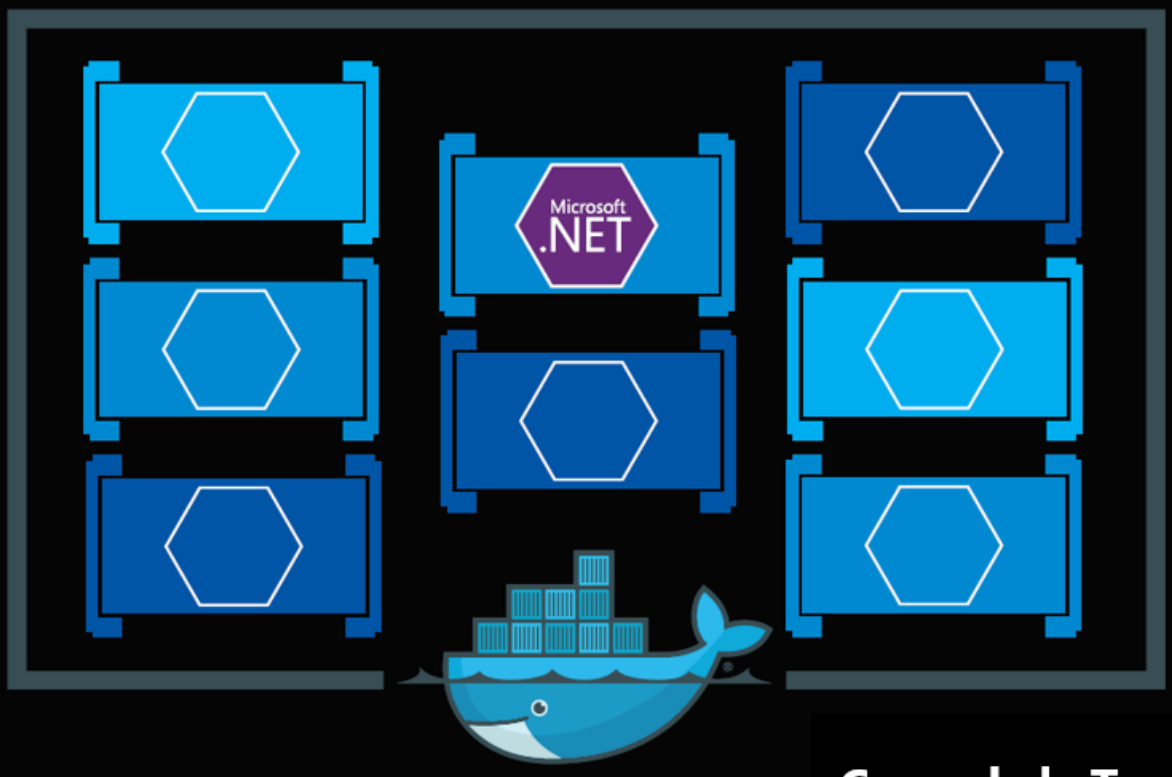
2da Edición
(Versión 2.0.5)
Soporte para .NET Core 2

Microsoft

Microservicios .NET: Arquitectura para Aplicaciones .NET Contenerizadas



Cesar de la Torre
Bill Wagner
Mike Rousos

Microsoft Corporation

EDICIÓN v2.0.5

DESCARGA disponible en: https://aka.ms/microservicesebook-es-es

PUBLICADO POR

Microsoft Developer Division, .NET and Visual Studio product teams
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Co-Autores:

Cesar de la Torre, Sr. PM, .NET product team, Microsoft Corp.

Bill Wagner, Sr. Content Developer, C+E, Microsoft Corp.

Mike Rousos, Principal Software Engineer, DevDiv CAT team, Microsoft

Editores:

Mike Pope

Steve Hoag

Participantes y revisores:

Jeffrey Ritcher, Partner Software Eng, Azure team, Microsoft

Jimmy Bogard, Chief Architect at Headspring

Udi Dahan, Founder & CEO, Particular Software

Jimmy Nilsson, Co-founder and CEO of Factor10

Glenn Condron, Sr. Program Manager, ASP.NET team

Mark Fussell, Principal PM Lead, Azure Service Fabric team, Microsoft

Diego Vega, PM Lead, Entity Framework team, Microsoft

Barry Dorrans, Sr. Security Program Manager

Rowan Miller, Sr. Program Manager, Microsoft

Ankit Asthana, Principal PM Manager, .NET team, Microsoft

Scott Hunter, Partner Director PM, .NET team, Microsoft

Dylan Reisenberger, Architect and Dev Lead at Polly

Steve Smith, Software Craftsman & Trainer at ASPSmith Ltd.

Ian Cooper, Coding Architect at Brighter

Unai Zorrilla, Architect and Dev Lead at Plain Concepts

Eduard Tomas, Dev Lead at Plain Concepts

Ramon Tomas, Developer at Plain Concepts

David Sanz, Developer at Plain Concepts

Javier Valero, Chief Operating Officer at Grupo Solutio

Pierre Millet, Sr. Consultant, Microsoft

Michael Friis, Product Manager, Docker Inc

Charles Lowell, Software Engineer, VS CAT team, Microsoft

Miguel Veloso, Sr. Consultant at Turing Challenge

Traductor:

Miguel Veloso, Sr. Consultant at Turing Challenge

Contenido

Introducción

Cada vez más, las empresas están logrando ahorros en los costes, resolviendo problemas de despliegue de las aplicaciones y mejorando las actividades de desarrollo y operaciones, con la utilización de contenedores. Microsoft ha desarrollado y liberado innovaciones en contenedores para Windows y para Linux, creando productos como como el Azure Container Service y el Azure Service Fabric y aliándose con líderes de la industria como Docker, Mesosphere y Kubernetes. Estas compañías proveen soluciones de contenedores que ayudan a las empresas a construir y desplegar aplicaciones a la velocidad y escala que se manejan en la nube, sin importar las plataformas o herramientas que utilicen.

Docker se ha convertido en el estándar de facto en la industria de contenedores, siendo soportado por las empresas más significativas en los ecosistemas de Windows y Linux. (Microsoft es uno de los principales proveedores de la nube que soportan Docker.) En el futuro, Docker será omnipresente en cualquier centro de datos en la nube o en instalaciones internas (on-premises).

Además, la arquitectura de microservicios está surgiendo como un enfoque importante para las aplicaciones distribuidas de misión-crítica. En una arquitectura basada en microservicios, la aplicación se construye como una colección de servicios que se pueden desarrollar, probar, desplegar y versionar de forma independiente.

Acerca de esta guía

Esta guía es una introducción al desarrollo de aplicaciones basadas en microservicios y en su gestión usando contenedores. Se discuten enfoques arquitectónicos y de implementación con .NET Core y contenedores Docker. Con el fin de facilitar el inicio con contenedores y microservicios, la guía se enfoca en una aplicación de referencia basada en microservicios y desplegada en contenedores, que puede explorar fácilmente. La aplicación de ejemplo está disponible en el repositorio eShopOnContainers en GitHub.

Esta guía provee una orientación en cuanto a fundamentos de desarrollo y arquitectura, principalmente en el ámbito del desarrollo, enfocándose en dos tecnologías base: Docker y .NET Core. Nuestra intención es que lea la guía cuando esté trabajando en el diseño de su aplicación, sin preocuparse por la infraestructura del entorno de producción (sin importar si será en la nube o interno). Las decisiones de infraestructura se realizarán más tarde, cuando al desarrollar aplicaciones a nivel de producción real. Por lo tanto, está guía pretende ser agnóstica de la infraestructura y estar más orientada hacia el entorno de desarrollo.

Al terminar el estudio de esta guía, su próximo paso debería ser aprender acerca de microservicios a nivel de producción en Microsoft Azure.

Versión

Está guía ha sido actualizada para cubrir la versión **.NET Core 2**, así como muchas actualizaciones relacionadas con tecnologías de misma "onda" (esto es Azure y otras tecnologías de terceros) que se han liberado simultáneamente con .NET Core 2.

Lo que no se cubre en esta guía

Esta guía no se enfoca en el ciclo de vida de las aplicaciones, es decir, DevOps, CI/CD *pipelines* (Líneas de procesos de integración y entrega continua) o trabajo en equipo. La guía complementaria Containerized Docker Application Lifecycle with Microsoft Platform and Tools se enfoca en esos temas.

Tampoco se cubre el despliegue de aplicaciones web ASP.NET Core basadas en un solo contendor o la migración de aplicaciones *legacy* monolíticas, en .NET Framework, a contenedores Windows. La guía complementaria Modernize existing .NET Framework applications with Azure and Windows Containers.

Tampoco se incluyen detalles de implementación en Azure, tal como información sobre orquestadores específicos.

Recursos adicionales

- **Containerized Docker Application Lifecycle with Microsoft Platform and Tools** (eBook)
  https://aka.ms/dockerlifecycleebook
- **eBook: Modernize existing .NET Framework applications with Azure and Windows Containers**
  https://aka.ms/liftandshiftwithcontainersebook

Quién debería usar esta guía

Escribimos está guía para desarrolladores y arquitectos que quieren o están comenzando en el desarrollo de soluciones basadas en Docker y la arquitectura basada en micro servicios. Esta guía le ayudará si quiere aprender cómo definir la arquitectura, diseñar e implementar aplicaciones, como pruebas de concepto, con las tecnologías de desarrollo de Microsoft (especialmente con .NET Core) y con contenedores Docker.

También la encontrará útil si es responsable de decisiones técnicas, como un arquitecto empresarial, que desea tener una visión general de la arquitectura y la tecnología, antes de seleccionar el enfoque adecuado para nuevas aplicaciones modernas y distribuidas.

Cómo debe usar esta guía

La primera parte de esta guía presenta los contenedores Docker y la discusión sobre cómo seleccionar entre .NET Core y .NET Framework, como *framework* de desarrollo y también presenta una visión general de los microservicios. Este contenido es para arquitectos y responsables de las decisiones técnicas que quieren tener una visión general pero no necesitan enfocarse en los detalles de implementación.

La segunda parte de la guía comienza con la sección El proceso de desarrollo de aplicaciones basadas en Docker. Se enfoca en patrones de desarrollo y microservicios para implementar aplicaciones usando .NET Core y Docker. Esta sección será muy interesante para arquitectos y desarrolladores que se quieran enfocar en el código y en los patrones, así como en los detalles de implementación.

Aplicación de referencia basada en microservicios y contenedores: eShopOnContainers

La aplicación eShopOnContainers es una aplicación de referencia open source para .NET Core y microservicios, diseñada para ser desplegada en contenedores Docker. La aplicación contiene múltiples sub-sistemas, incluyendo varios tipos de interfaces de usuario para la tienda en línea (una aplicación web y una aplicación móvil nativa). También incluye un *back-end* de microservicios y contenedores para todas las operaciones en el servidor.

Envíenos sus comentarios

Escribimos esta guía para ayudarle a entender la arquitectura de aplicaciones basada en microservicios y contenedores en .NET. Tanto la guía como la aplicación de referencia estarán evolucionando continuamente, así que le agradecemos sus comentarios. Si tiene alguna opinión o sugerencia para mejorar esta guía, por favor envíelas a:

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

Introducción a los Contenedores y a Docker

La **contenerización**, como la llamaremos en esta guía (del inglés *containerization*), es un enfoque de desarrollo de software en el que una aplicación o servicio, sus dependencias y su configuración (en forma de ficheros de manifiesto para despliegue) se empaquetan juntos como una **imagen de contenedor** (o simplemente **imagen** para simplificar, cuando no haya ambigüedad). Una aplicación basada en contenedores se puede probar como una unidad y desplegar como una instancia de la imagen, en el sistema operativo que funciona como *host*.

Así como los contenedores de carga facilitan el transporte por barco, tren o camión, sin importar el contenido, los contenedores de software funcionan como una unidad estándar que puede agrupar distintos programas y dependencias. La contenerización permite que los desarrolladores y los responsables de infraestructura puedan desplegar aplicaciones o servicios en diferentes entornos, con pocas o ninguna modificación.

Los contenedores también aíslan las aplicaciones de otras instaladas en un sistema operativo (SO) compartido. Las aplicaciones contenerizadas se ejecutan en un *host* de contenedores que, a su vez, se ejecuta sobre el sistema operativo (Linux o Windows). Por eso, los contenedores son significativamente más livianos que las imágenes de máquinas virtuales (VM).

Cada contenedor puede ejecutar una aplicación o servicio, como se muestra en la figura 2-1. En este ejemplo en *host* Docker es un *host* de contenedores y App1, App2, Svc1 y Svc2 son aplicaciones y servicios contenerizados.

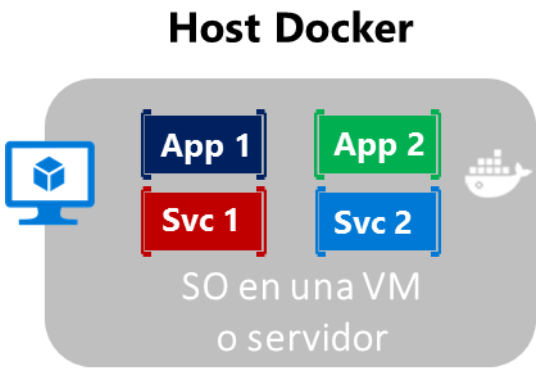


***Figura 2-1**. Múltiples contenedores corriendo en un host de contenedores*

Otro beneficio de la contenerización es la escalabilidad. Se puede ampliar rápidamente la capacidad creando nuevos contenedores para cargas o tareas de corta duración. Desde el punto de vista de la aplicación, el instanciar una imagen (crear un contenedor) es similar a instanciar un proceso como un servicio o una aplicación web. Sin embargo, por razones de confiabilidad y disponibilidad, cuando se ejecutan múltiples instancias de una imagen, típicamente se distribuyen entre varios servidores o máquinas virtuales en diferentes dominios de falla.

En resumen, los contenedores ofrecen los beneficios de aislamiento, portabilidad, agilidad, escalabilidad y control, a lo largo de todo el ciclo de vida de la aplicación. El beneficio más importante es la separación o independencia de la aplicación de los entornos de desarrollo (Dev) y operaciones (Ops).

¿Qué es Docker?

Docker es un proyecto open-source para automatizar el despliegue de aplicaciones como contenedores independientes y autosuficientes, que se pueden ejecutar en la nube o en servidores internos. Docker es también una compañía que promociona y desarrolla esta tecnología, colaborando con varios proveedores de Linux, Windows y plataformas en la nube, incluyendo a Microsoft.

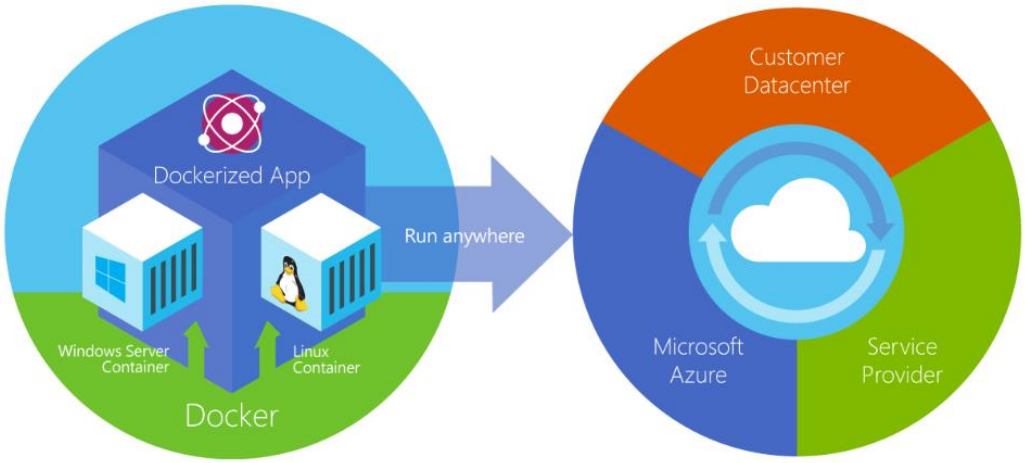


***Figura 2-2**. Con Docker se pueden desplegar contenedores en todas las capas de la nube híbrida*

Las imágenes Docker pueden correr de forma nativa en Linux o Windows, sin embargo, las imágenes Windows sólo se pueden ejecutar en hosts Windows y las imágenes Linux sólo en hosts Linux, entendiendo *host* como un servidor o una máquina virtual.

El entorno de desarrollo puede ser Windows, Linux o macOS. En la máquina de desarrollo se ejecuta un *host* Docker, donde se despliegan las imágenes Docker, incluyendo la aplicación o servicio y sus dependencias. Para desarrollar en Linux o Mac se utiliza un *host* Docker basado en Linux y sólo se pueden crear imágenes para contenedores Linux. (En la Mac se puede editar código y ejecutar la interfaz de comandos de Docker (Docker CLI) desde macOS, pero, al menos hasta este momento, no se pueden ejecutar contenedores directamente en macOS). Desde Windows sí se pueden crear imágenes de contenedores tanto para Linux como para Windows.

Para manejar contenedores en entornos de desarrollo y manejar herramientas adicionales para el desarrollo, Docker provee el Docker Community Edition (CE) tanto para Windows como para macOS.

Este producto instala la máquina virtual (el *host* Docker) necesaria para manejar los contenedores. Docker también tiene disponible la versión Docker Enterprise Edition (EE), que está diseñada para los entornos de desarrollo empresarial y se usa para desarrollar, desplegar y ejecutar grandes aplicaciones críticas para el negocio, en entornos de producción.

