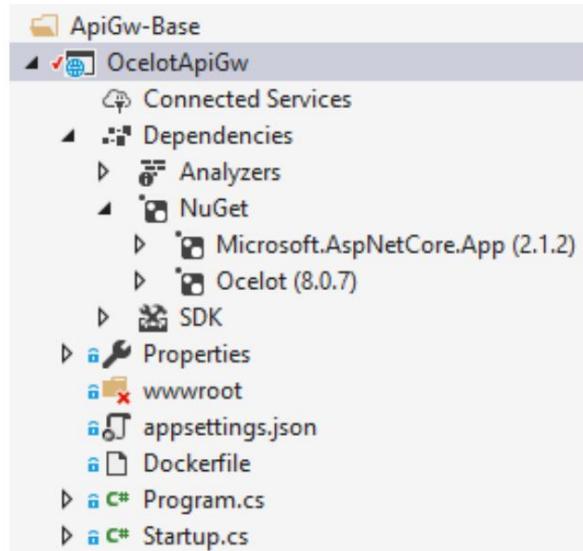


Figura 6-32. El proyecto base OcelotApiGw en eShopOnContainers

Este proyecto ASP.NET Core WebHost está construido con dos archivos simples: Program.cs y Startup.cs.

El Program.cs solo necesita crear y configurar el ASP.NET Core BuildWebHost típico.

```
espacio de nombres OcelotApiGw
{
    Programa de clase pública
    {
        public static void Principal(cadena[] argumentos) {

            BuildWebHost(argumentos).Ejecutar();
        }

        public static IWebHost BuildWebHost(string[] args) {

            constructor de var = WebHost.CreateDefaultBuilder(args);

            constructor.ConfigureServices(s => s.AddSingleton(constructor))
                .ConfigureAppConfiguration(ic =>
                    ic.AddJsonFile(Path.Combine("configuración",
                        "configuración.json")))
                .UseStartup<Inicio>(); var host
                = constructor.Build(); anfitrión de retorno ;
        }
    }
}
```

El punto importante aquí para Ocelot es el archivo configuration.json que debe proporcionar al constructor a través del método AddJsonFile() . Ese archivo configuration.json es donde especifica todos los ReRoutes de API Gateway, es decir, los puntos finales externos con puertos específicos y los puntos finales internos correlacionados, que generalmente usan puertos diferentes.

```
{
    "Rutas": [],
    "ConfiguraciónGlobal": {}
}
```

Hay dos secciones para la configuración. Una matriz de ReRoutes y una configuración global. Los ReRoutes son los objetos que le dicen a Ocelot cómo tratar una solicitud ascendente. La configuración global permite anular la configuración específica de ReRoute. Es útil si no desea administrar muchas configuraciones específicas de ReRoute.

Este es un ejemplo simplificado del [archivo de configuración de ReRoute](#) de una de las API Gateways de eShopOnContainers.

```
{
    "Rutas": [ {
        "DownstreamPathTemplate": "/api/{versión}/{todo}",
        "EsquemaDescendente": "http",
        "DownstreamHostAndPorts": [
            {
                "Anfitrión": "catálogo-api",
                "Puerto": 80
            }
        ],
        "UpstreamPathTemplate": "/api/{versión}/c/{todo}",
        "UpstreamHttpMethod": [ "POST", "PUT", "GET" ], {
            "DownstreamPathTemplate": "/api/{versión}/{todo}",
            "EsquemaDescendente": "http",
            "DownstreamHostAndPorts": [
                {
                    "Anfitrión": "cesta-api",
                    "Puerto": 80
                }
            ],
            "UpstreamPathTemplate": "/api/{versión}/b/{todo}",
            "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
            "Opciones de autenticación": {
                "AuthenticationProviderKey": "IdentityApiKey",
                "Ámbitos permitidos": []
            }
        }
    ],
    "ConfiguraciónGlobal": {
        "RequestIdKey": "OcRequestId",
        "AdministrationPath": "/administración"
    }
}
```

La funcionalidad principal de Ocelot API Gateway es recibir solicitudes HTTP entrantes y reenviarlas a un servicio descendente, actualmente como otra solicitud HTTP. Ocelot describe el enrutamiento de una solicitud a otra como ReRoute.

Por ejemplo, centrémonos en uno de los ReRoutes en el archivo configuration.json de arriba, la configuración del microservicio Basket.

```
{
  "DownstreamPathTemplate": "/api/{versión}/{todo}",
  "EsquemaDescendente": "http",
  "DownstreamHostAndPorts": [
    {
      "Anfitrión": "cesta-api",
      "Puerto": 80
    }
  ],
  "UpstreamPathTemplate": "/api/{versión}/b/{todo}",
  "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
  "Opciones de autenticación": {
    "AuthenticationProviderKey": "IdentityApiKey",
    "Ámbitos permitidos": []
  }
}
```

DownstreamPathTemplate, Scheme y DownstreamHostAndPorts crean la URL interna del microservicio a la que se reenviará esta solicitud.

El puerto es el puerto interno utilizado por el servicio. Al usar contenedores, el puerto especificado en su dockerfile.

El host es un nombre de servicio que depende de la resolución de nombre de servicio que esté utilizando. Cuando se usa docker-compose, los nombres de los servicios los proporciona Docker Host, que usa los nombres de servicio provistos en los archivos de docker-compose. Si usa un orquestador como Kubernetes o Service Fabric, ese nombre debe ser resuelto por el DNS o la resolución de nombres proporcionada por cada orquestador.

DownstreamHostAndPorts es una matriz que contiene el host y el puerto de cualquier servicio descendente al que desee reenviar solicitudes. Por lo general, esta configuración solo contendrá una entrada, pero a veces es posible que desee cargar solicitudes de equilibrio a sus servicios posteriores y Ocelot le permite agregar más de una entrada y luego seleccionar un equilibrador de carga. Pero si usa Azure y cualquier orquestador, probablemente sea una mejor idea equilibrar la carga con la nube y la infraestructura del orquestador.

UpstreamPathTemplate es la URL que Ocelot usará para identificar qué DownstreamPathTemplate usar para una solicitud determinada del cliente. Finalmente, se usa UpstreamHttpMethod para que Ocelot pueda distinguir entre diferentes solicitudes (GET, POST, PUT) a la misma URL.

En este punto, podría tener una sola Ocelot API Gateway (ASP.NET Core WebHost) usando uno o varios [archivos de configuración.json combinados](#) o también puede almacenar la [configuración en una tienda Consul KV](#).

Pero como se presentó en las secciones de arquitectura y diseño, si realmente desea tener microservicios autónomos, podría ser mejor dividir esa puerta de enlace de API monolítica única en varias puertas de enlace de API y/o BFF (Backend para frontend). Para ese propósito, veamos cómo implementar ese enfoque con contenedores Docker.

## Uso de una única imagen de contenedor Docker para ejecutar varios tipos de contenedores API Gateway/BFF diferentes

En eShopOnContainers, usamos una sola imagen de contenedor de Docker con Ocelot API Gateway pero luego, en tiempo de ejecución, creamos diferentes servicios/contenedores para cada tipo de API-Gateway/BFF al proporcionar un archivo de configuración.json diferente, usando un docker volumen para acceder a una carpeta de PC diferente para cada servicio.

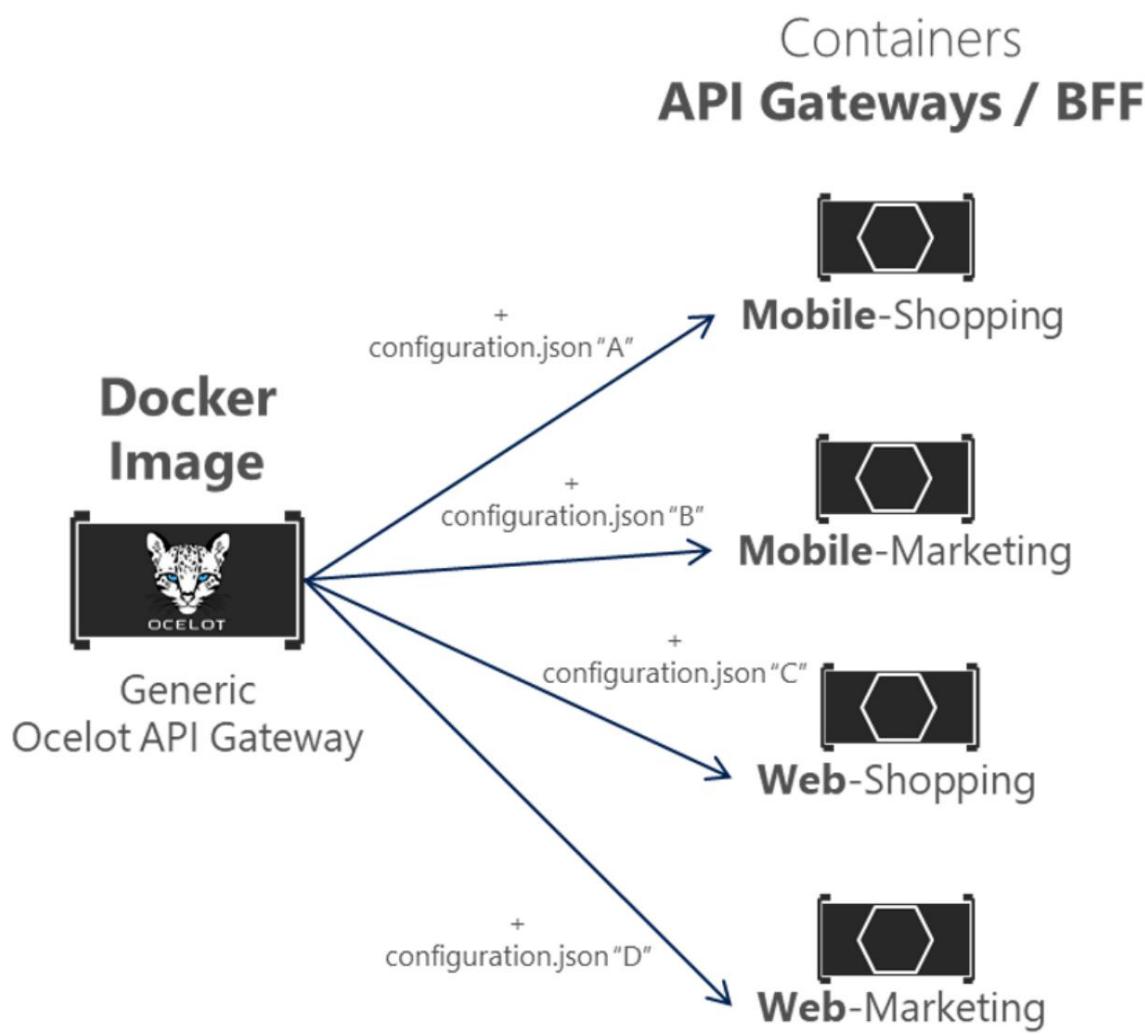


Figura 6-33. Reutilización de una sola imagen de Ocelot Docker en múltiples tipos de API Gateway

En eShopOnContainers, la "Imagen genérica de Ocelot API Gateway Docker" se crea con el proyecto llamado 'OcelotApiGw' y el nombre de imagen "eshop/ocelotapigw" que se especifica en el archivo docker compose.yml.