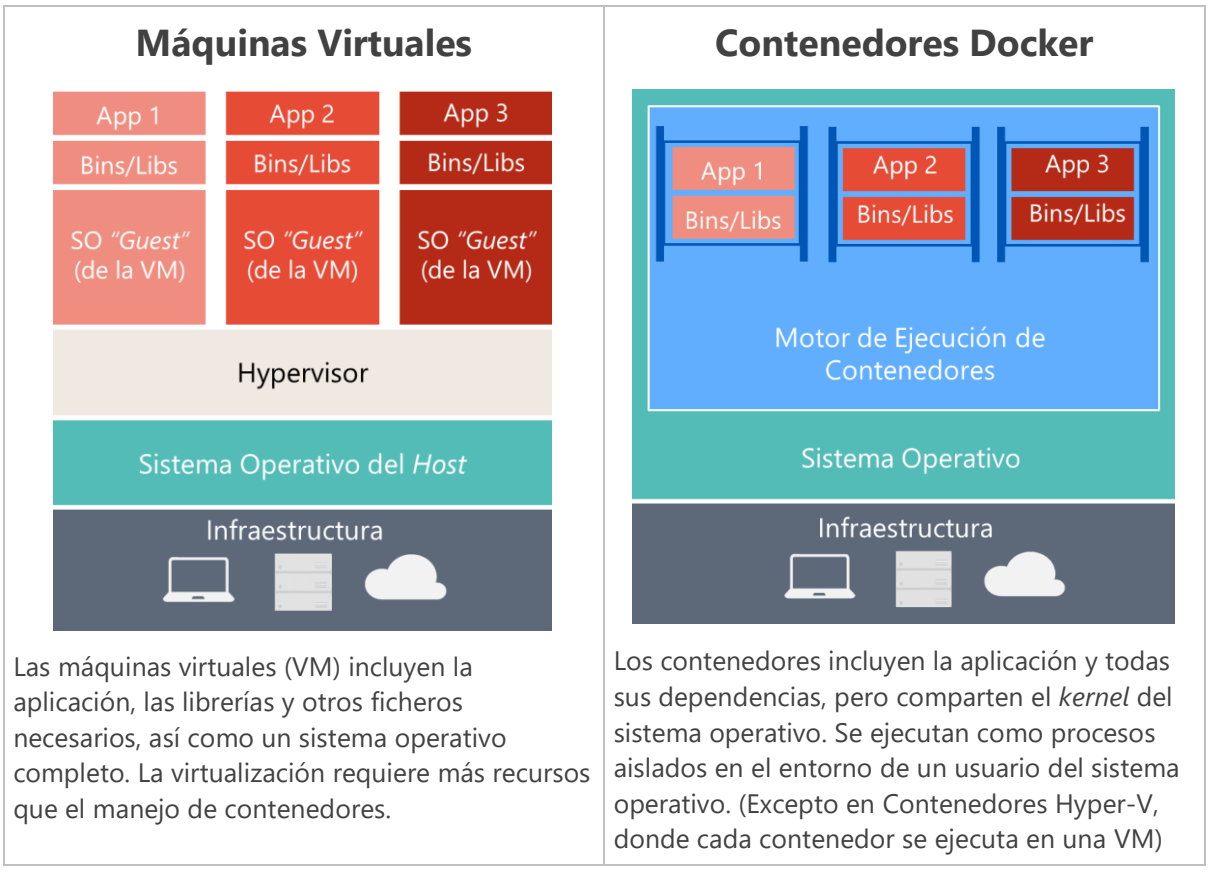
Hay dos tipos de motor de ejecución (*runtime*) para Contenedores de Windows:

- Contendores Windows Server, que proveen aislamiento de las aplicaciones con tecnologías de asilamiento de procesos y *namespaces*. Un Contenedor Windows Server comparte un núcleo de proceso (*kernel*) con el *host* de contenedores y con todos los contenedores que se ejecutan en el *host*.
- Contenedores Hyper-V, que aumentan el aislamiento ofrecido por Contenedores Windows Server, ejecutando cada contenedor en una máquina virtual optimizada. En esta configuración el *kernel* del *host* de contenedores no se comparte con los Contenedores Hyper-V, lo que ofrece mayor aislamiento.

Las imágenes para estos contenedores se crean y funcionan de la misma forma. La diferencia está en cómo se crea el contenedor a partir de la imagen, cuando se ejecuta un Contenedor Hyper-V hace falta un parámetro adicional. Para más detalle, consulte Contenedores de Hyper-V.

Comparando contenedores Docker con máquinas virtuales

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



Las máquinas virtuales (VM) incluyen la aplicación, las librerías y otros ficheros necesarios, así como un sistema operativo completo. La virtualización requiere más recursos que el manejo de contenedores.

Los contenedores incluyen la aplicación y todas sus dependencias, pero comparten el *kernel* del sistema operativo. Se ejecutan como procesos aislados en el entorno de un usuario del sistema operativo. (Excepto en Contenedores Hyper-V, donde cada contenedor se ejecuta en una VM)

***Figura 2-3**. Comparación entre las máquinas virtuales tradicionales y los contenedores Docker*

Como los contenedores requieren muchos menos recursos (por ejemplo, no necesitan un sistema operativo completo) son fáciles de desplegar y arrancan rápido. Esto permite tener una mayor densidad, lo que significa que se pueden ejecutar más servicios en el mismo equipo y, por lo tanto, reduce los costes.

Como efecto secundario, al compartir el mismo núcleo del sistema operativo, se tiene menos aislamiento que en las máquinas virtuales.

El propósito de una imagen es hacer que el entorno (es decir, las dependencias) de la aplicación sea el mismo para todos los despliegues. Esto significa que puede depurar una imagen en su máquina y luego desplegarla en otra, asegurando que se mantiene el mismo entorno.

Una imagen es una forma de empaquetar una aplicación o servicio para desplegarlo de forma confiable y reproducible. Se podría decir que Docker no es sólo una tecnología sino también una filosofía y un proceso.

Cuando se usa Docker no escuchará a los desarrolladores decir "funciona en mi máquina, ¿por qué no en producción?". Simplemente dirán "Funciona en Docker", porque una aplicación empaquetada en Docker se puede ejecutar en cualquier entorno soportado por Docker y se ejecutará de la forma prevista en todos los entornos de despliegue (Desarrollo, control de calidad, pre-producción (*staging*) y producción).

Terminología Docker

En esta sección se presentan los términos y definiciones que debe manejar antes de entrar en profundidades con Docker. Para más definiciones, consulte el glosario provisto por Docker (https://docs.docker.com/glossary/).

Nota: En la traducción se mantienen muchos términos en inglés, porque son de uso habitual y el traducirlos podría más bien crear una barrera a la comunicación y comprensión.

Imagen de Contenedor (Container Image/Docker Image): O simplemente **imagen**, cuando no haya ambigüedad en el contexto. Un paquete con todas las dependencias e información necesaria para crear un contenedor. La imagen contiene todas las dependencias (como *frameworks*, por ejemplo) así como toda la configuración de despliegue y ejecución que será utilizada por el contenedor en tiempo de ejecución. Usualmente una imagen se deriva de otras imágenes base, que forman capas apiladas para conformar el sistema de ficheros del contenedor. Una imagen es inmutable, es decir, no se puede modificar después de crearla.

Contenedor (Container): Una instancia de una imagen (*Docker Image*). Un contenedor representa la ejecución de una sola aplicación, proceso o servicio. Están formados por el contenido de la imagen Docker, el entorno de ejecución y un conjunto estándar de instrucciones. Cuando se escala un servicio, se crean varios contenedores a partir de la misma imagen, pasando parámetros diferentes a cada instancia. También, un programa batch puede crear múltiples contenedores a partir de la misma imagen, pasando parámetros diferentes a cada instancia.

Etiqueta (Tag): Un elemento que se puede aplicar a las imágenes para identificar versiones o entornos de destino.

Dockerfile: Un fichero de texto que contiene instrucciones sobre cómo construir la imagen Docker.

Construir (*Build*): La acción de preparar una imagen Docker según la información y contexto definido en su **`Dockerfile`**, más los ficheros adicionales de la carpeta donde se prepara la imagen. Para preparar imágenes Docker se utiliza el comando: **`docker build`**.

Repositorio (Repo): Una colección de imágenes Docker, identificadas con una etiqueta para diferenciar la versión. Algunos repositorios contienen múltiples variantes de una imagen específica, por ejemplo, imágenes con SDK (más pesadas) o imágenes sólo con entorno de ejecución (más livianas), etc. Todas esas variantes se identifican con etiquetas. Puede haber variantes por plataforma (imágenes Linux y Windows) en un mismo repositorio.

Registro (Registry): Un servicio que provee el acceso a los repositorios. El registro por defecto para la mayoría de las imágenes públicas en Docker Hub (mantenido por la organización Docker). Un registro usualmente contiene repositorios de varios equipos de trabajo. Las empresas frecuentemente tienen registros privados para gestionar las imágenes que han creado. Otro ejemplo de registro es el Azure Container Registry.

Docker Hub: Un registro público para gestionar imágenes y trabajar con ellas. Docker Hub provee almacenamiento de las imágenes, registros públicos y privados, así como *web hooks*, *triggers* e integración con GitHub y Bitbucket.

Azure Container Registry: Un recurso público para trabajar con imágenes Docker y sus componentes en Azure. Este provee un registro cercano a los despliegues en Azure y permite controlar el acceso a través de los grupos y permisos de Azure Active Directory.

Docker Trusted Registry (DTR): Un servicio de registro ofrecido por la organización Docker, que se puede instalar en los servidores de la organización. Es conveniente para imágenes privadas que deban ser manejadas dentro de la empresa. El Docker Trusted Registry es parte del producto Docker Datacenter. Para más información consulte Docker Trusted Registry (DTR).

Docker Community Edition (CE): Es una herramienta de desarrollo para Windows y macOS, para preparar, ejecutar y probar contenedores localmente. Docker CE para Windows provee los entornos de desarrollo para contenedores de Linux y de Windows. El Docker *host* de Linux está basado en una máquina virtual Hyper-V. El *host* para contenedores Windows está basado directamente en Windows. Docker CE para Mac está basado en el Apple Hypervisor Framework y el xhyve hypervisor que provee una máquina virtual con el *host* Docker para Linux sobre Mac OS X. Docker CE para Windows y para Mac reemplaza al Docker Toolbox, que estaba basado en el VirtualBox de Oracle.

Docker Enterprise Edition (EE): Es una versión, a escala empresarial, de las herramientas Docker para desarrollo y producción en Linux y Windows.

Compose: Una herramienta de línea de comandos y un fichero en formato YAML con metadata, para definir y ejecutar aplicaciones de múltiples contenedores. Se define una aplicación basada en múltiples imágenes con uno o más ficheros .yml, que pueden reemplazar parámetros dependiendo del entorno. Después de crear las definiciones, se puede desplegar la aplicación multi-contenedores con un solo comando (**`docker-compose up`**) que crea un contenedor en el *host* Docker por cada imagen.

Cluster: Una colección de hosts Docker presentada como si fuera un *host* Docker virtual único, para que una aplicación pueda escalar a múltiples instancias de servicios, distribuidos en múltiples hosts dentro del *cluster*. Se puede crear un *cluster* de Docker con Docker Swarm, Mesosphere DC/OS,

Kubernetes y Azure Service Fabric. (Cuando se usa el Docker Swarm para gestionar un *cluster*, habitualmente se hace referencia a la colección como "swarm" en vez de "*cluster*".)

Orquestador (*Orchestrator*): Una herramienta que simplifica la gestión de *clusters* y *hosts* Docker. Los orquestadores permiten gestionar sus imágenes, contenedores y hosts a través de interfaces gráficas o de línea de comandos (CLI). Se pueden gestionar las redes de contenedores, las configuraciones, balanceo de carga, descubrimiento de servicios, alta disponibilidad, configuración de los *host* Docker y más. Un orquestador es responsable de ejecutar, distribuir, escalar y reparar las cargas de trabajo en una colección de nodos. Usualmente los orquestadores son los mismos productos que proveen la infraestructura para el *cluster*, como Mesosphere DC/OS, Kubernetes, Docker Swarm y Azure Service Fabric.

Contenedores Docker, imágenes y registros

Al usar Docker, el 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 junto con su configuración y sus dependencias.

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

Los desarrolladores deberían guardar las imágenes en un registro, que funciona como una librería y se necesita para poder desplegar a través de los orquestadores de producción. Docker mantiene un registro público a través del Docker Hub; otros proveedores también ofrecen registros para diferentes colecciones de imágenes. Por otro lado, las empresas también pueden tener registros internos (*on-premises*), para mantener sus propias imágenes Docker.

la figura 2-4 muestra cómo se relacionan las imágenes y registros de Docker con los otros componentes del sistema. También se muestran las ofertas de registros de algunos proveedores.

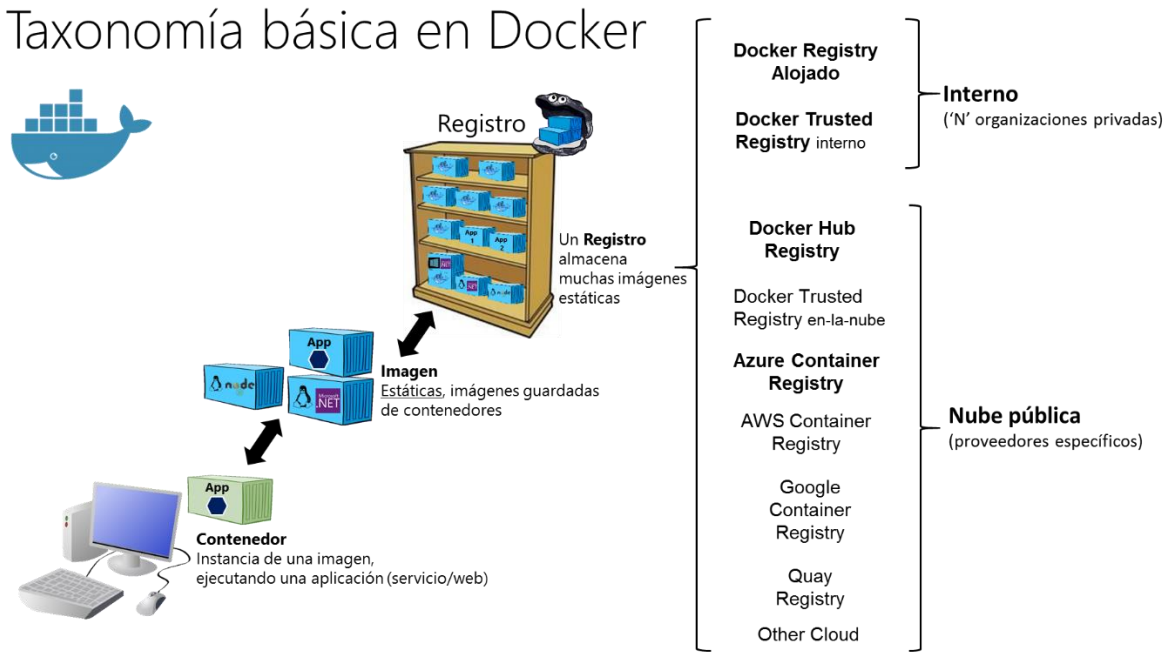


***Figura 2-4**. Taxonomía de los términos y conceptos de Docker*

Guardar las imágenes en un registro permite mantener todos los elementos de la aplicación, incluyendo sus dependencias hasta nivel del *framework*. Así, esas imágenes se pueden versionar y desplegar en múltiples entornos y, por lo tanto, ofrecer una unidad consistente de despliegue.

los registros privados de imágenes, tanto en servidores internos como en la nube se recomiendan cuando:

- Las imágenes no se deben compartir de forma pública por razones de confidencialidad.
- Necesita mantener una latencia mínima entre sus imágenes y el entorno de despliegue. Por ejemplo, si el entorno de producción está en Azure, probablemente lo mejor es guardar sus imágenes en el Azure Container Registry, para mantener al mínimo la latencia de la red. Sí el entorno de producción está en servidores internos, probablemente le interese mantener el Docker Trusted Registry disponible dentro de la misma red local.

Escogiendo entre .NET Core y .NET Framework para contenedores Docker

Para construir aplicaciones en contenedores Docker con .NET se puede utilizar tanto .NET Framework como .NET Core. Estos comparten muchos componentes de la plataforma .NET y, en general, se puede compartir mucho código entre ambos. Sin embargo, hay diferencias fundamentales entre ellos y la decisión dependerá de lo que se quiera lograr. En esta sección se ofrece una guía para decidir.

Orientación general

En esta sección se presenta un resumen para decidir cuándo seleccionar .NET Core y .NET Framework. Mostraremos más detalles en las secciones siguientes.

Debería utilizar .NET Core, con Linux o contenedores Windows, cuando:

- Necesita compatibilidad entre plataformas. Por ejemplo, cuando necesita usar tanto contenedores Linux como Windows.
- La arquitectura de la aplicación está basada en microservicios.
- Necesita arrancar contenedores de forma rápida y quiere mantener un tamaño pequeño por contenedor para conseguir mayor densidad, o más contenedores por servidor para disminuir sus costes.

En resumen, debería considerar .NET Core como la opción preferida para crear aplicaciones .NET en contenedores, ya que esto aporta muchos beneficios y encaja mejor en la filosofía y el estilo de trabajo con contenedores.

Un beneficio adicional de utilizar .NET Core, es que pueden ejecutar aplicaciones con versiones diferentes de .NET en la misma máquina. Aunque, definitivamente, esto es más importante para servidores o máquinas virtuales que no utilizan contenedores, ya que los contenedores permiten aislar las aplicaciones por completo. (Siempre que, por supuesto, las aplicaciones sean compatibles con el sistema operativo.)

Por otro lado, debería utilizar .NET Framework para sus aplicaciones en contenedores Docker cuando:

- Su aplicación utiliza actualmente .NET Framework y tiene fuertes dependencias con Windows.
- Necesita usar APIs de Windows que no son soportadas por .NET Core.

- Necesita utilizar librerías .NET o paquetes NuGet que no están disponibles para .NET Core.

Usar .NET Framework en Docker puede mejorar la experiencia de despliegue, al minimizar los problemas de asociados.

Esto permite manejar un escenario de migración rápida, "Lift and Shift", que es importante para contenerizar, aplicaciones antiguas (*legacy*) que fueron desarrolladas originalmente con el .NET Framework tradicional, por ejemplo, aplicaciones ASP.NET WebForms, Web MVC o con servicios WCF (Windows Communication Foundation).

Recursos adicionales

- **eBook: Modernize existing .NET Framework applications with Azure and Windows Containers**
  https://aka.ms/liftandshiftwithcontainersebook
- **Sample apps: Modernization of legacy .NET applications with Azure Cloud and Windows Containers**
  https://aka.ms/eshopmodernizing

Cuándo usar .NET Core para contenedores Docker

La naturaleza modular y liviana de .NET Core lo hace perfecto para trabajar en contenedores. Cuando se despliega y arranca un contenedor, su imagen es mucho menor con .NET Core que con .NET Framework. Porque para usar .NET Framework en un contenedor, se debe basar la imagen en el Windows Server Core, que es mucho más pesado que el Windows Nano Server o las imágenes de Linux que se utilizan para .NET Core.

Además, .NET Core es multiplataforma, así que puede desplegar sus aplicaciones tanto con imágenes Linux como Windows. Sin embargo, si está utilizando .NET Framework, sólo puede desplegar imágenes basadas en Windows.

A continuación, se presenta una explicación más detallada de por qué escoger .NET Core.

Desarrollo y despliegue para múltiples plataformas

Evidentemente, si su meta es tener una aplicación (web o servicio) que pueda correr en múltiples plataformas soportadas por Docker (Linux y Windows), la opción correcta es .NET Core, porque .NET Framework sólo soporta Windows.

.NET Core también soporta macOS como plataforma de desarrollo. Sin embargo, cuando se despliegan contenedores a un *host* Docker, ese *host* debe estar basado en Linux o Windows (al menos hasta comienzos de 2018).

Visual Studio ofrece un entorno integrado de desarrollo (IDE por sus siglas en inglés) para Windows con soporte para desarrollo en Docker.

Visual Studio para Mac es un IDE, la evolución de Xamarin Studio, que corre en macOS y también soporta el desarrollo de aplicaciones basadas en Docker. Este debería ser el entorno preferido para desarrollar en Mac ya que es muy poderoso.

También puede usar Visual Studio Code (VS Code) en macOS, Linux y Windows. VS Code soporta .NET Core por completo, incluyendo IntelliSense y depuración. Como VS Code es un editor ligero, se puede utilizar para desarrollar aplicaciones contenerizadas en la Mac, junto con la interfaz de línea de comandos de Docker (Docker CLI) y la interfaz de línea de comandos de .NET Core. También se puede desarrollar para .NET Core con la mayoría de los editores de terceros como Sublime, Emacs, vi y el proyecto open source OmniSharp, que también ofrece soporte para IntelliSense.

Además de los IDEs y editores mencionados, también puede usar las herramientas de la CLI de .NET Core para todas las plataformas soportadas.

Utilizando contenedores para proyectos nuevos ("green-field")

Los contenedores se utilizan frecuentemente con una arquitectura de microservicios, aunque también se puede utilizar para aplicaciones web o servicios que siguen cualquier patrón arquitectónico. Puede usar .NET Framework en contenedores Windows, pero la naturaleza modular y liviana de .NET Core lo hace perfecto para contenedores y arquitectura de microservicios. Cuando se crea y se despliega un contenedor con .NET Core, su imagen es mucho más pequeña que con .NET Framework.

Creando y desplegando microservicios en contenedores

Se podría utilizar el .NET Framework tradicional para desarrollar aplicaciones basadas en microservicios, corriendo como procesos simples en vez de contenedores. De esta forma, como el .NET Framework ya está instalado y compartido entre todos los procesos, éstos son livianos y arrancan rápido. Sin embargo, si está usando contenedores, como la imagen para el .NET Framework tradicional se basa en el Windows Server Core, es muy pesada para un enfoque basado en microservicios.

En cambio, .NET Core es la mejor opción si ha decidido trabajar orientado a microservicios basado en contenedores, porque .NET Core es muy liviano. Además, los contenedores correspondientes, tanto en Linux como en Windows Nano Server, son sencillos y livianos, haciendo que las imágenes sean livianas y arranquen rápido.

Un microservicio debe ser lo más pequeño posible, para ser ligero al arrancar, ocupar poco espacio, tener un *Bounded Context* pequeño (ver DDD, Domain-Driven Design), representar ámbito reducido de funciones y ser capaz de arrancar y terminar rápidamente. Para lograr esto es necesario utilizar imágenes pequeñas y rápidas de instanciar, como las que se consiguen con .NET Core.

Una arquitectura de microservicios también le permite combinar servicios con distintas tecnologías. Esto facilita la migración gradual a .NET Core para microservicios nuevos o crear otros, desarrollados con Node.js, Python, Java, Go u otras tecnologías.

Despliegues de alta densidad en sistemas escalables

Cuando se necesita la mejor densidad, granularidad y rendimiento posible, para una aplicación basada en contenedores, la mejor opción es usar .NET Core y ASP.NET Core. ASP.NET Core es hasta diez veces más rápido que ASP.NET en el .NET Framework tradicional y está de puntero, junto con otras tecnologías populares en microservicios, como Java Servlets, Go y Node.js.

Esto es especialmente importante para una arquitectura de microservicios, donde puede haber cientos de contenedores corriendo al mismo tiempo. Con las imágenes de ASP.NET Core (basadas en

.NET Core) en Linux o Windows Nano, puede ejecutar su sistema con una cantidad mucho menor de servidores o máquinas virtuales, ahorrando en costes de infraestructura y hospedaje.

Cuando usar .NET Framework para contenedores Docker

Aunque .NET Core ofrece beneficios importantes para nuevas aplicaciones y patrones, .NET Framework seguirá siendo una buena opción para muchos escenarios existentes

Migrando aplicaciones existentes directamente en un contenedor Windows Server

Podría ser interesante usar contenedores Docker simplemente para simplificar el despliegue, incluso si no está creando microservicios. Por ejemplo, quizás quiera mejorar su proceso DevOps con Docker (con los contenedores se puede conseguir un entorno aislado de pruebas y también eliminar problemas de despliegue causados por dependencias faltantes cuando se mueven al entorno de producción). En casos como este, incluso si está desplegando una aplicación monolítica, tiene sentido usar Docker y contenedores Windows para sus aplicaciones actuales en .NET Framework.

En la mayoría de los casos para este escenario, no necesitará migrar su aplicación a .NET Core, simplemente puede usar contenedores Docker que incluyan el .NET Framework tradicional. Sin embargo, se recomienda usar .NET Core en la medida que se vayan extender las aplicaciones existentes, por ejemplo, desarrollando un nuevo servicio en ASP.NET Core.

Usando librerías .NET de terceros o paquetes NuGet que no estén disponibles en .NET Core

Las librerías de terceros están adoptando rápidamente el .NET Standard, que permite compartir código entre las distintas variantes de .NET, incluyendo .NET Core. Con la versión 2.0 o superior de .NET Standard se ha ampliado significativamente el área de compatibilidad entre los distintos *frameworks* .NET y las aplicaciones .NET Core 2 pueden hacer referencia directa a los binarios de librerías existentes, gracias a un adaptador de compatibilidad (compat shim).

Además, recientemente se presentó el Windows Compatibility Pack, que amplía significativamente el conjunto de APIs disponibles y permite que el código existente se pueda compilar casi sin modificaciones, para correr en Windows.

Sin embargo, aún con el avance excepcional desde .NET Standard 2.0 y .NET Core 2.0, podría haber casos donde ciertos paquetes NuGet necesitan Windows para funcionar y no sean soportados en .NET Core. Si esos paquetes son críticos para su aplicación, entonces deberá usar el .NET Framework en contenedores Windows.

Usando tecnologías .NET no disponibles en .NET Core

Algunas tecnologías en .NET Framework no están disponibles en la versión actual de .NET Core (versión 2.0 al momento de este escribir esta guía). Algunas estarán disponibles en versiones posteriores de .NET Core, pero, en otros casos, cuando no se aplican los patrones hacia donde apunta .NET Core, puede que nunca estén disponibles.

A continuación, se muestra la lista de algunas tecnologías que no están disponibles en .NET Core 2:

- **ASP.NET WebForms**. Esta tecnología sólo está disponible en .NET framework. Actualmente no hay planes para migrar ASP.NET WebForms a .NET Core.
- **Servicios WCF**. Aunque existe una librería cliente-WCF disponible para consumir servicios WCF desde .NET Core, la implementación de servidor para WCF solo está disponible, al menos hasta noviembre de 2017, en .NET Framework. Esto podría cambiar para versiones futuras de .NET Core, incluso ya hay algunas APIs consideradas dentro del Windows Compatibility Pack.
- **Servicios relacionados con Workflow**. El Windows Workflow Foundation (WF), Workflow Services (WCF + WF en un solo servicio) y WCF Data Services (antes conocido como ADO.NET Data Services) sólo están disponibles en .NET Framework. Actualmente no hay planes para migrarlos a .NET Core.

Además de las tecnologías incluidas oficialmente en el .NET Core roadmap, puede que se incluyan otras características para ser migradas a .NET Core. Para ver la lista completa, busque los ítems marcados como port-to-core, en el repositorio GitHub de CoreFX. tenga en cuenta que está lista no representa un compromiso de Microsoft para migrar esos componentes a .NET Core, esa lista sólo registra las solicitudes de la comunidad. Si le interesa alguno de los componentes indicados, considere participar en las discusiones en GitHub, para poder aportar sus opiniones. Y si le parece que falta algo, por favor, registre una incidencia en el repositorio de CoreFX.

Usando una plataforma o API que no soporta .NET Core

Algunas plataformas de Microsoft o de terceros no soportan .NET Core. Por ejemplo, algunos servicios de Azure tienen un SDK que todavía no está disponible en .NET Core. Esto es una situación temporal, porque todos los servicios de Azure eventualmente utilizarán .NET Core. Por ejemplo, el Azure DocumentDB SDK for .NET Core fue lanzado como una versión preliminar el 16 de noviembre de 2016, pero desde hace tiempo está disponible (GA – General Avalability) como una versión estable.

Mientras tanto, si alguna plataforma o servicio en Azure aún no es compatible con .NET Core a través de su API de cliente, se puede usar el API REST equivalente desde el servicio Azure o el SDK de cliente en .NET Framework.

Recursos adicionales

- **Guía de .NET Core**
  https://docs.microsoft.com/dotnet/articles/core/index
- **Portabilidad a .NET Core desde .NET Framework**
  https://docs.microsoft.com/dotnet/articles/core/porting/index
- **Implementación de aplicaciones de .NET Framework con Docker**
  https://docs.microsoft.com/dotnet/articles/framework/docker/
- **Componentes de la arquitectura .NET**
  https://docs.microsoft.com/dotnet/articles/standard/components

Tabla de decisiones: Cuál versión de .NET usar con Docker

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

Arquitectura / Tipo de Aplicación	Contenedores Linux	Contenedores Windows
Microservicios en contenedores	.NET Core	.NET Core
Aplicaciones monolíticas	.NET Core	.NET Framework .NET Core
El mejor rendimiento y escalabilidad	.NET Core	.NET Core
Migración de aplicaciones *legacy* Windows Server a contenedores ("brown-field")	--	.NET Framework
Nuevos desarrollos basados en contenedores ("green-field")	.NET Core	.NET Core
ASP.NET Core	.NET Core	.NET Core (recomendado) .NET Framework
ASP.NET 4 (MVC 5, Web API 2, y Web Forms)	--	.NET Framework
Servicios SignalR	.NET Core 2.1 (cuando se libere) o posterior	.NET Framework .NET Core 2.1 (cuando se libere) o posterior
WCF, WF y otros *frameworks legacy*	WCF en .NET Core (sólo la librería cliente WCF)	.NET Framework WCF en .NET Core (sólo la librería cliente WCF)
Consumo de servicios de Azure	.NET Core (eventualmente todos los servicios Azure proveerán SDKs cliente para.NET Core)	.NET Framework .NET Core (eventualmente todos los servicios Azure proveerán SDKs cliente para.NET Core)

A qué sistema operativo apuntar con contenedores .NET

Dada la diversidad de sistemas operativos soportados por Docker y las diferencias entre .NET Framework y .NET Core, debe apuntar a un sistema operativo específico y versiones específicas según el framework que esté utilizando.

Para Windows, puede usar Windows Server Core o Windows Nano Server. Estas versiones de Windows proporcionan diferentes características (IIS en Windows Server Core frente a un servidor web *self-hosted* como Kestrel en Nano Server) que podrían ser necesarias para .NET Framework o .NET Core, respectivamente.

Para Linux, están disponibles múltiples distribuciones y están soportadas en imágenes oficiales de .NET en Docker (como Debian).

En la Figura 3-1, puede ver las versiones posibles del sistema operativo según la versión de .NET utilizada.

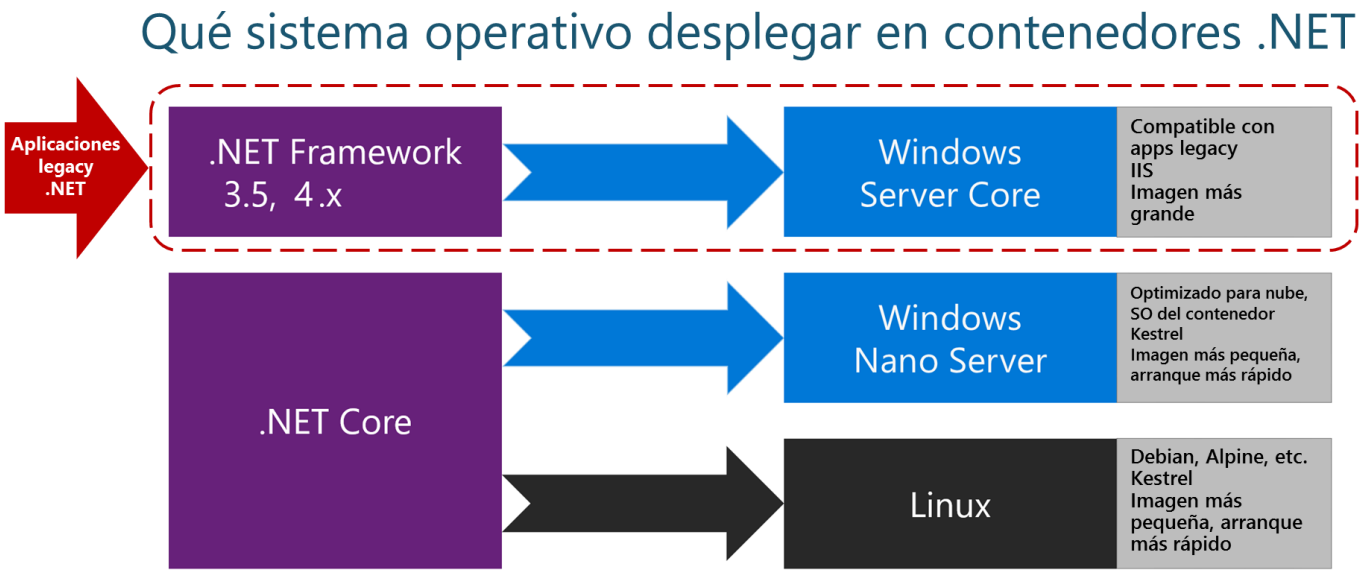


Figura 3-1. *Sistemas operativos a desplegar dependiendo de la version de .NET*

También puede crear su propia imagen de Docker en los casos en que desee utilizar una distribución de Linux diferente o cuando necesite 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 no es un escenario tan común para Docker.

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

microsoft/dotnet:**2.0.0-runtime-jessie**	.NET Core 2.0 runtime-only en Linux
microsoft/dotnet:**2.0.0-runtime-nanoserver-1709**	.NET Core 2.0 runtime-only en Windows Nano Server (Windows Server 2016 Fall Creators Update version 1709)
microsoft/aspnetcore:**2.0**	.NET Core 2.0 multi-arquitectura: Soporta Linux y Windows Nano Server dependiendo del host. La imagen aspnetcore tiene algunas optimizaciones para ASP.NET Core.

Imágenes oficiales Docker para.NET

Las imágenes oficiales Docker para .NET son creadas y optimizadas por Microsoft. Están disponibles públicamente en los repositorios de Microsoft en Docker Hub. Cada repositorio puede contener múltiples imágenes, dependiendo de las versiones de .NET y el sistema operativo y sus versiones (Linux Debian, Linux Alpine, Windows Nano Server, Windows Server Core, etc.).

La visión de Microsoft en este aspecto, es tener repositorios .NET granulares y enfocados, donde un repositorio representa un escenario o tipo de trabajo específico. Por ejemplo, las imágenes en microsoft/aspnetcore se deben usar para trabajar con ASP.NET Core en Docker, porque incluyen optimizaciones adicionales para iniciar los contenedores más rápidamente.