Luego, al implementar en Docker, se crearán cuatro contenedores API-Gateway a partir de esa misma imagen de Docker, como se muestra en el siguiente extracto del archivo docker-compose.yml.

```
compra móvilapigw:
  imagen: eshop/ocelotapigw:${TAG:-latest}
```

```

construir:
  contexto: .
  dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

mobilemarketingapigw:
  imagen: eshop/ocelotapigw:${TAG:-latest} compilación:

  contexto: .
  dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

webshoppingapigw:
  imagen: eshop/ocelotapigw:${TAG:-latest} compilación:

  contexto: .
  dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

webmarketingapigw:
  imagen: eshop/ocelotapigw:${TAG:-latest} compilación:
  contexto: . dockerfile: src/ApiGateways/ApiGw-Base/
    Dockerfile

```

Además, como puede ver en el siguiente archivo docker-compose.override.yml, la única diferencia entre esos contenedores API Gateway es el archivo de configuración de Ocelot, que es diferente para cada contenedor de servicio y se especifica en tiempo de ejecución a través de un volumen de Docker.

```

mobileshoppingapigw:
  entorno:
    - ASPNETCORE_ENVIRONMENT=Desarrollo
    - IdentityUrl=http://identity-api puertos: -
  Volúmenes "5200:80":
    - ./src/ApiGateways/Mobile.Bff.Shopping/apigw:/app/configuration

mobilemarketingapigw:
  entorno:
    - ASPNETCORE_ENVIRONMENT=Desarrollo
    - IdentityUrl=http://identity-api puertos: -
  Volúmenes "5201:80":
    - ./src/ApiGateways/Mobile.Bff.Marketplace/apigw:/app/configuration

webshoppingapigw:
  entorno:
    - ASPNETCORE_ENVIRONMENT=Desarrollo
    - IdentityUrl=http://identity-api puertos: -
  Volúmenes "5202:80":
    - ./src/ApiGateways/Web.Bff.Shopping/apigw:/app/configuration

webmarketingapigw:
  entorno:
    - ASPNETCORE_ENVIRONMENT=Desarrollo
    - IdentityUrl=http://identity-api puertos: -
  "5203:80"

```

volúmenes:

```
- ./src/ApiGateways/Web.Bff.Marketeting/apigw:/app/configuration
```

Debido a ese 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 comercial/BFF específica es solo un archivo de configuración.json, porque las cuatro puertas de enlace de API se basan en la misma imagen de Docker.

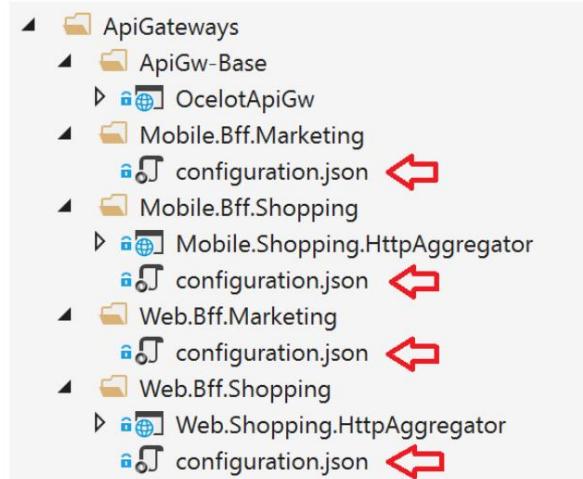


Figura 6-34. El único archivo necesario para definir cada API Gateway/BFF con Ocelot es un archivo de configuración

Al dividir la API Gateway en varias API Gateway, los diferentes equipos de desarrollo que se centran en diferentes subconjuntos de microservicios pueden administrar sus propias API Gateway mediante el uso de archivos de configuración de Ocelot independientes. Además, al mismo tiempo, pueden reutilizar la misma imagen de Ocelot Docker.

Ahora, si ejecuta eShopOnContainers con API Gateways (incluido de forma predeterminada en VS al abrir la solución eShopOnContainers-ServicesAndWebApps.sln o si ejecuta "docker-compose up"), se realizarán las siguientes rutas de muestra.

Por ejemplo, al visitar la URL ascendente <http://host.docker.internal:5202/api/v1/c/catalog/items/2/>

el resultado de la URL descendente interna <http://catalog-api/api/v1/2> dentro del host de Docker, como en el siguiente navegador.

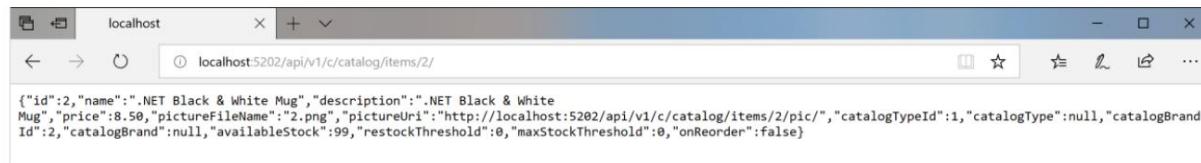


Figura 6-35. Acceder a un microservicio a través de una URL proporcionada por API Gateway

Por motivos de prueba o depuración, si desea acceder directamente al contenedor de Catalog Docker (solo en el entorno de desarrollo) sin pasar por API Gateway, ya que 'catalog-api' es una resolución DNS interna del host de Docker (detección de servicios manejado por nombres de servicio de redacción de docker), la única forma de acceder directamente al contenedor es a través del puerto externo publicado en docker-compose.override.yml, que se proporciona solo para pruebas de desarrollo, como <http://host.docker.internal:5101/api/v1/Catalog/items/1> en el siguiente navegador.



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 API Gateways, no a través de los "accesos directos" de puertos directos.

## El patrón de agregación de Gateway en eShopOnContainers

Como se presentó anteriormente, una forma flexible de implementar la agregación de solicitudes es con servicios personalizados, por código. También podría implementar la agregación de solicitudes con la [función Agregación de solicitudes en Ocelot](#), pero es posible que no sea tan flexible como necesita. Por lo tanto, la forma seleccionada de implementar la agregación en eShopOnContainers es con un servicio de API web ASP.NET Core explícito para cada agregador.

De acuerdo con ese enfoque, el diagrama de composición de API Gateway es en realidad un poco más extenso cuando se consideran los servicios del agregador que no se muestran en el diagrama de arquitectura global simplificado que se mostró anteriormente.

En el siguiente diagrama, también puede ver cómo funcionan los servicios del agregador con sus puertas de enlace API relacionadas.