Por otro lado, las imágenes .NET Core en microsoft/dotnet están pensadas para aplicaciones de consola basadas en .NET Core. Por ejemplo, los procesos por lotes, Azure WebJobs y otros escenarios de consola deben usar .NET Core. Esas imágenes no incluyen el *framework* ASP.NET Core, lo que resulta en una imagen de contenedor más pequeña.

La mayoría de los repositorios de imágenes proporcionan un etiquetado amplio para ayudarle a seleccionar no sólo una versión específica de framework, sino también para elegir un sistema operativo (distribución de Linux o versión de Windows).

Para obtener más información acerca de las imágenes oficiales Docker para .NET proporcionadas por Microsoft, consulte el resumen de imágenes Docker para .NET.

Optimizaciones de imágenes .NET Core y Docker para desarrollo versus producción

Al construir imágenes Docker para desarrolladores, Microsoft se enfocó en los siguientes escenarios principales:

- Imágenes utilizadas para desarrollar y construir aplicaciones .NET Core.
- Imágenes utilizadas para ejecutar aplicaciones .NET Core.

¿Por qué múltiples imágenes? Al desarrollar, construir y ejecutar aplicaciones en contenedores, generalmente se tienen diferentes prioridades. Al proporcionar diferentes imágenes para estos

escenarios, Microsoft ayuda a optimizar los procesos de desarrollo, construcción y despliegue de aplicaciones.

Durante el desarrollo y la construcción

Durante el desarrollo, lo importante es qué tan rápido puede iterar al hacer los cambios y la capacidad de depurarlos. El tamaño de la imagen no es tan importante como la capacidad de realizar cambios en el código y verlos rápidamente. Algunas herramientas y "contenedores de *build-agents*" utilizan de la imagen de desarrollo de ASP.NET Core (microsoft/aspnetcore-build) durante el desarrollo y el proceso de construcción. Al construir dentro de un contenedor Docker, los aspectos importantes son los elementos que se necesitan para compilar la aplicación. Esto incluye el compilador y cualquier otra dependencia de .NET, además de dependencias de desarrollo web como npm, Gulp y Bower.

¿Por qué es importante este tipo de imagen de construcción? Porque esta imagen no se despliega en producción. En cambio, es una imagen que usas para construir el contenido que se coloca en una imagen de producción. Esta imagen se usaría en un entorno de integración continua (CI) o entorno de construcción. Por ejemplo, en lugar de instalar manualmente todas las dependencias de las aplicaciones directamente en un *host* con el agente de construcción (una máquina virtual, por ejemplo), el agente de construcción instanciaría una imagen de construcción .NET Core con todas las dependencias requeridas para construir la aplicación. El agente de construcción sólo necesita saber cómo ejecutar esta imagen Docker. Esto simplifica su entorno de CI y lo hace mucho más predecible.

En producción

En producción, lo importante es qué tan rápido se pueden desplegar e iniciar los contenedores en base a una imagen de producción .NET Core. Por lo tanto, esta imagen está basada en microsoft/aspnetcore y es pequeña, para que pueda viajar rápidamente a través de la red desde el registro Docker hasta los *host* Docker. Los contenidos están listos para ejecución, para lograr el menor tiempo posible desde el inicio del contenedor hasta obtener los resultados del procesamiento. En producción no hay necesidad de compilar el código C#, como ocurre cuando ejecuta **`dotnet build`** o **`dotnet publish`** al usar el contenedor de construcción.

En esta imagen optimizada, sólo se colocan los binarios y otro contenido necesario para ejecutar la aplicación. Por ejemplo, el contenido creado por **`dotnet publish`** sólo incluye los ficheros binarios .NET compilados, imágenes, ficheros .js y .css. Con el tiempo, incluso verá imágenes que contienen los paquetes *pre-jitted* (la compilación del IL al código nativo de la plataforma, que se hace en tiempo de ejecución al arrancar la aplicación).

Aunque hay múltiples versiones de las imágenes .NET Core y ASP.NET Core, todas comparten una o más capas, incluida una capa base. Por lo tanto, la cantidad de espacio en disco necesaria para almacenar una imagen es pequeña; sólo consiste en la variación (el delta) entre su imagen personalizada y su imagen base. El resultado es que las imágenes se extraen rápidamente del registro.

Al explorar los repositorios de imágenes .NET en Docker Hub, encontrará múltiples versiones de imágenes clasificadas o marcadas con etiquetas. Estas etiquetas ayudan a decidir cuál usar, dependiendo de la versión que necesite, como las de la siguiente tabla:

`microsoft/`**`aspnetcore:2.0`**	ASP.NET Core, sólo con runtime y optimizaciones para ASP.NET Core, en Linux y Windows (multi-arch)
`microsoft/`**`aspnetcore-build:2.0`**	ASP.NET Core, incluyendo los SDKs, en Linux y Windows (multi-arch)

Definiendo la arquitectura de aplicaciones basadas en Contenedores y Microservicios

Visión

Los microservicios ofrecen grandes beneficios, pero también plantean enormes desafíos. Los patrones de arquitectura de microservicios son los pilares fundamentales para crear aplicaciones basadas en microservicios.

Anteriormente vimos conceptos básicos sobre contenedores y Docker. Esa era la información mínima necesaria para comenzar. Sin embargo, aun cuando los contenedores son habilitadores y se adaptan perfectamente a los microservicios, no son obligatorios este tipo de arquitectura y muchos de los conceptos que veremos, también podrían aplicarse sin contenedores. Sin embargo, esta guía se centra en la intersección de ambos debido a la importancia de los contenedores.

Las aplicaciones empresariales pueden ser complejas y a menudo se componen de múltiples servicios en lugar de una sola aplicación basada en servicios. Para esos casos, se deben comprender los enfoques arquitectónicos adicionales, como los microservicios y ciertos patrones del diseño orientado por el dominio (DDD – Domain-Driven Design), más los conceptos de orquestación de contenedores. Tenga en cuenta que este capítulo describe no sólo microservicios en contenedores, sino también cualquier aplicación contenerizada.

Principios de diseño de los contenedores

En el modelo de contenedores, cada contenedor, o instancia de una imagen, representa un proceso único. Al definir una imagen como los límites de un proceso, se pueden crear primitivas que se pueden usar para escalar el proceso o ejecutarlo por lotes (en modo batch).

Cuando se diseña una imagen, verá la definición de un ENTRYPOINT en el **`Dockerfile`**. Esto define el proceso cuya vida determinará la vida útil del contenedor. Cuando el proceso finaliza, también termina la vida del contenedor. Los contenedores pueden representar procesos de larga ejecución como servidores web, pero también pueden representar procesos de corta duración como trabajos por lotes, que anteriormente se podrían haber 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 terminado. Más adelante profundizaremos sobre los orquestadores.

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

Contenerizando aplicaciones monolíticas

Es posible que desee construir una única aplicación web o servicio que se desplieguen monolíticamente y hacerlo como un contenedor. La aplicación en sí misma puede no ser monolítica internamente, sino estructurarse como varias librerías, componentes o incluso capas (capa de aplicación, capa de dominio, capa de acceso a datos, etc.). Externamente, sin embargo, es un contenedor único: un proceso único, una aplicación web o un servicio únicos.

Para gestionar esta arquitectura, se despliega un solo contenedor para representar la aplicación. Para aumentar la capacidad, simplemente se agregan más copias, en servidores o máquinas virtuales diferentes, con un balanceador de carga al frente. La simplicidad proviene de administrar un despliegue único, en un solo contenedor o máquina virtual.

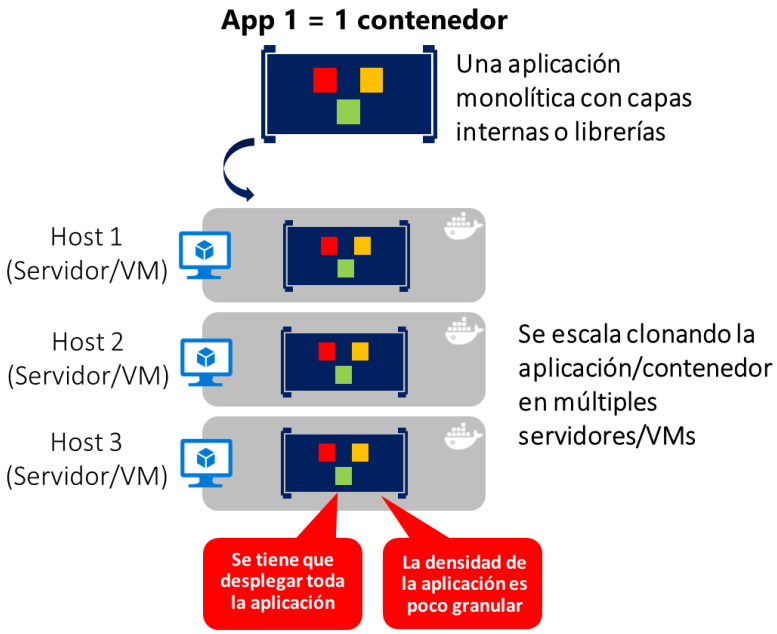


***Figura 4-1**. Ejemplo de la arquitectura de una aplicación monolítica contenerizada*

Puede incluir múltiples componentes, librerías o capas internas en cada contenedor, como se ilustra en la Figura 4-1. Sin embargo, aunque este patrón monolítico podría entrar en conflicto con el

principio del contenedor "un contenedor hace una cosa y lo hace en un proceso", podría estar justificado en algunos casos.

La desventaja de este enfoque resulta evidente si la aplicación crece y es necesario escalarla. Si toda la aplicación se puede escalar, no es realmente un problema. Sin embargo, en la mayoría de los casos, sólo algunas partes de la aplicación son los "cuellos de botella" que requieren escalado, mientras que el resto se usa menos.

Por ejemplo, en una aplicación típica de comercio electrónico, es probable que necesite escalar el subsistema de información de productos, porque hay muchos más clientes buscando productos que los que compran. Hay más clientes que usan su carrito de compras que los que usan el canal de pagos. Menos clientes agregan comentarios o ven su historial de compras. Y es posible que sólo 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 tareas diferentes se despliega varias veces y se escala en el mismo grado, lo que representa un uso poco eficiente de los recursos.

Hay muchas formas de escalar una aplicación: duplicación horizontal, división de diferentes áreas de la aplicación y creación de particiones por conceptos similares o data del negocio. 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 un despliegue completo de todas las instancias.

Sin embargo, el enfoque monolítico es de uso frecuente, porque el desarrollo de la aplicación es inicialmente más fácil que para los enfoques de microservicios. Por lo tanto, muchas organizaciones desarrollan usando este enfoque arquitectónico. Si bien algunas organizaciones han tenido buenos resultados, otras están alcanzando los límites. Muchas organizaciones diseñaron sus aplicaciones utilizando este modelo porque las herramientas y la infraestructura hicieron que fuera muy difícil construir 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 y tener una relación aceptable de eficiencia en el uso de recursos, como se muestra en la Figura 4-2.

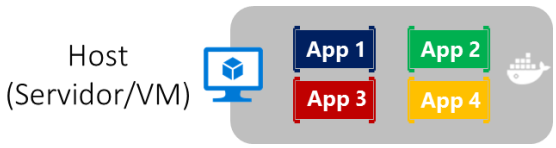


***Figura 4-2**. Enfoque monolítico: Host ejecutando multiples aplicaciones, cada una como un contenedor*

Las aplicaciones monolíticas en Microsoft Azure se pueden implementar utilizando máquinas virtuales dedicadas para cada instancia. Además, al usar los conjuntos de escalado de Azure, se pueden escalar fácilmente las máquinas virtuales. También se puede usar un Azure App Service para aplicaciones monolíticas y escalar instancias fácilmente sin necesidad de administrar las máquinas virtuales. Desde 2016, Azure App Services también puede ejecutar instancias individuales de contenedores Docker, lo que simplifica el despliegue.

Como entorno de control de calidad o entorno limitado de producción, puede desplegar múltiples máquinas virtuales como *host* Docker y balancearlas usando el balanceador de carga de Azure, como se muestra en la Figura 4-3. Esto le permite gestionar el escalado, aunque sólo de forma gruesa, porque toda la aplicación vive en un solo contenedor.

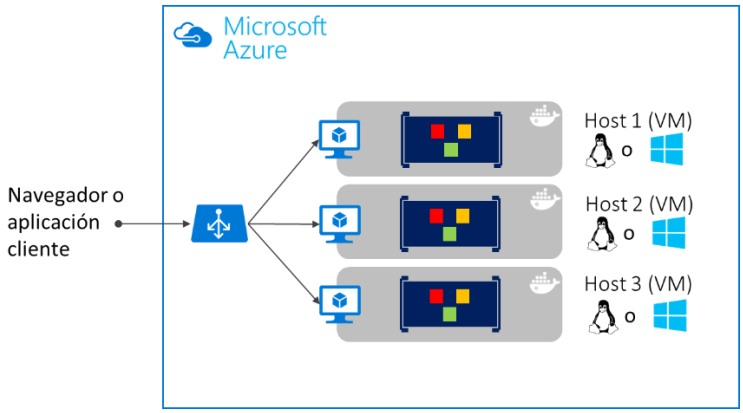


***Figura 4-3**. Ejemplo de múltiples hosts escalando una aplicación de contenedor único*

El despliegue en los distintos hosts se puede gestionar con técnicas de despliegue tradicionales. Los hosts de Docker se pueden administrar con comandos como **`docker run`** o **`docker-compose`** realizados manualmente, o mediante la automatización, como los procesos de entrega continua (CD – *Continuous Delivery*).

Desplegando una aplicación monolítica en un contenedor

El uso de contenedores para gestionar el despliegue de aplicaciones monolíticas tiene sus ventajas. Escalar instancias de contenedor es mucho más rápido y más fácil que desplegar máquinas virtuales adicionales. Incluso si usa los conjuntos de escalado, las máquinas virtuales requieren tiempo para arrancar. Cuando se despliegan como instancias de aplicaciones tradicionales en lugar de contenedores, la configuración de la aplicación se administra como parte de la máquina virtual, lo que no es ideal.

El despliegue de actualizaciones como imágenes Docker es mucho más rápido y eficiente en el uso de la red. Las imágenes de Docker generalmente comienzan en segundos, lo que acelera el proceso. Destruir una instancia de imagen de Docker es tan fácil como emitir un comando **`docker stop`** y generalmente se completa en menos de un segundo.

Como los contenedores son inmutables por diseño, nunca tendrá que preocuparse por máquinas virtuales corruptas. Por el contrario, una máquina virtual tiene muchas partes móviles que pueden verse afectadas por factores externos no controlados.

Si bien las aplicaciones monolíticas pueden beneficiarse de Docker, apenas estamos echándole un vistazo a los beneficios. Las ventajas adicionales de la gestión de contenedores provienen del despliegue con orquestadores, que administran las diversas instancias y el ciclo de vida de cada contenedor. Descomponer una aplicación monolítica en subsistemas que puedan escalarse, desarrollarse y desplegarse individualmente es el punto de entrada al mundo de los microservicios.

Publicando una aplicación basada en un contenedor único a un Azure App Service

Si quiere evaluar un contenedor desplegado en Azure o cuando la aplicación requiere sólo un contenedor, el Azure App Service proporciona una manera excelente de ofrecer servicios escalables basados en un solo contenedor. Es sencillo usar un Azure App Service. Proporciona una buen

integración con Git para que sea más fácil tomar su código, compilarlo en Visual Studio y desplegarlo directamente en Azure.

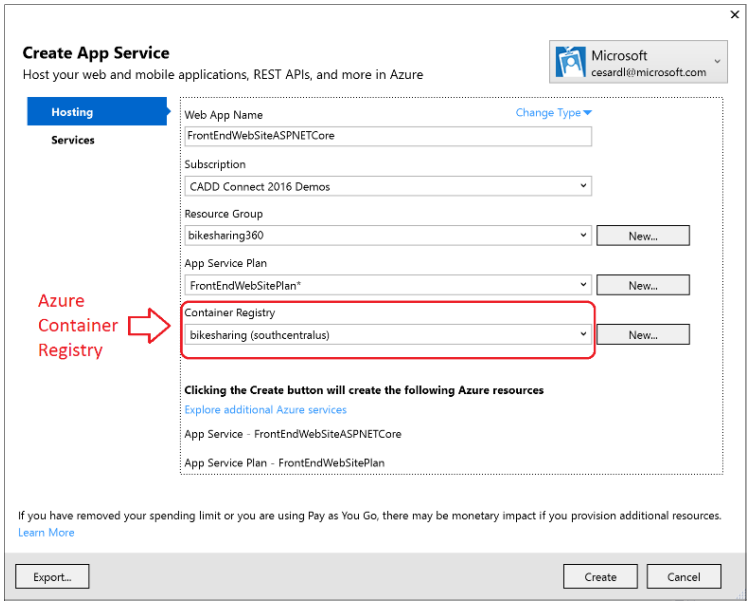


***Figura 4-4**. Publicando una aplicación de contenedor único a un Azure App Service desde Visual Studio*

Sin Docker, si necesitara otras facilidades, *frameworks* o dependencias que no son compatibles con Azure App Service, tendría que esperar hasta que el equipo de Azure actualizara esas dependencias en el App Service. O tendría que cambiar a otros servicios como Azure Service Fabric, Azure Cloud Services o incluso máquinas virtuales, donde tiene más control y puede instalar un componente o un framework necesario para su aplicación.

El soporte de contenedores en Visual Studio 2017 le da la posibilidad de incluir lo que desee en el entorno de la aplicación, como se muestra en la Figura 4-4. Como lo está ejecutando en un contenedor, si agrega una dependencia a su aplicación, puede incluir la dependencia en su **`Dockerfile`** o en la imagen Docker.

Como también puede ver en la Figura 4-4, el flujo de publicación canaliza la imagen a través de un registro de contenedores. Puede ser Azure Container Registry (un registro cercano a sus despliegues en Azure y asegurado por grupos y cuentas de Azure Active Directory) o cualquier otro registro de Docker, como Docker Hub o un registro local.

La data y el estado en las aplicaciones 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 se mantendrá indefinidamente sería como suponer que una única ubicación en la memoria será duradera. Se debe entender que las imágenes, como los procesos, pueden tener múltiples instancias que eventualmente serán eliminadas. Si se administran con un orquestador, también debe suponer que se pueden mover de un nodo o máquina virtual a otra.

Se pueden usar las siguientes soluciones para gestionar datos persistentes en aplicaciones Docker:

Desde el *host* Docker, como Docker Volumes:

- **Volumes**, que se guardan en un área del sistema de ficheros del *host* que es manejada por Docker.
- **Bind mounts**, que permiten mapear cualquier carpeta del sistema de ficheros del *host*, pero Docker no puede controlar el acceso y eso puede representar un riesgo alto de seguridad, ya que un contenedor podría acceder a carpetas sensibles del sistema operativo.
- **tmpfs mounts**, que son carpetas virtuales que existen sólo en la memoria del *host* y no se escriben nunca al sistema de ficheros.

Desde almacenamiento remoto:

- Azure Storage, que proporciona almacenamiento geo-distribuido, que ofrece una Buena solución de persistencia a largo plazo para contenedores.
- Bases de datos relacionales remotas, como Azure SQL Database or bases de datos NoSQL como Azure Cosmos DB, servicios de caché como Redis.

Desde el contenedor Docker:

> Docker proporciona una función denominada *overlay file system*. Esto implementa una tarea *copy-on-write* que almacena información actualizada en el sistema de ficheros raíz del contenedor. Esta información se añade a la imagen original en la que se basa el contenedor. Si se borra el contenedor del sistema, se pierden esas modificaciones. Por lo tanto, si bien es posible salvar el estado de un contenedor dentro de su almacenamiento local, el diseño de un sistema en torno a esto entraría en conflicto con la premisa del diseño de contenedor, que por defecto es *stateless*.
>
> Sin embargo, los volúmenes Docker presentados anteriormente son ahora la forma preferida de manejar los datos locales en Docker. Si necesita más información sobre el almacenamiento en contenedores, consulte en Docker storage drivers y About images, containers and storage drivers.

A continuación, se proporcionan más detalles sobre estas opciones.

Los ***volumes*** son directorios mapeados desde el sistema operativo host a directorios en los contenedores. Cuando el código en el contenedor accede al directorio mapeado, está accediendo realmente a un directorio en el sistema operativo del host. Este directorio no está vinculado a la vida útil del contenedor en sí mismo y el directorio es administrado por Docker y esta aislado de la funcionalidad básica del host. Por lo tanto, los volúmenes de datos están diseñados para que persistan independientemente de la vida útil del contenedor. Si elimina un contenedor o una imagen del host Docker, los datos que persisten en el volumen no se borran.

Los volúmenes pueden ser nombrados o anónimos (por defecto). Los volúmenes nombrados son la evolución de los Data Volume Containers y facilitan el intercambio de datos entre contenedores. Los volúmenes también soportan volume drivers, que le permiten almacenar datos en hosts remotos, entre otras opciones.

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

Los ***tmpfs mounts*** son básicamente carpetas virtuales que viven sólo en la memoria del *host* y nunca se escriben en el sistema de archivos. Son rápidos y seguros, pero usan memoria y obviamente sólo están pensados para datos no persistentes.

Como se muestra en la Figura 4-5, los volúmenes regulares de Docker se almacenan fuera de los propios contenedores, pero dentro de los límites físicos del *host*. Sin embargo, los contenedores Docker no pueden acceder a un volumen desde un *host* a otro. En otras palabras, con estos volúmenes no es posible gestionar los datos compartidos entre contenedores que se ejecutan en diferentes hosts Docker, aunque podría lograrse con un *volume driver* compatible con *hosts* remotos.

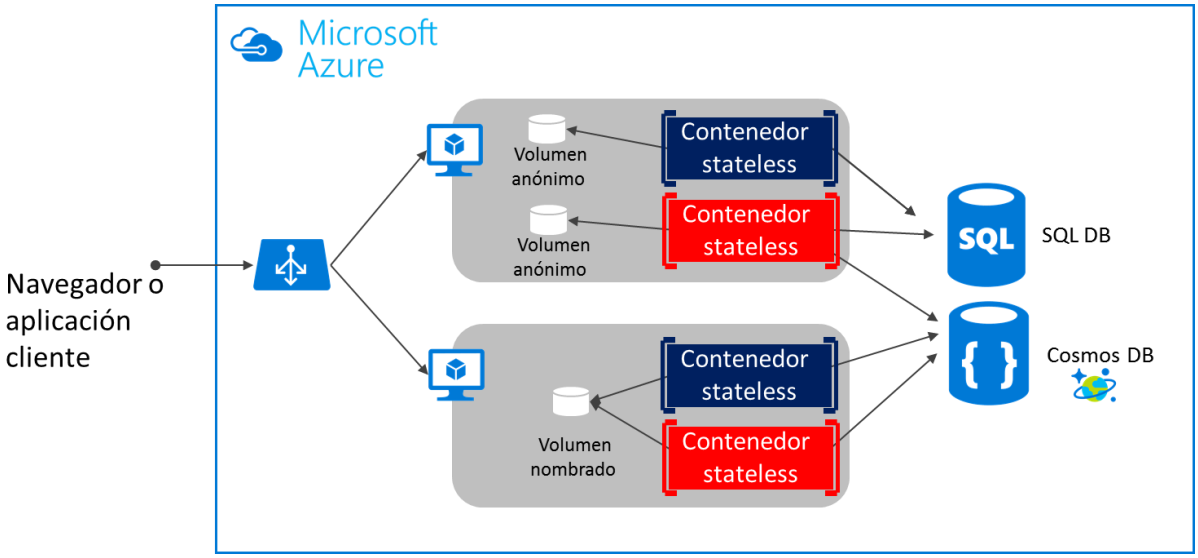


Figura 4-5. *Volúmenes Dpcker y fuentes de datos externos para aplicaciones en contenedores*

Además, cuando los contenedores Docker son manejados por un orquestador, los contenedores se pueden "mover" entre hosts, dependiendo de las optimizaciones realizadas por el cluster. Por lo tanto, no se recomienda utilizar volúmenes para datos del negocio. Pero son un buen mecanismo para trabajar con ficheros de trazas, temporales o similares que no afectarán la consistencia de los datos del negocio.

Las fuentes de datos remotas como Azure SQL Database, Azure Cosmos DB o un caché remoto como Redis, se pueden usar en aplicaciones en contenedores, de la misma forma como cuando se desarrollan sin contenedores. Esta es una forma comprobada de almacenar datos de aplicaciones de negocio.

Azure Storage. Por lo general, los datos del negocio deberán ubicarse en recursos externos o bases de datos, como Azure Storage. Azure Storage, en concreto, proporciona los siguientes servicios en la nube:

- El almacenamiento de blobs (*Blob Storage*) es adecuado para datos de objetos no estructurados. Un blob puede ser cualquier tipo de texto o datos binarios, como documentos o ficheros multimedia (imágenes, audio y ficheros de video). El almacenamiento de blobs también se conoce como almacenamiento de objetos.
- El almacenamiento de ficheros (*File Storage*) ofrece almacenamiento compartido para aplicaciones *legacy*, usando el protocolo SMB estándar. Las máquinas virtuales Azure y los servicios en la nube pueden compartir datos de ficheros entre los componentes de la

aplicación a través de recursos compartidos montados. Las aplicaciones locales pueden acceder a los datos del fichero en un recurso compartido a través de la API REST del servicio de ficheros.

- El almacenamiento de tablas (*Table Storage*) almacena conjuntos de datos estructurados. El almacenamiento de tablas es un almacén de datos clave-valor NoSQL, que permite un desarrollo rápido y un acceso rápido a grandes cantidades de datos.

Bases de datos relacionales y 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 van a ser explicadas como parte de esta guía ya que son un tema completamente diferente.

Arquitectura orientada a servicios

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

Esos servicios ahora se pueden implementar como contenedores Docker, lo que resuelve problemas de despliegue, porque todas las dependencias están incluidas en la imagen del contenedor. Sin embargo, cuando necesite escalar aplicaciones SOA, puede tener problemas de escalabilidad y disponibilidad si está desplegando en base a hosts Docker únicos. Aquí es donde el software de gestión de *cluster* Docker o un orquestador le ayudarán, tal como se explica en secciones posteriores donde describimos enfoques de despliegue para microservicios.

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

Los microservicios se derivan de SOA, pero SOA es diferente de la arquitectura de microservicios. Las características como grandes agentes centrales (*central brokers*), orquestadores centrales a nivel de organización y el Enterprise Service Bus (ESB) son típicos en SOA. Pero en la mayoría de los casos, estos son anti patrones en la comunidad de microservicios. De hecho, algunas personas argumentan que "la arquitectura de microservicio es SOA implementada correctamente".

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

Arquitectura de microservicios

Una arquitectura de microservicios es un enfoque para construir una aplicación de servidor como un conjunto de pequeños servicios. Es decir, una arquitectura de microservicios está orientada fundamentalmente hacia el *back-end*, aunque también se está usando el enfoque para el *front-end*. Cada servicio se ejecuta en su propio proceso y se comunica con otros procesos utilizando protocolos como HTTP/HTTPS, WebSockets o AMQP. Cada microservicio implementa una funcionalidad específica, de principio o fin, del negocio o del dominio, dentro de los límites de cierto contexto. Además, cada uno se debe poder desarrollar y desplegar de forma independiente. Finalmente, cada microservicio debe poseer su modelo de datos y su lógica de dominio relacionados (soberanía y gestión descentralizada de datos) que pueden estar basados en diferentes tecnologías de almacenamiento de datos (SQL, NoSQL) y diferentes lenguajes de programación.

¿Qué tamaño debería tener un microservicio? Al desarrollar un microservicio, el tamaño no debe ser el punto importante. En cambio, lo importante debería ser crear servicios ligeramente acoplados para tener autonomía en el desarrollo, despliegue y escalamiento en cada servicio. Por supuesto, al identificar y diseñar microservicios, se deben hacer lo más pequeños posible, siempre que no tenga demasiadas dependencias directas con otros microservicios. Más importante que el tamaño del microservicio es la cohesión interna que debe tener y su independencia de otros servicios.

¿Por qué una arquitectura de microservicios? En resumen, porque proporciona agilidad a largo plazo. Los microservicios facilitan el mantenimiento en sistemas complejos, grandes y que requieren alta escalabilidad, porque permiten crear aplicaciones basadas en muchos servicios, que se despliegan independientemente y que tienen ciclos de vida granulares y autónomos.

Como beneficio adicional, los microservicios se pueden escalar de forma independiente. En lugar de tener una aplicación única monolítica, que se debe escalar como una unidad, se pueden escalar individualmente microservicios específicos. De esta forma, puede escalar sólo el área funcional que necesita más potencia de procesamiento o ancho de banda de red para soportar la demanda, en lugar de ampliar otras áreas de la aplicación que no necesitan escalarse. Eso significa ahorros de costes porque se aprovechan mejor los recursos y se necesita menos hardware.

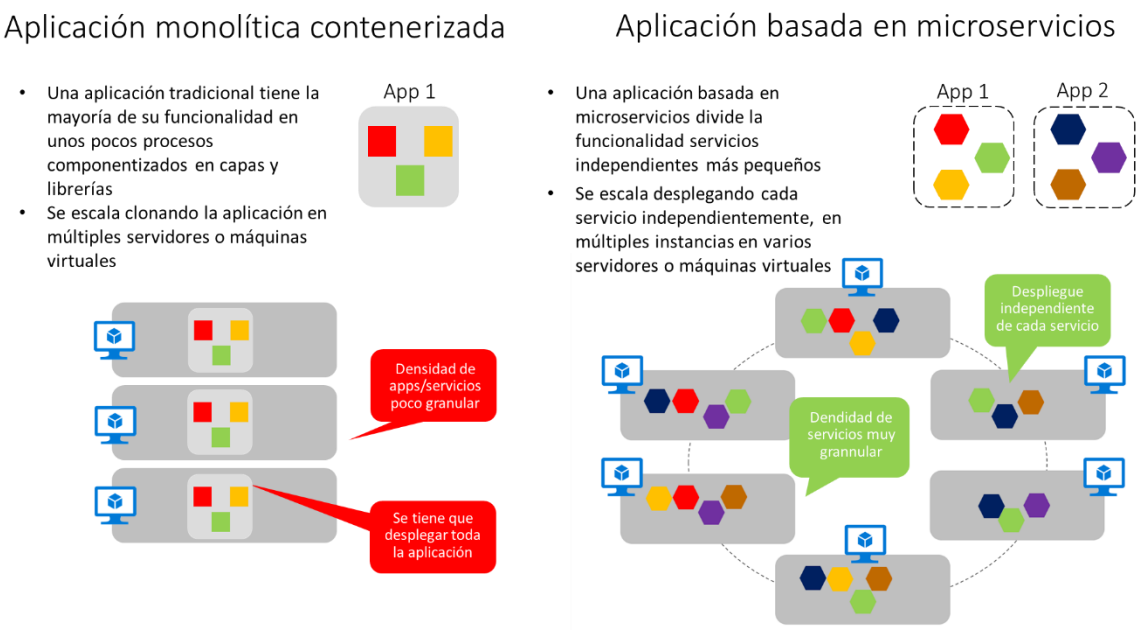


***Figura 4-6**. Despliegue monolítico versus el enfoque de microservicios*

Como muestra la Figura 4-6, el enfoque de microservicios permite agilizar los cambios e iterar rápidamente en cada microservicio, ya que es más fácil cambiar áreas pequeñas y específicas de aplicaciones que, en conjunto, son grandes, complejas y escalables.

Una arquitectura basada en microservicios granulares, es un facilitador para las prácticas de integración y entrega continuas (CI/CD). También ayuda a entregar más rápido las nuevas funciones en la aplicación, ya que la composición fina de las funciones le permite ejecutar y probar microservicios en forma aislada y desarrollarlos de manera autónoma, manteniendo claros los contratos entre ellos. Siempre que no cambie las interfaces o los contratos, puede cambiar la implementación interna de cualquier microservicio o agregar nuevas funcionalidades sin romper otros microservicios.

Es importante tener en cuenta los siguientes aspectos, para tener éxito al entrar en producción con un sistema basado en microservicios:

- Monitorización y controles del estado de salud de los servicios y la infraestructura.
- Infraestructura escalable para los servicios (es decir, nube y orquestadores).
- Diseño e implementación de seguridad en múltiples niveles: autenticación, autorización, gestión de secretos, comunicación segura, etc.
- Entrega rápida de aplicaciones, generalmente con diferentes equipos enfocados en diferentes microservicios.
- Prácticas e infraestructura para DevOps, integración continua (CI) y entrega continua (CD).

De estos, sólo se cubren o se presentan los tres primeros en esta guía. Los dos últimos puntos, que están relacionados con el ciclo de vida de las aplicaciones, se tratan en la guía Containerized Docker Application Lifecycle with Microsoft Platform and Tools.

Recursos adicionales

- **Mark Russinovich. Microservices: An application revolution powered by the cloud**
  https://azure.microsoft.com/blog/microservices-an-application-revolution-powered-by-the-cloud/
- **Martin Fowler. Microservices**
  http://www.martinfowler.com/articles/microservices.html
- **Martin Fowler. Microservice Prerequisites**
  http://martinfowler.com/bliki/MicroservicePrerequisites.html
- **Jimmy Nilsson. Chunk Cloud Computing**
  https://www.infoq.com/articles/CCC-Jimmy-Nilsson
- **Cesar de la Torre. Containerized Docker Application Lifecycle with Microsoft Platform and Tools** (downloadable eBook)
  https://aka.ms/dockerlifecycleebook

Soberanía de datos por microservicio

Una regla importante en la arquitectura de microservicios es que cada microservicio debe poseer sus datos y lógica de dominio. Del mismo modo que una aplicación completa posee su lógica y datos, también cada microservicio debe tener su lógica y datos bajo un ciclo de vida autónomo, con despliegue independiente de otros.

Esto significa que el modelo conceptual del dominio puede diferir entre subsistemas o microservicios. Por ejemplo, las aplicaciones empresariales, donde las aplicaciones de gestión de relaciones con los clientes (CRM – *Customer Relationship Management*), los subsistemas de compras transaccionales y los subsistemas de atención al cliente, utilizan cada uno atributos y datos particulares del cliente, pero cada uno ve al cliente de una forma particular y emplea un *Bounded Context* diferente.

Este principio es similar en el diseño orientado por el dominio (DDD – Domain-Driven Design), donde cada *Bounded Context* o subsistema autónomo o servicio debe poseer su modelo de dominio (datos más lógica y comportamiento). Cada *Bounded Context* por DDD se correlaciona con un microservicio empresarial (uno o varios servicios). (Ampliamos este punto sobre el patrón de *Bounded Context* en la siguiente sección).