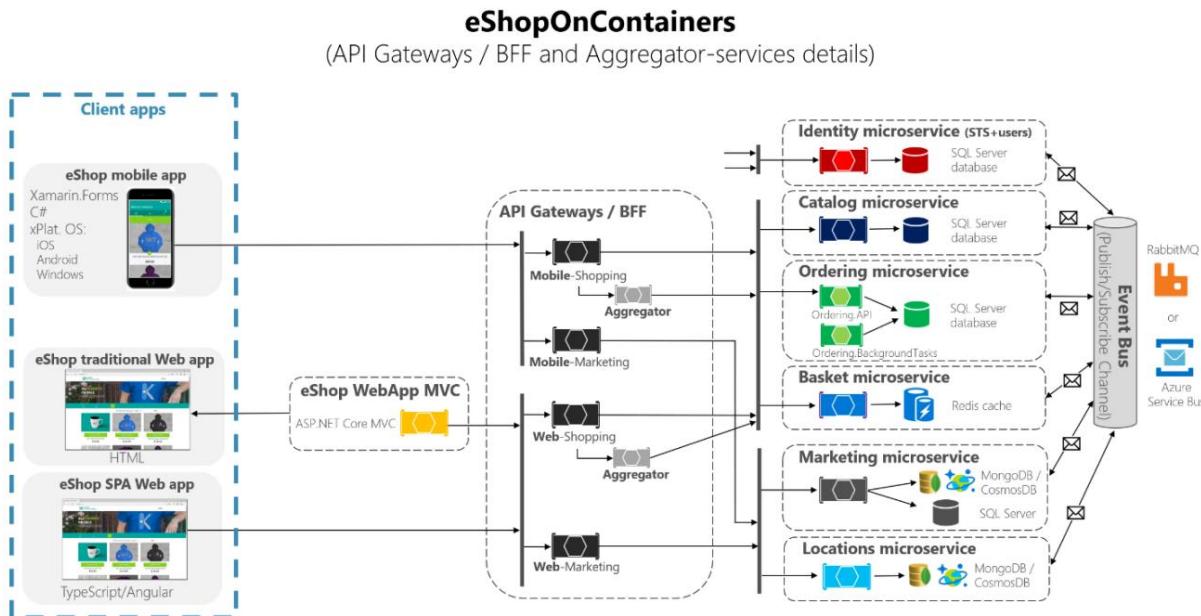


Figura 6-37. Arquitectura eShopOnContainers con servicios de agregador

Si se acerca más, en el área de negocios "Compras" en la siguiente imagen, puede ver que la conversación entre las aplicaciones cliente y los microservicios se reduce cuando se usan los servicios de agregación en API Gateways.

## eShopOnContainers

(API Gateways / BFF and Aggregator-services zoom-in)

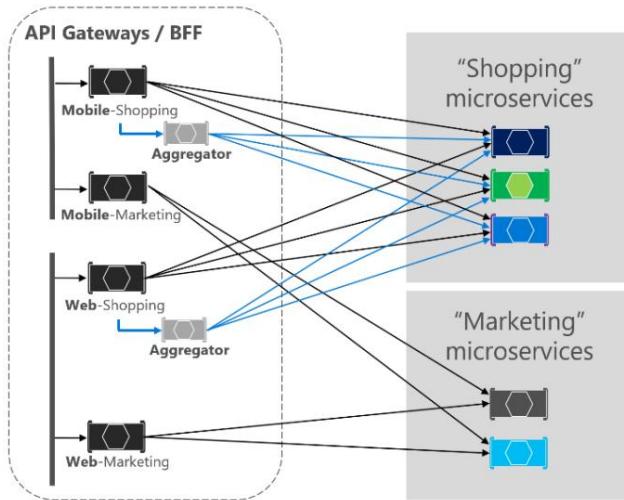


Figura 6-38. Ampliar la visión de los servicios del Agregador

Puede notar cómo cuando el diagrama muestra las posibles solicitudes provenientes de las API Gateways, puede volverse complejo.

Por otro lado, cuando usa el patrón de agregador, puede ver cómo las flechas en azul simplificarían la comunicación desde la perspectiva de la aplicación del cliente. Este patrón no solo ayuda a reducir la charlatanería y la latencia en la comunicación, sino que también mejora significativamente la experiencia del usuario para las aplicaciones remotas (aplicaciones móviles y SPA).

En el caso del área de negocio "Marketing" y microservicios, es un caso de uso sencillo por lo que no había necesidad de utilizar agregadores, pero también podría ser posible, en caso de ser necesario.

### Autenticación y autorización en Ocelot API Gateways

En una puerta de enlace de API de Ocelot, puede instalar el servicio de autenticación, como un servicio de API web ASP.NET Core utilizando [IdentityServer](#) que proporciona el token de autenticación, ya sea fuera o dentro de la puerta de enlace de API.

Dado que eShopOnContainers utiliza varias puertas de enlace de API con límites basados en BFF y áreas comerciales, el servicio de identidad/autenticación queda fuera de las puertas de enlace de API, como se resalta en amarillo en el siguiente diagrama.

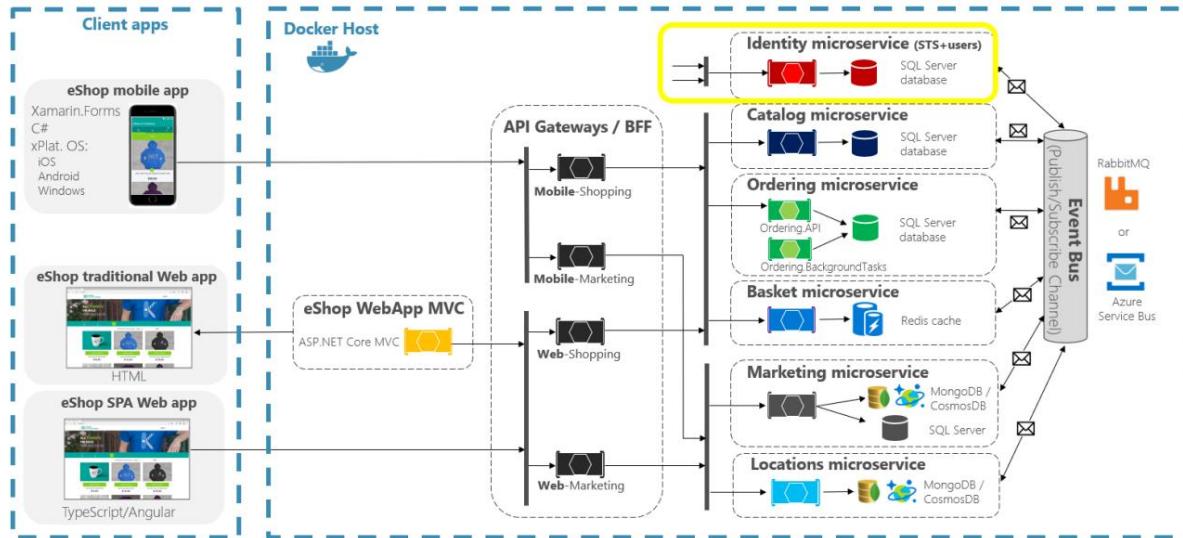


Figura 6-39. Posición del servicio de Identidad en eShopOnContainers

Sin embargo, Ocelot también admite colocar el microservicio Identity/Auth dentro del límite de API Gateway, como en este otro diagrama.

## Authentication in Ocelot API Gateway

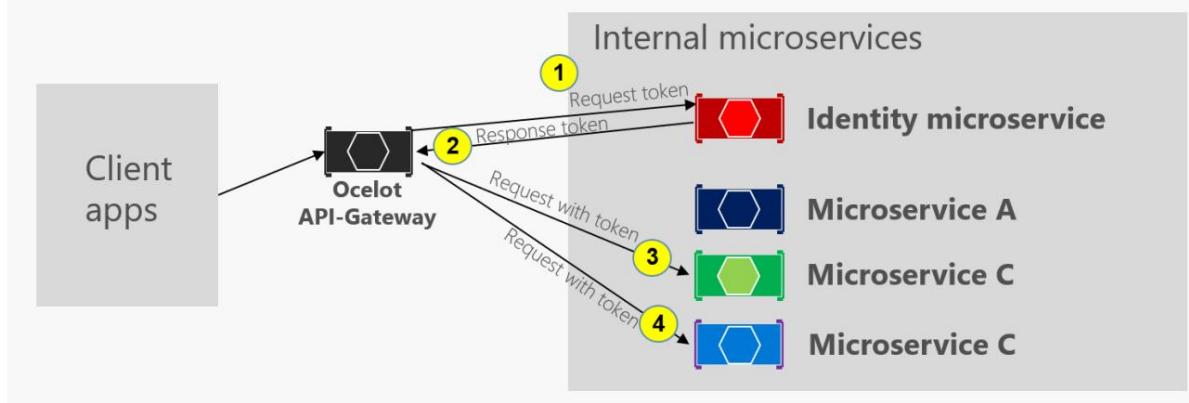


Figura 6-40. Autenticación en Ocelot

Como muestra el diagrama anterior, cuando el microservicio de identidad está debajo de la puerta de enlace API (AG): 1) AG solicita un token de autenticación del microservicio de identidad, 2) El microservicio de identidad devuelve el token a AG, 3-4) Solicitudes de AG de microservicios utilizando el token de autenticación Debido a que la aplicación eShopOnContainers ha dividido API Gateway en múltiples BFF (Backend para Frontend) y API Gateways de áreas comerciales, otra opción habría sido crear una API Gateway adicional para cuestiones transversales. Esa elección sería justa en una arquitectura basada en microservicios más compleja con múltiples microservicios transversales. Dado que solo hay una preocupación transversal en eShopOnContainers, se decidió manejar el servicio de seguridad fuera del ámbito de API Gateway, por motivos de simplicidad.

En cualquier caso, si la aplicación está asegurada en el nivel de API Gateway, el módulo de autenticación de Ocelot API Gateway se visita primero cuando se intenta usar cualquier microservicio seguro. Eso redirige la solicitud HTTP para visitar el microservicio de identidad o autenticación para obtener el token de acceso para que pueda visitar los servicios protegidos con access\_token.

La forma de proteger con autenticación cualquier servicio en el nivel de puerta de enlace API es establecer AuthenticationProviderKey en su configuración relacionada en el archivo configuration.json.

```
{
    "DownstreamPathTemplate": "/api/{versión}/{todo}",
    "EsquemaDescendente": "http",
    "DownstreamHostAndPorts": [
        {
            "Anfitrión": "cesta-api",
            "Puerto": 80
        }
    ],
    "UpstreamPathTemplate": "/api/{versión}/b/{todo}",
    "UpstreamHttpMethod": [],
    "Opciones de autenticación": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "Ámbitos permitidos": []
    }
}
```

Cuando se ejecuta Ocelot, buscará ReRoutes AuthenticationOptions.AuthenticationProviderKey y verificará que haya un proveedor de autenticación registrado con la clave dada. Si no lo hay, entonces Ocelot no se iniciará. Si lo hay, ReRoute utilizará ese proveedor cuando se ejecute.