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

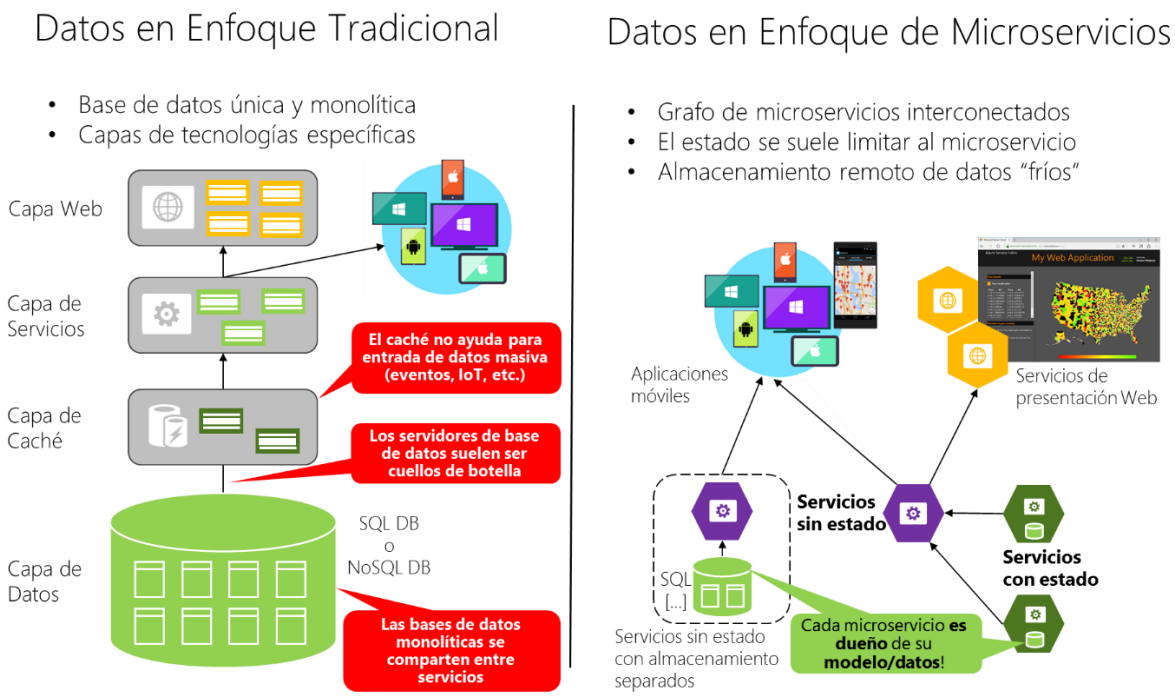


***Figure 4-7**. Comparación de la soberanía de datos: base de datos monolítica versus microservicios*

El enfoque de la base de datos centralizada parece más simple inicialmente y también parece permitir la reutilización de entidades en diferentes subsistemas para que todo sea coherente. Pero la realidad es que se suele terminar 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 para hacer senderismo, un viaje largo en automóvil y aprender geografía.

Una aplicación monolítica con una sola base de datos relacional tiene dos ventajas 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 hacer fácilmente una consulta que combina datos de múltiples tablas.

Sin embargo, el acceso a los datos se vuelve mucho más complejo cuando se trabaja en una arquitectura de microservicios. Pero, de todas formas, se deben usar transacciones ACID dentro de un microservicio o *Bounded Context* y considerar que los datos son privados para cada microservicio y sólo se puede acceder a ellos a través de su API. El encapsulamiento de los datos garantiza que los microservicios se acoplen de manera flexible y puedan evolucionar independientemente el uno del otro. Si varios servicios accedieran a los mismos datos, las actualizaciones del modelo de datos requerirían actualizaciones coordinadas de todos los servicios. Esto rompería la autonomía del ciclo de vida del microservicio. Pero las estructuras de datos distribuidas significan que no se puede realizar una transacción ACID única en microservicios. Esto a su vez significa que debe usar consistencia eventual cuando un proceso del negocio abarca múltiples microservicios. Esto es mucho más difícil de implementar que hacer simples "joins" en SQL, porque no se pueden establecer restricciones de integridad referencial ni usar transacciones distribuidas entre bases de datos diferentes, como veremos más adelante. De forma similar, muchas otras características de bases de datos relacionales no están disponibles cuando se trabaja con múltiples microservicios.

Yendo aún más lejos, los microservicios diferentes podrían usar 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, una base de datos NoSQL como Azure DocumentDB o MongoDB podría ser un modelo más conveniente y ofrecer un mejor rendimiento y escalabilidad que una base de datos SQL como SQL Server o Azure SQL Database. En otros casos, una base de datos relacional podría seguir siendo la mejor opción. Por lo tanto, las aplicaciones basadas en microservicios a menudo usan una mezcla de bases de datos SQL y NoSQL, que a veces se denomina enfoque de persistencia políglota.

Una arquitectura particionada y de persistencia políglota para los datos tiene muchos beneficios. Estos incluyen servicios poco acoplados y un mejor rendimiento, escalabilidad, costes y capacidad de administración. Sin embargo, puede presentar algunos desafíos de administración de datos distribuidos, como explicaremos 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 orientado por el dominio (DDD). DDD maneja modelos grandes dividiéndolos en múltiples BC y siendo explícito sobre sus límites. Cada BC debe tener su propio modelo y base de datos, es decir, cada microservicio es el dueño de sus datos. Además, cada BC generalmente tiene sus propios términos, que establecen el lenguaje ubicuo que facilita la comunicación entre los desarrolladores y los expertos del dominio.

Esos términos (principalmente entidades de dominio) en el lenguaje ubicuo pueden tener diferentes significados en diferentes *Bounded Contexts*, incluso cuando diferentes entidades de dominio comparten la misma identidad (es decir, la ID única que se utiliza para leer la entidad del

almacenamiento). Por ejemplo, en un BC de perfiles de usuario, la entidad "Usuario" puede compartir el identificador con la entidad "Comprador" en el *Bounded Context* de pedidos.

Por lo tanto, un microservicio es como un *Bounded Context*, pero también se trata de un servicio distribuido. Se construye como un proceso separado para cada BC y debe usar los protocolos distribuidos mencionados anteriormente, como HTTP/HTTPS, WebSockets o AMQP. El patrón de *Bounded Context*, sin embargo, no especifica si es un servicio distribuido o simplemente es un límite lógico (como un subsistema genérico) dentro de una aplicación que se despliega monolíticamente.

Es importante destacar que la definición de un servicio por cada BC es un buen punto de inicio. Pero el diseño no tiene por qué restringirse a eso. En ocasiones, es necesario diseñar un BC o un microservicio comercial componiendo varios servicios físicos. Pero, en última instancia, ambos patrones, el *Bounded Context* y el microservicio, están estrechamente relacionados.

DDD se beneficia de los microservicios al obtener límites reales en forma de microservicios distribuidos. Pero las ideas como mantener modelos privados a los microservicios y no compartirlos, es lo que también se busca en un *Bounded Context*.

Recursos adicionales

- **Chris Richardson. Pattern: Database per service**
  http://microservices.io/patterns/data/database-per-service.html
- **Martin Fowler. BoundedContext**
  http://martinfowler.com/bliki/BoundedContext.html
- **Martin Fowler. PolyglotPersistence**
  http://martinfowler.com/bliki/PolyglotPersistence.html
- **Alberto Brandolini. Strategic Domain-Driven Design with Context Mapping**
  https://www.infoq.com/articles/ddd-contextmapping

Arquitectura lógica versus arquitectura física

Es bueno detenerse en este punto y analizar la diferencia entre arquitectura lógica y arquitectura física y cómo se aplica esto al diseño de aplicaciones basadas en microservicios.

Para comenzar, el desarrollo de microservicios no requiere ninguna tecnología específica. Por ejemplo, los contenedores Docker no son obligatorios para crear una arquitectura basada en microservicios. Esos microservicios también podrían ejecutarse como procesos simples. Es decir, microservicios es una arquitectura lógica.

Además, incluso cuando se podría implementar un microservicio como un servicio único, proceso o contenedor (por simplicidad, ese es el enfoque adoptado en la versión inicial de eShopOnContainers), esta paridad entre el microservicio del negocio y el servicio o contenedor no es necesariamente requerida en todos los casos, cuando se desarrolla una aplicación grande y compleja compuesta de muchas docenas o incluso cientos de servicios.

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

Aunque es posible que se hayan identificado algunos microservicios del negocio o *Bounded Context*s, esto no significa que la mejor forma de implementarlos sea siempre mediante la creación de un servicio único (como una API web de ASP.NET) o un contenedor Docker único para cada microservicio

del negocio. Resulta demasiado rígido tener una regla que diga que cada microservicio debe implementarse como un servicio o contenedor único.

Por lo tanto, un microservicio del negocio o un *Bounded Context* se refieren a una arquitectura lógica, que puede coincidir, o no, con la arquitectura física. El punto importante es que un microservicio del negocio o un BC debe ser autónomo y permitir que el código y el estado se actualicen, desplieguen y escalen independientemente.

Como se muestra en la Figura 4-8, el microservicio del catálogo podría estar compuesto por varios servicios o procesos. Estos podrían ser múltiples servicios API Web de ASP.NET o cualquier otro tipo de servicios utilizando 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 del negocio.

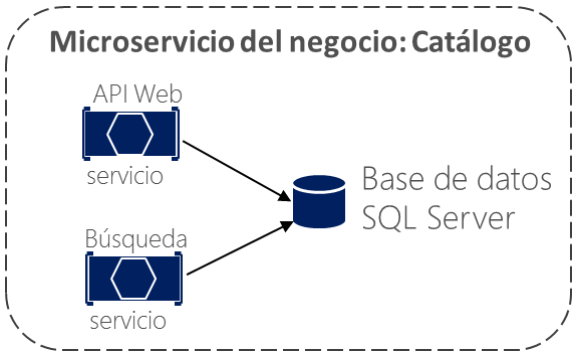


***Figura 4-8**. Microservicio del negocio con varios servicios físicos*

Los servicios del ejemplo comparten el mismo modelo de datos porque el servicio API Web tiene los mismos datos que el servicio de búsqueda. Entonces, en la implementación física del microservicio, se está dividiendo esa funcionalidad, para poder escalar cada uno de esos servicios hacia arriba o hacia abajo según sea necesario. Tal vez el servicio API Web generalmente necesite más instancias que el servicio de búsqueda, o viceversa.

En resumen, la arquitectura lógica de microservicios no siempre tiene que coincidir con la arquitectura física de despliegue. En esta guía, cada vez que mencionamos un microservicio, nos referimos a un microservicio del negocio o lógico, que se puede asignar a uno o más servicios (físicos). En la mayoría de los casos, este será un servicio único, pero podrían ser más.

Retos y soluciones para la gestión de datos distribuidos

Reto #1: Cómo definir los límites de cada microservicio

Definir los límites de un microservicio es probablemente el primer desafío que encontramos. Cada microservicio tiene que ser parte de su aplicación y cada microservicio debe ser autónomo con todos los beneficios y retos que eso implica. ¿Pero cómo se identifican esos límites?

En primer lugar, debemos enfocarnos en el modelo lógico de dominio de la aplicación y los datos relacionados. Se debe intentar identificar las islas de datos desacopladas y los diferentes contextos dentro de la aplicación. Cada contexto podría tener un lenguaje de negocio diferente (diferentes

términos del negocio). Los contextos de deberían definir y administrar de forma independiente. Los términos y entidades utilizados en esos contextos podrían sonar similares, pero también podría descubrir que, en un contexto particular, un concepto de negocio se usa para un propósito diferente en otro contexto e, incluso, el mismo concepto podría tener nombres diferentes. Por ejemplo, un "Usuario" puede referirse a un usuario en el contexto de identidad o membresía, a un cliente en el contexto de CRM, a un comprador en el contexto de pedidos y así sucesivamente.

La forma para identificar los límites de múltiples contextos, con un dominio diferente para cada uno, es exactamente la forma como puede identificar los límites para cada microservicio del negocio y su modelo de dominio y datos relacionados. Siempre se intenta minimizar el acoplamiento entre esos microservicios. Más adelante en esta guía, entraremos en los detalles sobre esta identificación y diseño del modelo de dominio, en la sección Identificación de los límites del modelo de dominio para cada microservicio.

Reto #2: Cómo crear consultas que recuperan datos de varios microservicios

El segundo desafío es cómo implementar consultas que recuperen datos de varios microservicios, al tiempo que se evita la conversación excesiva con los microservicios desde aplicaciones cliente remotas. Un ejemplo podría ser una pantalla de una aplicación móvil que necesita mostrar información del usuario que proviene del carrito de compras, el catálogo y la identidad del usuario. Otro ejemplo sería un informe complejo que involucre muchas tablas ubicadas en múltiples microservicios. La solución correcta depende de la complejidad de las consultas. Pero, en cualquier caso, será necesaria una forma de consolidar información, para mejorar la eficiencia en las comunicaciones del sistema. Las soluciones más populares son las siguientes.

API Gateway. Para la consolidación de datos de múltiples microservicios con bases de datos diferentes, el enfoque recomendado es un microservicio de agregación denominado API Gateway. Sin embargo, se debe tener cuidado con el despliegue de este patrón, ya que puede ser un "cuello de botella" en su sistema y violar el principio de autonomía de los microservicios. Para mitigar esta posibilidad, puede tener múltiples API Gateways de grano fino, cada una de ellas centrada en un "sector" vertical o área de negocio del sistema. Más adelante se explica con mayor detalle el patrón API Gateway, en la sección sobre uso de una API Gateway.

CQRS con tablas de consulta. Otra solución para agregar datos de múltiples microservicios es el patrón Materialized View. En este enfoque, se genera de antemano, antes de que sucedan las consultas reales, una tabla de solo lectura con los datos (desnormalizados) que provienen de múltiples microservicios. La tabla tiene un formato adecuado a las necesidades de la aplicación del cliente.

Considere algo así como la pantalla de una aplicación móvil. Si tiene una única base de datos, puede juntar los datos para esa pantalla usando una consulta SQL que realiza una combinación compleja que involucra múltiples tablas. Sin embargo, cuando tiene múltiples bases de datos y cada una pertenece a un microservicio diferente, no puede consultar esas bases de datos creado un "join" SQL. Esa consulta compleja se convierte en todo un desafío. Se puede resolver ese requerimiento usando un enfoque CQRS: Se crea una tabla desnormalizada en una base de datos diferente, que se usa sólo para consultas. La tabla se puede diseñar específicamente para los datos que se necesitan para esa consulta compleja, con una relación de uno a uno entre los campos que necesita la pantalla de la aplicación y las columnas en la tabla de consulta. También podría usar esa tabla para generar informes.

Este enfoque no solo resuelve el problema original (cómo consultar y agregar información de varios microservicios); también mejora el rendimiento considerablemente en comparación con un join SQL complejo, porque ya se tienen los datos que necesita la aplicación, en la tabla de consulta. Por supuesto, el uso de Command and Query Responsibility Segregation (CQRS) con tablas de consulta / lectura significa trabajo adicional y será necesario que adoptar la consistencia eventual. Sin embargo, cuando hay requerimientos de rendimiento y alta escalabilidad en escenarios colaborativos (o escenarios competitivos, según el punto de vista) se debe aplicar CQRS con múltiples bases de datos.

"Datos fríos" en bases de datos centrales. Para informes complejos y consultas que no necesitan datos en tiempo real, se suelen exportar sus "datos calientes" (datos transaccionales de los microservicios, en tiempo real) como "datos fríos" (históricos, que no se modifican) a grandes bases de datos que se usan sólo para informes. Ese sistema de base de datos central puede ser un sistema basado en Big Data, como Hadoop, un depósito de datos basado en Azure SQL Data Warehouse o incluso una base de datos SQL única, utilizada sólo para informes (si el tamaño no es un problema).

Tenga en cuenta que esta base de datos centralizada, se usará sólo para consultas e informes que no necesitan datos en tiempo real. Las actualizaciones y transacciones originales, como su fuente original, tienen que estar en las bases de datos de los microservicios. La forma para sincronizar los datos sería mediante comunicación basada en eventos (tratada en las siguientes secciones) o mediante el uso de otras herramientas de importación/exportación de bases de datos. Si se utiliza una comunicación basada en eventos, ese proceso de integración sería similar a como se propagan los datos en el caso de las tablas de consulta CQRS.

Sin embargo, si el diseño de la aplicación implica consolidar información constantemente de múltiples microservicios para consultas complejas, podría ser un síntoma de un mal diseño: un microservicio debería estar lo más aislado posible de otros microservicios. (Excluyendo informes y análisis, que siempre deberían usar bases de datos centrales de datos fríos.) Tener frecuentemente este problema, podría ser una razón para fusionar microservicios. Se debe equilibrar la autonomía de la evolución y el despliegue de cada microservicio, con fuertes dependencias, cohesión y agregación de datos.

Reto #3: Cómo lograr consistencia entre múltiples microservicios

Como se indicó anteriormente, los datos de cada microservicio deben ser privados y sólo se debe poder acceder a ellos usando el API del microservicio. Por lo tanto, un desafío que se presenta es cómo implementar procesos de negocio que involucran a varios microservicios mientras se mantiene la consistencia entre ellos.

Para analizar este problema, veamos un ejemplo de la aplicación de referencia eShopOnContainers. El microservicio Catalog mantiene información sobre todos los productos, incluido su precio. El microservicio Basket administra los datos temporales sobre los artículos en los carritos de compra de los usuarios, que incluye los precios de cuando se agregaron al carrito. Cuando el precio de un producto se actualiza en el catálogo, ese precio también se debe actualizar en los carritos activos que lo contienen, además el sistema probablemente debería advertir al usuario que el precio de un artículo en su carrito ha cambiado desde que lo agregó.