Debido a que OcelotWebHost está configurado con authenticationProviderKey = "IdentityApiKey", requerirá autenticación cada vez que ese servicio tenga solicitudes sin ningún token de autenticación.

```
espacio de nombres OcelotApiGw
{
    Inicio de clase pública {

        IConfiguration privado de solo lectura _cfg;

        inicio público (configuración de IConfiguration) => _cfg = configuración;

        public void ConfigureServices(servicios IServiceCollection) {

            var IdentityUrl = _cfg.GetValue<string>("IdentityUrl"); var authenticationProviderKey
            = "IdentityApiKey"; //... servicios.AddAuthentication()

            .AddJwtBearer(authenticationProviderKey, x => {

                x.Autoridad = identidadUrl;
                x.RequireHttpsMetadata = falso;
                x.TokenValidationParameters = nuevo

            Microsoft.IdentityModel.Tokens.TokenValidationParameters()
            {
                Audiencias válidas = new[] { "pedidos", "canasta", "ubicaciones", "marketing",
                "mobileshoppingagg", "webshoppingagg" }
            };
        });
    }
}
```

```
//...
}
}
```

Luego, también debe configurar la autorización con el atributo [Autorizar] en cualquier recurso al que se acceda como los microservicios, como en el siguiente controlador de microservicios Basket.

```
espacio de nombres Microsoft.eShopOnContainers.Services.Basket.API.Controllers {
    [Ruta("api/v1/[controlador]")]
    [Autorizar]
    clase pública BasketController: Controlador {
        //...
    }
}
```

Las audiencias válidas como "canasta" se correlacionan con la audiencia definida en cada microservicio con AddJwtBearer() en ConfigureServices() de la clase Startup, como en el código a continuación.

```
// evitar que se asigne el reclamo "sub" al identificador de nombre.
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

var IdentityUrl = Configuration.GetValue<string>("IdentityUrl");

services.AddAuthentication(opciones => {
    opciones.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme ;
    opciones.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme ;

}).AddJwtBearer(opciones => {
    opciones.Autoridad = identidadUrl;
    opciones.RequireHttpsMetadata = falso;
    opciones.Audiencia = "cesta";
});
```

Si intenta acceder a cualquier microservicio seguro, como el microservicio Basket con una URL ReRoute basada en API Gateway como http://host.docker.internal:5202/api/v1/b-basket/1, obtendrá un 401 No autorizado a menos que proporcione un token válido. Por otro lado, si se autentica una URL de ReRoute, Ocelot invocará cualquier esquema descendente asociado con ella (la URL interna del microservicio).

Autorización en el nivel ReRoutes de Ocelot. Ocelot admite la autorización basada en notificaciones evaluada después de la autenticación. Establece la autorización a nivel de ruta agregando las siguientes líneas a la configuración de ReRoute.

```
"RouteClaimsRequirement": {
    "UserType": "empleado"
}
```

En ese ejemplo, cuando se llama al middleware de autorización, Ocelot encontrará si el usuario tiene el tipo de reclamo 'UserType' en el token y si el valor de ese reclamo es 'empleado'. Si no es así, entonces el usuario no estará autorizado y la respuesta será 403 prohibida.

## Uso de Kubernetes Ingress más Ocelot API Gateways

Cuando usa Kubernetes (como en un clúster de Azure Kubernetes Service), generalmente unifica todas las solicitudes HTTP a través del [nivel de ingreso de Kubernetes](#) basado en Nginx.

En Kubernetes, si no usa ningún enfoque de ingreso, sus servicios y pods tienen direcciones IP que solo puede enrutar la red del clúster.

Pero si usa un enfoque de ingreso, tendrá un nivel intermedio entre Internet y sus servicios (incluidos sus API Gateways), que actuará como un proxy inverso.

Como definición, un Ingress es una colección de reglas que permiten que las conexiones entrantes lleguen a los servicios del clúster. Un ingreso está configurado para proporcionar servicios URL accesibles externamente, tráfico de equilibrio de carga, terminación SSL y más. Los usuarios solicitan el ingreso enviando el recurso de ingreso al servidor API.

En eShopOnContainers, al desarrollar localmente y usar solo su máquina de desarrollo como host de Docker, no está usando ningún ingreso, sino solo las múltiples puertas de enlace API.

Sin embargo, cuando se dirige a un entorno de "producción" basado en Kubernetes, eShopOnContainers utiliza una entrada frente a las puertas de enlace API. De esa forma, los clientes siguen llamando a la misma URL base, pero las solicitudes se enrutan a varias puertas de enlace API o BFF.

Las puertas de enlace API son front-end o fachadas que muestran solo los servicios pero no las aplicaciones web que generalmente están fuera de su alcance. Además, las API Gateways pueden ocultar ciertos microservicios internos.

El ingreso, sin embargo, solo está redirigiendo las solicitudes HTTP, pero no intenta ocultar ningún microservicio o sitio web.

aplicación

Tener un nivel de entrada de Nginx en Kubernetes frente a las aplicaciones web más varios Ocelot API Gateways/BFF es la arquitectura ideal, como se muestra en el siguiente diagrama.

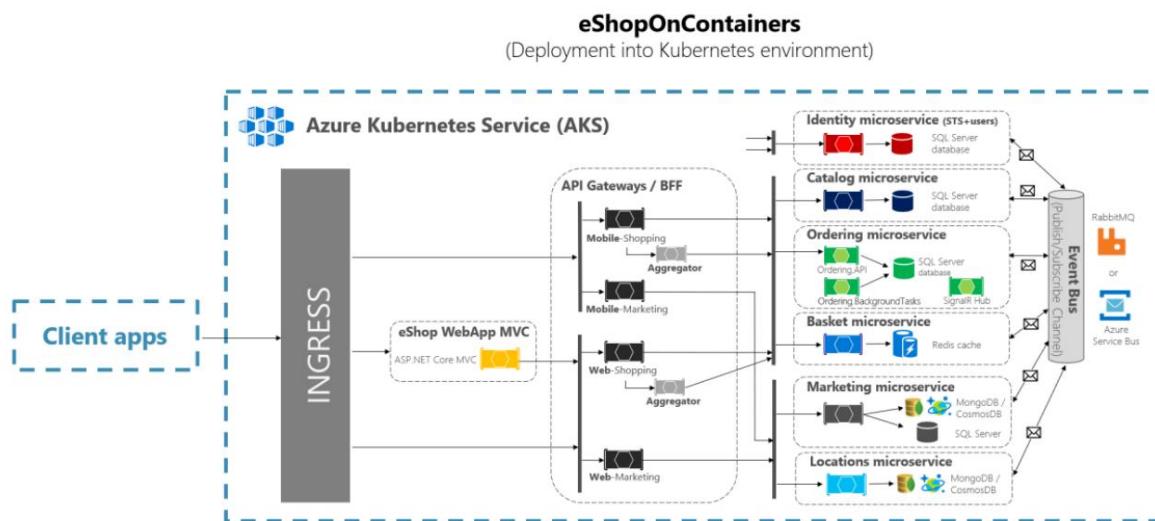


Figura 6-41. El nivel de ingreso en eShopOnContainers cuando se implementa en Kubernetes

Un ingreso de Kubernetes actúa como un proxy inverso para todo el tráfico a la aplicación, incluidas las aplicaciones web, que están fuera del alcance de la puerta de enlace API. Cuando implementa eShopOnContainers en Kubernetes,

expone solo algunos servicios o puntos finales a través del ingreso, básicamente la siguiente lista de sufijos en el URL:

- / para la aplicación web cliente SPA /
- webmvc para la aplicación web cliente MVC /
- webstatus para la aplicación web cliente que muestra el estado/chequeos de
- salud /webshoppingapigw para la web BFF y procesos comerciales de compras /
- webmarketingapigw para la web BFF y procesos comerciales de marketing /
- mobileshoppingapigw para el BFF móvil y los procesos comerciales de compras /
- mobilemarketingapigw para el BFF móvil y los procesos comerciales de marketing

Cuando se implementa en Kubernetes, cada Ocelot API Gateway usa un archivo "configuration.json" diferente para cada pod que ejecuta API Gateways. Esos archivos "configuration.json" se proporcionan al montar (originalmente con el script deployment.ps1) un volumen creado en base a un mapa de configuración de Kubernetes llamado 'ocelot'.

Cada contenedor monta su archivo de configuración relacionado en la carpeta del contenedor denominada /app/configuration.

En los archivos de código fuente de eShopOnContainers, los archivos "configuration.json" originales se pueden encontrar dentro de la carpeta k8s/ocelot/. Hay un archivo para cada BFF/APIGateway.

## Funciones transversales adicionales en Ocelot API Gateway

Hay otras características importantes para investigar y usar, cuando se usa un Ocelot API Gateway, que se describen en los siguientes enlaces.

- Detección de servicios en el lado del cliente integrando Ocelot con Consul o Eureka <https://ocelot.readthedocs.io/en/latest/features/servicediscovery.html>
- Almacenamiento en caché en el nivel API Gateway <https://ocelot.readthedocs.io/en/latest/features/caching.html>
- Registro en el nivel de API Gateway <https://ocelot.readthedocs.io/en/latest/features/logging.html>
- Calidad de servicio (reintentos y disyuntores) en el nivel API Gateway <https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html>
- Limitación de velocidad <https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html>

# Abordar el negocio

## Complejidad en un

## Microservicio con patrones

## DDD y CQRS

Diseñe un modelo de dominio para cada microservicio o contexto acotado que refleje la comprensión del dominio empresarial.

Esta sección se centra en los microservicios más avanzados que implementa cuando necesita abordar subsistemas complejos o microservicios derivados del conocimiento de expertos en dominios con reglas comerciales en constante cambio. Los patrones de arquitectura utilizados en esta sección se basan en enfoques de diseño dirigido por dominio (DDD) y comando y consulta de responsabilidad segregación (CQRS), como se ilustra en la figura 7-1.

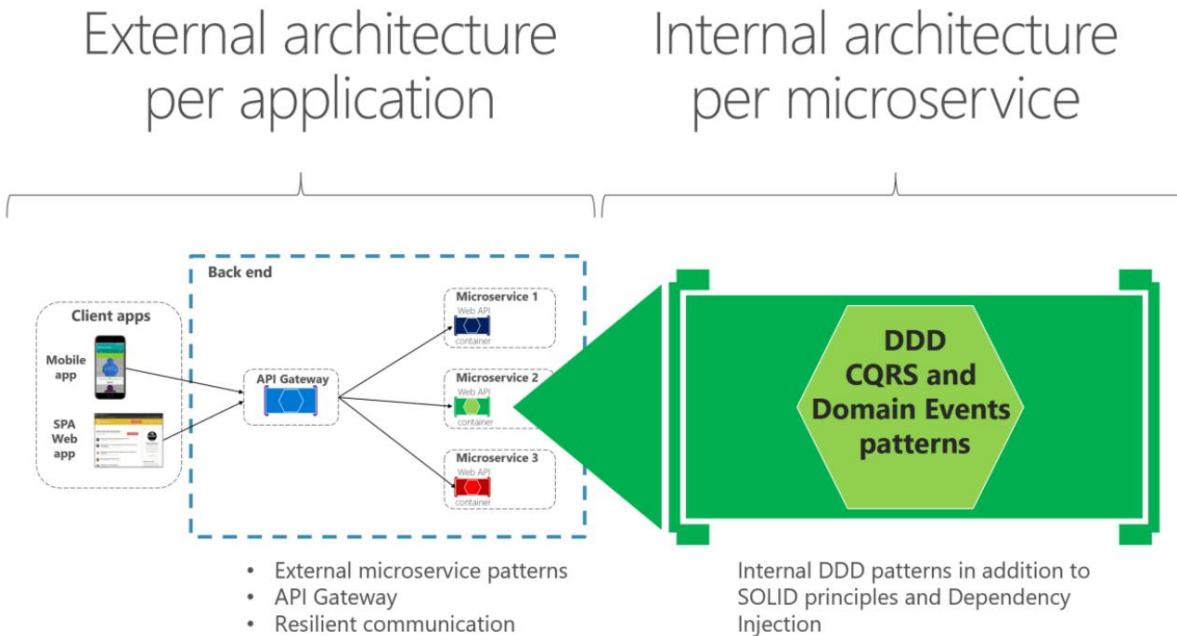


Figura 7-1. Arquitectura de microservicio externa frente a patrones de arquitectura interna para cada microservicio

Sin embargo, la mayoría de las técnicas para los microservicios basados en datos, como por ejemplo, cómo implementar un servicio de API web de ASP.NET Core o cómo exponer los metadatos de Swagger con Swashbuckle o NSwag, también son aplicables a los microservicios más avanzados implementados internamente con patrones DDD. Esta sección es una extensión de las secciones anteriores, porque la mayoría de las prácticas explicadas anteriormente también se aplican aquí o para cualquier tipo de microservicio.