En una hipotética versión monolítica de esta aplicación, cuando el precio cambia en la tabla Product, 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 de Product y Basket son propiedad de sus microservicios respectivos. Ningún microservicio debería usar tablas o

almacenamiento propiedad de otro microservicio, en sus propias transacciones, ni siquiera en consultas directas, como se muestra en la Figura 4-9.

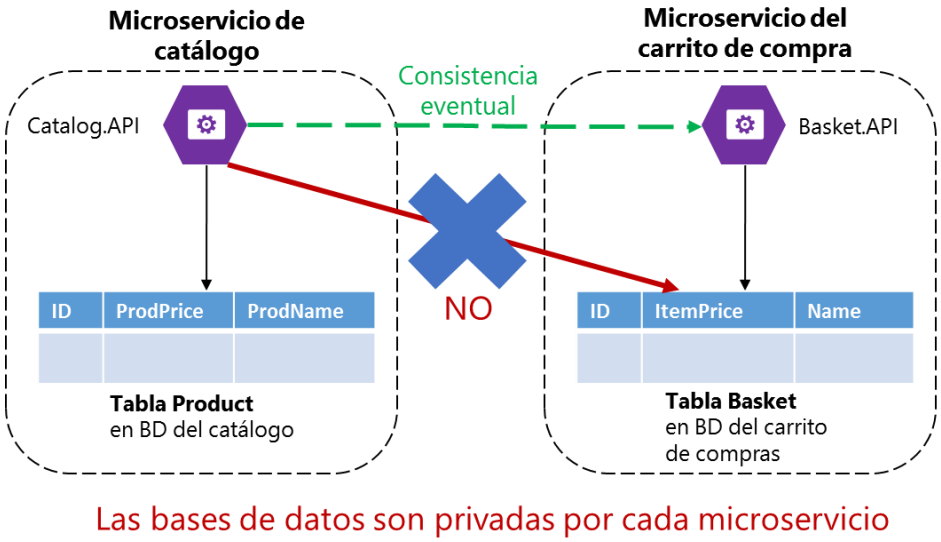


***Figura 4-9**. Un microservicio no puede acceder directamente a una tabla de otro*

El microservicio Catalog no debe actualizar la tabla Basket directamente, ya que la tabla Basket es propiedad del servicio Basket. Para realizar una actualización al microservicio Basket, el microservicio Catalog debe usar la consistencia eventual, basada probablemente en una comunicación asíncrona, como eventos de integración (mensajes y comunicaciones basadas en eventos). Así es como la aplicación de referencia eShopOnContainers consigue este tipo de consistencia en microservicios.

Como se establece en el teorema de CAP, se debe elegir entre la disponibilidad y la consistencia formal de ACID. La mayoría de los escenarios basados en microservicios exigen disponibilidad y alta escalabilidad en lugar de consistencia fuerte. Las aplicaciones de misión crítica deben permanecer en funcionamiento y los desarrolladores pueden buscar formas para no requerir consistencia fuerte, usando técnicas para trabajar con consistencia débil o eventual. Este es el enfoque adoptado por la mayoría de las arquitecturas basadas en microservicios.

Además, las transacciones de tipo ACID o *two-phase commit* (típicas en escenarios de bases de datos distribuidas) no sólo son contrarias a los principios de los microservicios, sino que la mayoría de las bases de datos NoSQL (como Azure Cosmos DB, MongoDB, etc.) no admiten transacciones *two-phase commit*. Sin embargo, es esencial mantener la consistencia de los datos entre los servicios y las bases de datos. Este desafío también está relacionado con la cuestión de cómo propagar los cambios en múltiples microservicios cuando ciertos datos deben ser redundantes, por ejemplo, cuando necesita tener el nombre o la descripción del producto en el microservicio Catalog y en el microservicio Basket.

Por lo tanto, y como conclusión, una buena solución para este problema es utilizar la consistencia eventual entre los microservicios, coordinados a través de una comunicación basada en eventos y un sistema de publicación y suscripción. Estos temas se tratan más adelante, en la sección Comunicación asíncrona basada en eventos.

Reto #4: Cómo diseñar comunicaciones 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 usar (HTTP y REST, AMQP, mensajes, etc.). En cambio, se refiere a qué estilo de comunicación usar y, especialmente, qué tan acoplados deben ser los microservicios. Dependiendo del nivel de acoplamiento, cuando ocurre una falla, el impacto de esa falla en el sistema cambiará significativamente.

En un sistema distribuido, como una aplicación basada en microservicios, con tantas partes móviles y servicios distribuidos en muchos servidores o *hosts*, los componentes eventualmente fallarán. Se producirán fallas parciales e incluso interrupciones más grandes, por lo que se deben diseñar los microservicios y la comunicación entre ellos, considerando los riesgos comunes en este tipo de sistemas.

Un enfoque popular es implementar microservicios basados en HTTP (REST), debido a su simplicidad. Un enfoque basado en HTTP es perfectamente aceptable; el problema aquí está relacionado con cómo se usa. Si se utilizan peticiones HTTP y las respuestas sólo para interactuar con los microservicios desde aplicaciones cliente o desde *API Gateways*, eso está bien. Pero si crea largas cadenas de llamadas HTTP síncronas entre microservicios, comunicándose entre ellos como si fueran objetos en una aplicación monolítica, la aplicación finalmente tendrá problemas.

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

- Bloqueo y bajo rendimiento. Debido a la naturaleza sincrónica de HTTP, la solicitud original no obtendrá una respuesta hasta que todas las llamadas HTTP internas hayan finalizado. Imagine si el número de estas llamadas aumenta significativamente y al mismo tiempo se bloquea una de las llamadas HTTP intermedias a un microservicio. El resultado es que el rendimiento se ve afectado y la escalabilidad general se verá afectada exponencialmente a medida que aumenten las solicitudes HTTP adicionales.
- Acoplamiento de microservicios con HTTP. Los microservicios del negocio no deben combinarse con otros microservicios del negocio. Idealmente, no deberían "saber" sobre la existencia de esos. Si su aplicación depende del acoplamiento de microservicios como en el ejemplo, será casi imposible lograr la autonomía que buscamos.
- Falla en cualquier microservicio. Si se implementa una cadena de microservicios enlazados mediante llamadas HTTP, cuando alguno de los microservicios falle (y eventualmente fallarán) toda la cadena de microservicios fallará. Un sistema basado en microservicios debe diseñarse para continuar funcionando lo mejor posible durante fallas parciales. Incluso si se implementa una lógica de reintentos con retraso exponencial en el cliente, u otra de cualquier naturaleza, cuanto más complejas sean las cadenas de llamadas HTTP, más compleja será la implementación de una estrategia de fallas basada en HTTP.

De hecho, si los microservicios internos se están comunicando creando cadenas de peticiones HTTP como se describe, se podría argumentar que, en realidad, tiene una aplicación monolítica, pero basada en HTTP entre procesos en lugar de mecanismos de comunicación interproceso.

Por lo tanto, para cumplir el principio de autonomía de los microservicios y tener mejor capacidad de recuperación, se debe minimizar el uso de cadenas de comunicación tipo petición/respuesta entre microservicios. Se recomienda utilizar sólo la interacción asíncrona para la comunicación entre microservicios, ya sea mediante comunicación asíncrona basada en eventos y mensajes, o mediante el uso de HTTP *polling* (asíncrono) independiente de la petición/respuesta original.

Más adelante se explica con mayor detalles el uso de la comunicación asíncrona, en las secciones La integración asíncrona refuerza la autonomía de los microservicios y Comunicación asíncrona basada en mensajes.

Additional resources

- **CAP theorem**
  https://en.wikipedia.org/wiki/CAP_theorem
- **Eventual consistency**
  https://en.wikipedia.org/wiki/Eventual_consistency
- **Data Consistency Primer**
  https://msdn.microsoft.com/library/dn589800.aspx
- **Martin Fowler. CQRS (Command and Query Responsibility Segregation)**
  http://martinfowler.com/bliki/CQRS.html
- **Patrón Materialized View**
  https://docs.microsoft.com/azure/architecture/patterns/materialized-view
- **Charles Row. ACID vs. BASE: The Shifting pH of Database Transaction Processing**
  http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/
- **Patrón Compensating Transaction (Transacción de compensación)**
  https://docs.microsoft.com/azure/architecture/patterns/compensating-transaction
- **Udi Dahan. Service Oriented Composition**
  http://udidahan.com/2014/07/30/service-oriented-composition-with-video/

Identificar los límites del modelo de dominio para cada microservicio

Cuando se identifican los límites y el tamaño del modelo para cada microservicio, el objetivo no es llegar a la separación más granular posible, aunque sí se debe tender hacia los microservicios pequeños. El objetivo debe ser llegar a la separación más significativa guiada por el conocimiento de su dominio. El énfasis no está en el tamaño, sino en las capacidades requeridas para el negocio. Por ejemplo, si para un área de la aplicación hay una cohesión evidente, basada en un alto número de dependencias, eso indica la necesidad de un microservicio único. La cohesión es una forma de identificar cómo separar o agrupar microservicios. En última instancia, mientras se adquiere mayor conocimiento sobre el dominio, se debe adaptar el tamaño de los microservicios de forma iterativa. Encontrar el tamaño correcto no es un proceso que se logra en un solo paso.

Sam Newman, un reconocido promotor de microservicios y autor del libro Building Microservices, destaca que se deben diseñar los microservicios basados en el patrón de *Bounded Context* (BC), como se presentó anteriormente. A veces, un BC podría estar compuesto de varios servicios físicos, pero no al revés.

Un modelo de dominio, con sus entidades específicas, tiene sentido dentro de un BC concreto o microservicio. Un BC delimita la aplicabilidad de un modelo de dominio y brinda a los miembros del equipo de desarrollo una comprensión clara y compartida de lo que debe se debe desarrollar dentro del BC o fuera, de manera independiente. Estos son, precisamente, los mismos objetivos que para los microservicios.

Otra herramienta que evidencia las decisiones 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 sucede lo contrario: la organización de la empresa se define por el software. Es posible que sea necesario revertir la ley de Conway y definir los límites de la forma en que desea organizar la empresa, inclinándose hacia la consultoría de procesos del negocio.

Para identificar los *Bounded Contexts*, se puede usar el patrón DDD de Mapeo de contexto (Context Mapping). Con el *Context Mapping* se identifican los distintos contextos en la aplicación y sus límites. Es común tener un contexto y un límite diferente para cada subsistema pequeño, 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, tales como sus entidades de dominio y define los contratos de integración con otros BC. Esto es similar a la definición de un microservicio: es autónomo, implementa cierta capacidad de dominio y debe proporcionar interfaces. Esta es la razón por la que la asignación de contexto y el patrón de *Bounded Context* son buenos enfoques para identificar los límites del modelo de dominio de los microservicios.

Al diseñar una aplicación grande, se suele fragmentar el modelo de dominio: por ejemplo, un experto del dominio de catálogo, nombrará las entidades de manera diferente en los dominios de catálogo e inventario, que un experto del dominio de envíos. O la entidad “Usuario” puede ser diferente en tamaño y número de atributos cuando se trata de un experto en CRM que quiere almacenar cada detalle sobre el cliente que, para un experto en el dominio de pedidos, que sólo necesita datos parciales sobre el cliente. Es muy difícil eliminar la ambigüedad de todos los términos de dominio relacionados en una aplicación grande. Pero lo más importante es no intentar unificarlos, si no, más bien, aceptar las diferencias y la riqueza proporcionadas por cada dominio. Tratar de tener una base de datos unificada para toda la aplicación y unificar el vocabulario, será incómodo y no le parecerá correcto a ninguno de los expertos de los dominios. Por lo tanto, los BC (implementados como

microservicios) le ayudarán a aclarar dónde puede usar ciertos términos de dominio y dónde tendrá que dividir el sistema y crear BC adicionales con diferentes dominios.

Sabremos si se lograron unos límites y tamaños correctos de cada BC y modelo de dominio, cuando se tengan pocas relaciones fuertes entre los distintos modelos de dominio y, generalmente, no sea necesario combinar información de múltiples modelos al realizar operaciones típicas de la aplicación.

Quizás la mejor respuesta a la pregunta de qué tan grande debe ser un modelo de dominio para cada microservicio, sea la siguiente: debe tener un BC autónomo, lo más aislado posible, que permita trabajar sin tener que cambiar constantemente a otros contextos (los modelos de otros microservicios). En la figura 4-10 se puede ver cómo, cada uno de los microservicios (múltiples BC) tiene su propio modelo y cómo se pueden definir sus entidades, según los requisitos específicos de cada dominio de la aplicación.

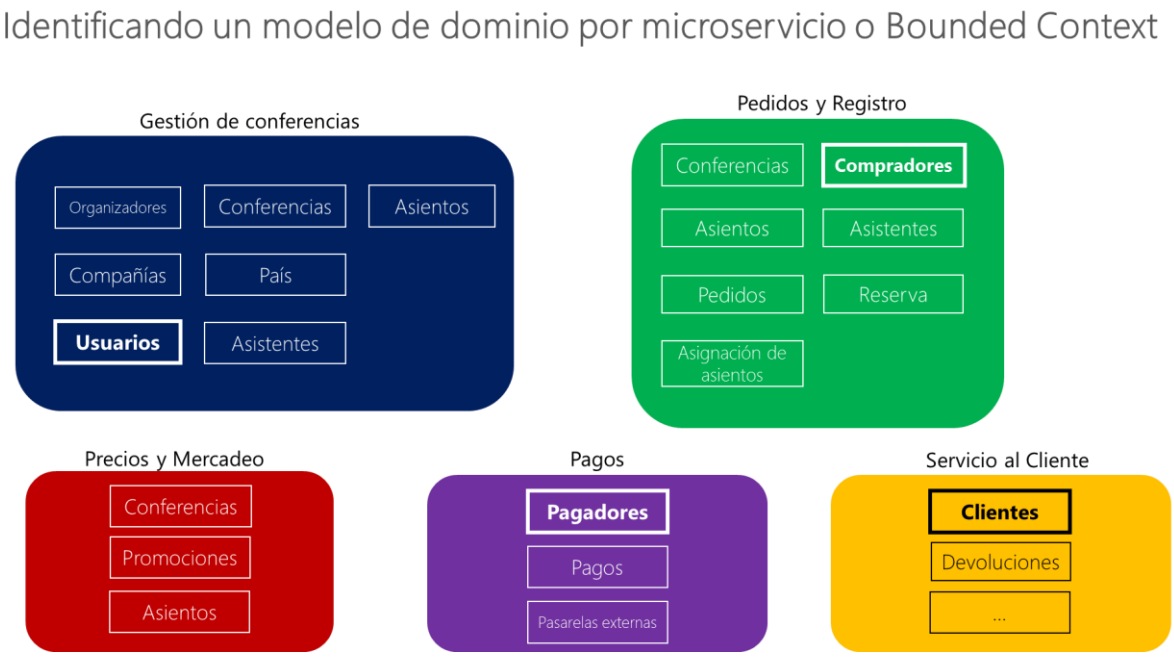


***Figura 4-10**. Identificando entidades y límites de los modelos de los microservicios*

En la Figura 4-10 se muestra un escenario relacionado con un sistema de gestión de conferencias en línea. Se han identificado varios BC que podrían implementarse como microservicios, según los dominios que los expertos de dominio definieron. Como se puede ver, hay entidades que están presentes en un solo modelo de microservicios, como Pagos en el microservicio Pagos. Esos serán fáciles de implementar.

Sin embargo, también podría tener entidades que tengan una forma diferente pero que compartan la misma identidad entre los múltiples modelos de dominio de los múltiples microservicios. Por ejemplo, la entidad Usuario se identifica en el microservicio de Gestión de Conferencias. Ese mismo usuario, con la misma identidad, es el que recibe el nombre de Compradores en el microservicio de pedidos, o el que recibe el nombre de Pagador en el microservicio de pago, e incluso el que se llama Cliente en el microservicio de Servicio al cliente. Esto se debe a que, dependiendo del lenguaje ubicuo que utiliza cada experto de dominio, un usuario puede tener una perspectiva diferente incluso con diferentes atributos. La entidad de usuario en el modelo de microservicios Gestión de conferencias puede tener

como atributos la mayoría de los datos personales. Sin embargo, ese mismo usuario en forma de Pagador en el microservicio de Pagos o en forma de Cliente en el microservicio de Atención al Cliente podría no necesitar la misma lista de atributos.