En primer lugar, esta sección proporciona detalles sobre los patrones CQRS simplificados utilizados en la aplicación de referencia eShopOnContainers. Más adelante, obtendrá una descripción general de las técnicas DDD que le permitirán encontrar patrones comunes que puede reutilizar en sus aplicaciones.

DDD es un tema amplio con un rico conjunto de recursos para el aprendizaje. Puede comenzar con libros como [Domain Driven Design](#) de Eric Evans y materiales adicionales de Vaughn Vernon, Jimmy Nilsson, Greg Young, Udi Dahan, Jimmy Bogard y muchos otros expertos en DDD/CQRS. Pero, sobre todo, debe tratar de aprender cómo aplicar las técnicas DDD a partir de las conversaciones, la pizarra y las sesiones de modelado de dominio con los expertos en su dominio comercial concreto.

## Recursos adicionales

### DDD (diseño dirigido por dominio)

- Erick Evans. Idioma del dominio  
<https://domainlanguage.com/>
- Martín Fowler. Diseño basado en dominios <https://martinfowler.com/tags/domain%20driven%20design.html>
- Jimmy Bogard. Fortalecimiento de su dominio: una introducción  
<https://lostechies.com/jimmybogard/2010/02/04/fortalecimiento-de-su-dominio-a-primer/>

## libros

- Erick Evans. Diseño basado en dominios: abordar la complejidad en el corazón del software  
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>
- Erick Evans. Referencia de diseño controlado por dominio: definiciones y resúmenes de patrones  
<https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-2014-09-22/dp/B01N8YB4ZO/>
- Vaughn Vernon. Implementación del diseño basado en dominios  
<https://www.amazon.com/Implementación-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/>
- Vaughn Vernon. Diseño impulsado por dominio destilado  
<https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420/>
- Jimmy Nilsson. Aplicación de diseños y patrones basados en dominios  
<https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202/>
- César de la Torre. Guía de arquitectura orientada al dominio N-Layered con .NET  
<https://www.amazon.com/N-Layered-Domain-Oriented-Architecture-Guide-NET/dp/8493903612/>
- Abel Avram y Floyd Marinescu. Diseño controlado por dominio rápidamente  
<https://www.amazon.com/Domain-Driven-Design-Quickly-Abel-Avram/dp/1411609255/>
- Scott Millett, Nick Tune: patrones, principios y prácticas del diseño basado en dominios  
<https://www.wiley.com/Patterns%2C+Principles%2C+and+Practices+of+Domain+Driven+Design-9781118714706>

## entrenamiento DDD

- Julie Lerman y Steve Smith. Fundamentos del diseño basado en dominios  
<https://bit.ly/PS-DDD>

## Aplicar patrones CQRS y DDD simplificados en un microservicio

CQRS es un patrón arquitectónico que separa los modelos para leer y escribir datos. El término relacionado [Command Query Separation \(CQS\)](#) fue definido originalmente por Bertrand Meyer en su libro [Object Oriented Software Construction](#). La idea básica es que puede dividir las operaciones de un sistema en dos categorías claramente separadas:

- Consultas. Estas consultas devuelven un resultado y no cambian el estado del sistema, y no tienen efectos secundarios.
- Comandos. Estos 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 el estado o muta el estado, pero no ambos. Incluso un solo objeto de patrón de repositorio puede cumplir con CQS. CQS puede considerarse un principio fundamental para CQRS.

[La segregación de responsabilidad de comando y consulta \(CQRS\)](#) fue presentada por Greg Young y Udi Dahan y otros la promovieron fuertemente. Se basa en el principio CQS, aunque es más detallado. Puede considerarse un patrón basado en comandos y eventos más, opcionalmente, en mensajes asíncronos. En muchos casos, CQRS está relacionado con escenarios más avanzados, como tener una base de datos física diferente para lecturas (consultas) que para escrituras (actualizaciones). Además, un sistema CQRS más evolucionado podría implementar [Event-Sourcing \(ES\)](#) para su base de datos de actualizaciones, por lo que solo almacenaría eventos en el modelo de dominio en lugar de almacenar los datos del estado actual. Sin embargo, este enfoque no se utiliza en esta guía. Esta guía utiliza el enfoque CQRS más simple, que consiste simplemente en separar las consultas de los comandos.

El aspecto de separación de CQRS se logra agrupando las operaciones de consulta en una capa y los comandos en otra capa. Cada capa tiene su propio modelo de datos (tenga en cuenta que decimos modelo, no necesariamente una base de datos diferente) y se crea utilizando su propia combinación de patrones y tecnologías. Más importante aún, las dos capas pueden estar dentro del mismo nivel o microservicio, como en el ejemplo (pedir microservicio) utilizado para esta guía. O podrían implementarse en diferentes microservicios o procesos para que puedan optimizarse y escalarse por separado sin afectarse entre sí.

CQRS significa tener dos objetos para una operación de lectura/escritura donde en otros contextos hay uno. Hay razones para tener una base de datos de lecturas desnormalizadas, sobre las que puede obtener información en la literatura CQRS más avanzada. Pero no estamos usando ese enfoque aquí, donde el objetivo es tener más flexibilidad en las consultas en lugar de limitar las consultas con restricciones de patrones DDD como agregados.

Un ejemplo de este tipo de servicio es el microservicio de pedidos de la aplicación de referencia eShopOnContainers. Este servicio implementa un microservicio basado en un enfoque CQRS simplificado. Utiliza una sola fuente de datos o base de datos, pero dos modelos lógicos más patrones DDD para el dominio transaccional, como se muestra en la Figura 7-2.