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

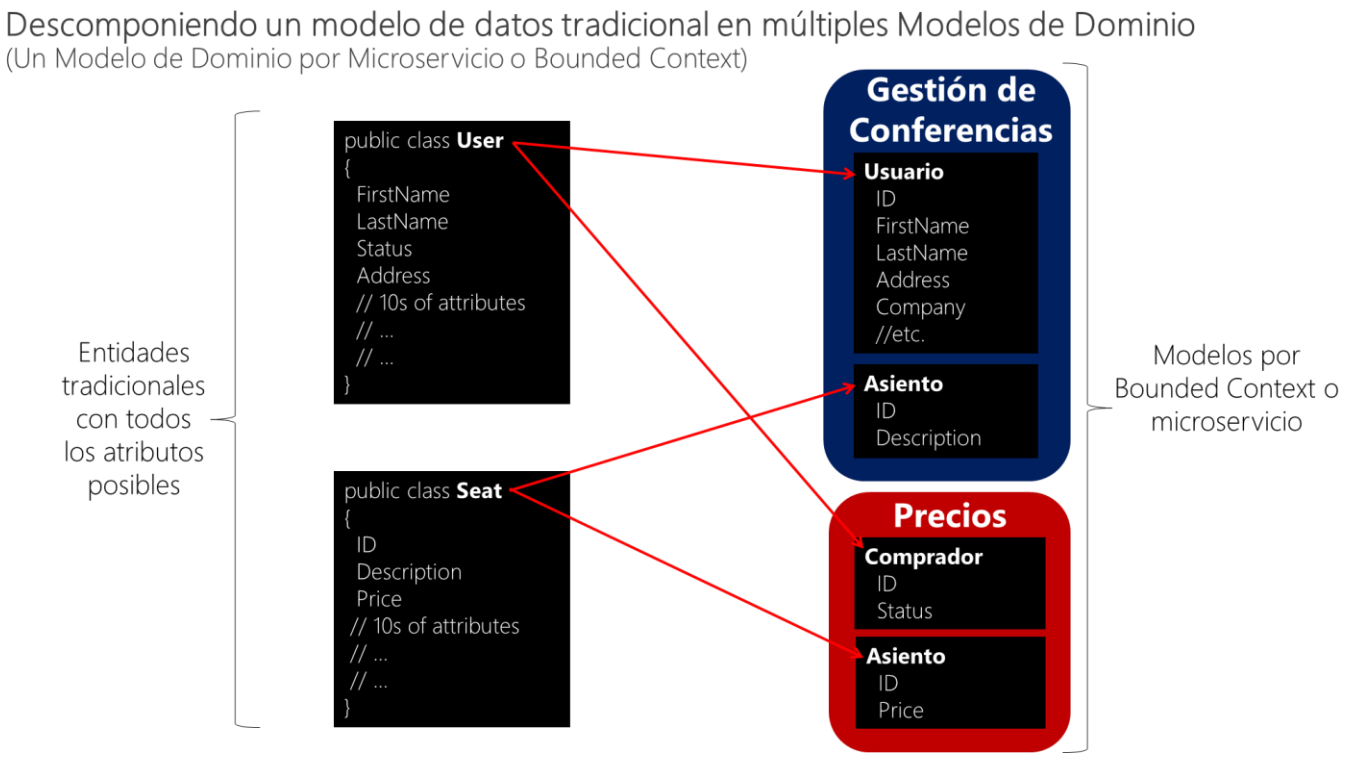


Figura 4-11. *Descomponiendo modelos de datos tradicionales en múltiples modelos de dominio*

Puede ver cómo el usuario está presente en el modelo de microservicio Gestión de Conferencias como la entidad Usuario y también está presente en la forma de la entidad Comprador en el microservicio de Precios, con atributos alternativos o detalles sobre el usuario cuando en realidad es un comprador. Cada microservicio o BC podría no necesitar todos los datos relacionados con la entidad Usuario, sino sólo una parte, dependiendo del problema a resolver o del contexto. Por ejemplo, en el modelo de microservicio Precios, no necesita la dirección o la ID del usuario, sólo 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 Seat tiene el mismo nombre, pero diferentes atributos en cada modelo de dominio. Sin embargo, Seat comparte identidad basada en la misma ID, como ocurre con el usuario y el comprador.

Básicamente, existe un concepto compartido de "usuario" que existe en múltiples servicios (dominios), y todos comparten la identidad de ese usuario. Pero en cada modelo de dominio puede haber detalles adicionales o diferentes sobre la entidad Usuario. Por lo tanto, es necesario que haya una forma de asignar una entidad de usuario de un dominio (microservicio) a otro.

Hay varios beneficios para no compartir la misma entidad "usuario" con el mismo número 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 maestro que posee un cierto tipo de datos por entidad, de modo que las actualizaciones y las consultas para ese tipo de datos sean dirigidas sólo por ese microservicio.

Comunicación directa de cliente a microservicio frente al patrón API Gateway

En una arquitectura de microservicios, cada microservicio expone un conjunto de *endpoints* (urls de acceso) de funciones, por lo general muy específicas. Esto puede afectar la comunicación de cliente a microservicio, como se explica en esta sección.

Comunicación directa de cliente a microservicio

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

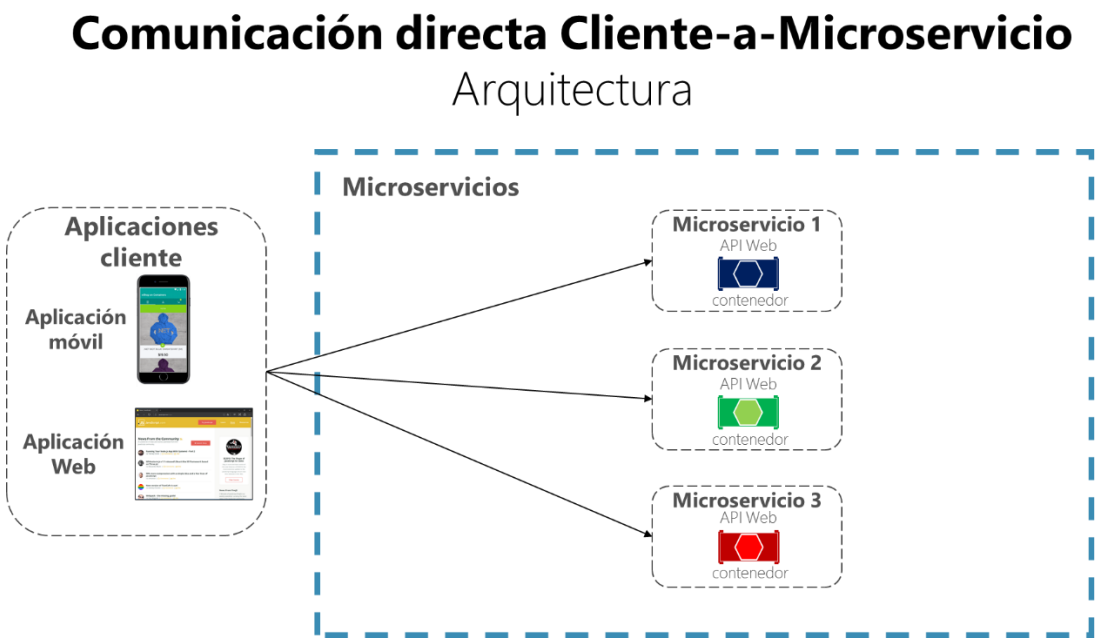


***Figura 4-12**. Usando una arquitectura de comunicación directa de cliente-a-microservicio*

En este caso, cada microservicio tiene un *endpoint* 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 *cluster*, esa URL se correlacionaría con el balanceador de carga del *cluster*, que a su vez se encargaría de distribuir las peticiones entre los microservicios. En entornos de producción, se puede tener un controlador de entrega para aplicaciones (ADC – Application Delivery Controller), como el Azure Application Gateway, entre la Internet y sus microservicios. Esto actúa como una capa de transparente que no sólo se encarga del balanceo de carga, sino que también agrega seguridad a los servicios al ofrecer una conexión SSL. Esto también mejora la carga de trabajo de los *hosts*, ya que se delega el manejo de la conexión SSL y otras tareas

de enrutamiento al Azure Application Gateway. En cualquier caso, un balanceador de carga y un ADC son transparentes, desde el punto de vista de la arquitectura lógica de aplicación.

Una arquitectura de comunicación directa de cliente a microservicio podría ser suficiente para una aplicación pequeña, especialmente si el cliente es una aplicación web, como una aplicación ASP.NET MVC. Sin embargo, cuando se crean aplicaciones grandes y complejas, por ejemplo, cuando maneja docenas de tipos de microservicios y, especialmente, cuando los clientes son aplicaciones móviles o aplicaciones web SPA, ese enfoque tiene algunos inconvenientes.

Al desarrollar una aplicación grande basada en microservicios, hay que considerar los siguientes aspectos:

- *¿Cómo pueden las aplicaciones cliente minimizar la cantidad de peticiones al servidor y reducir la comunicación conversacional a múltiples microservicios?*

  La interacción con múltiples microservicios para construir una pantalla única en la interfaz de usuario, aumenta la cantidad de peticiones/respuestas de la Internet. Esto aumenta la latencia y la complejidad en el lado de la interfaz de usuario. Lo ideal es consolidar las respuestas en el lado del servidor para reducir el número de viajes, con esto reduce la latencia, ya que se regresan múltiples datos en paralelo y algunos elementos de la interfaz de usuario podrían mostrar datos tan pronto como estén listos.

- *¿Cómo se pueden manejar aspectos comunes como la autorización, las transformaciones de datos y el envío dinámico de peticiones?*

  Resolver los aspectos comunes como seguridad y autorización en cada microservicio puede requerir un esfuerzo de desarrollo significativo. Una posible solución es tener esos servicios dentro del *host* Docker o *cluster* interno, para restringir el acceso directo a ellos desde el exterior y para resolverlos en un lugar centralizado, como un API Gateway.

- *¿Cómo pueden comunicarse las aplicaciones cliente con los servicios que usan protocolos que no están adaptados para la Internet?*

  Los protocolos que se suelen utilizar en el lado del servidor (como AMQP o RPC) generalmente no son compatibles con las aplicaciones cliente. Por lo tanto, las solicitudes se deben realizar a través de protocolos como HTTP/HTTPS y luego ser traducidas a los otros protocolos. Un enfoque tipo *man-in-the-middle* puede ayudar en esta situación.

- *¿Cómo se puede estructurar una fachada diseñada especialmente para aplicaciones móviles?*

  Las API de múltiples microservicios podrían no ser adecuadas para las necesidades de todas 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 sean más eficientes. Esto se puede hacer consolidando datos de múltiples microservicios y devolviendo un solo conjunto de datos y, algunas veces, eliminando cualquier dato en la respuesta que no sea necesario para la aplicación móvil. Además de que, por supuesto, también se puede comprimir esa información. Por eso, una fachada o API entre la aplicación móvil y los microservicios puede ser lo más adecuado para este escenario.

Usando una API Gateway

Cuando se diseñan y construyen aplicaciones grandes o complejas, basadas en microservicios y con múltiples aplicaciones cliente, puede ser un buen enfoque considerar es uso de una API Gateway. Este es un servicio que proporciona un punto único de entrada para ciertos grupos de microservicios. Es similar al patrón Fachada del diseño orientado a objetos, pero en este caso, es parte de un sistema distribuido. El patrón API Gateway también se conoce como el "*back-end* para *front-end*" (BFF) porque se construye pensando en las necesidades de la aplicación cliente.

La Figura 4-13 muestra cómo encaja un API Gateway en una arquitectura basada en microservicios.

Es importante resaltar que, en ese diagrama, se estaría usando un servicio de API Gateway único frente a múltiples aplicaciones de cliente. Ese hecho puede representar un riesgo importante porque la API Gateway crecerá y evolucionará según muchos requisitos diferentes de las aplicaciones del cliente. Eventualmente, estará sobrecargado por esas necesidades diferentes y podría, en efecto, ser muy similar a una aplicación o servicio monolítico. Por eso que es muy recomendable dividir la API Gateway en múltiples servicios o *API Gateways* más pequeñas, por ejemplo, una por tipo dispositivo.

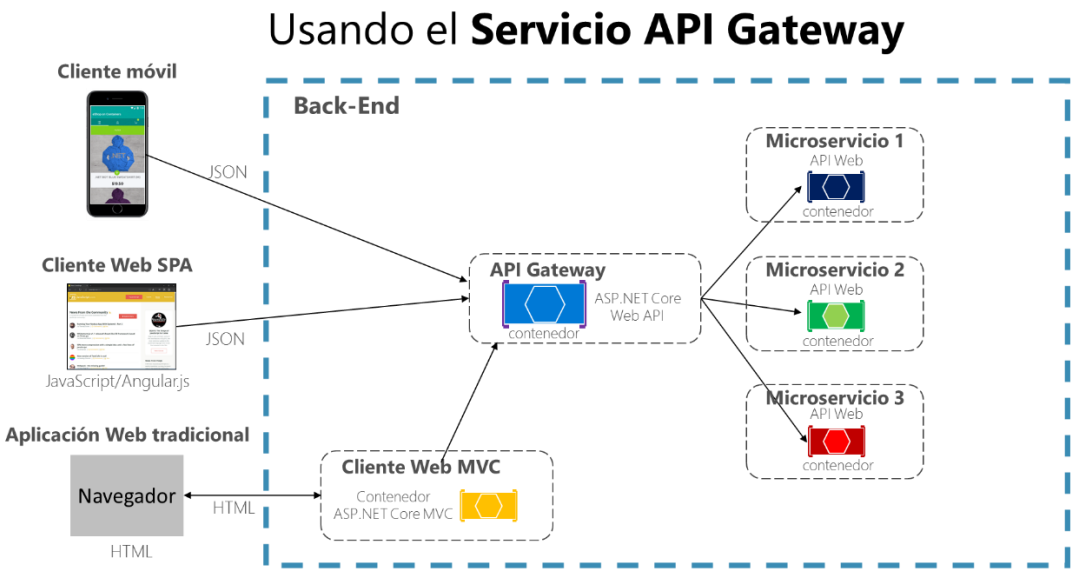


***Figura 4-13**. Usando una API Gateway implementada como un servicio Web API personalizado*

En este ejemplo, la API Gateway se implementaría como una API Web que se ejecuta en un contenedor.

Como se mencionó anteriormente, se deben implementar varias API Gateways para tener una fachada diferente, ajustada a las necesidades de cada aplicación cliente. Cada API Gateway puede proporcionar un API diferente diseñada para cada aplicación cliente, posiblemente en función del tipo de dispositivo, que, por debajo, llame a múltiples microservicios internos.

Para evitar que la consolidación de información en la API Gateway pueda terminar como un agregador monolítico y, entonces, violar la autonomía que buscamos con los microservicios, los API Gateway también deben segregarse en función de los límites de las áreas del negocio, para no actuar como un agregador para toda la aplicación.

Una API Gateway granular también puede ser un microservicio por sí mismo, e incluso puede tener un nombre tomado del dominio o área del negocio y sus datos relacionados. Definir los límites de las API Gateway en función de las áreas del negocio o del dominio ayudará a lograr un mejor diseño.

La granularidad en el nivel de la API Gateway puede ser especialmente útil para aplicaciones de interfaz de usuario compuestas, basadas en microservicios, ya que el concepto de una API Gateway de grano fino es similar a un servicio de composición interfaz de usuario. Esto lo veremos más adelante en Creando una interfaz de usuario compuesta basada en microservicios.

Por lo tanto, el uso de una API Gateway suele ser un buen enfoque para muchas aplicaciones grandes y medianas, pero no como un único agregador monolítico o API Gateway única y centralizada.

Otro enfoque es usar un producto como Azure API Management como se muestra en la Figura 4-14. Este enfoque no sólo resuelve las necesidades de la API Gateway, sino que también ofrece funciones como la recopilación de estadísticas de uso de las API. Tenga en cuenta que la API Gateway es sólo un componente en una solución de gestión de APIs.

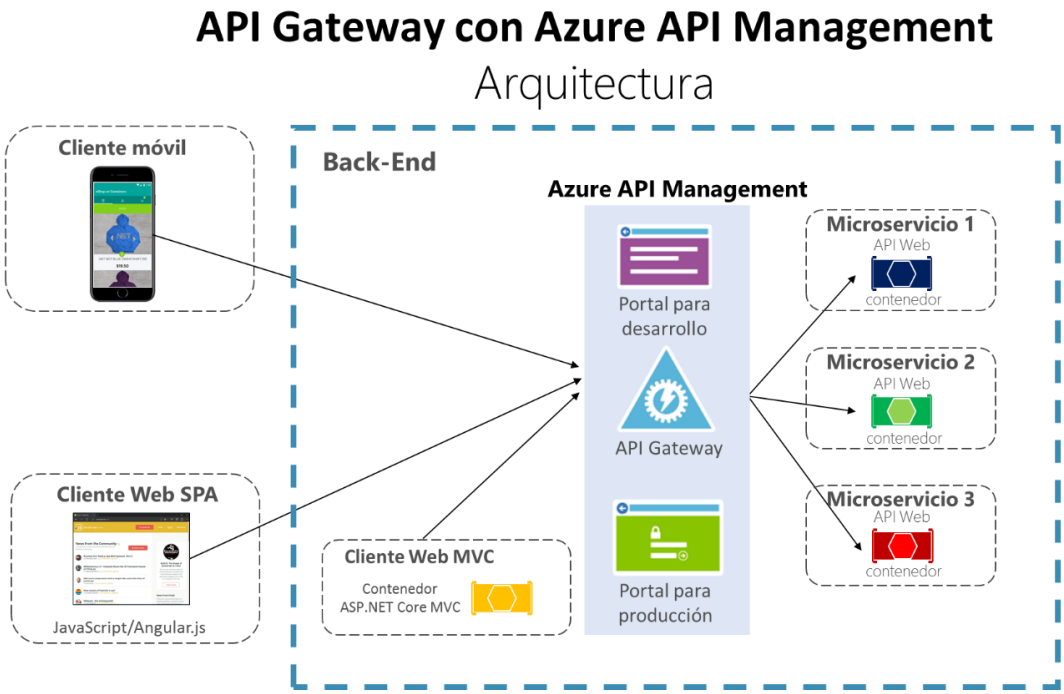


***Figura 4-14**. Usando el Azure API Management para su API Gateway*

En este caso, al usar un producto como Azure API Management, el hecho de que tenga una única puerta de enlace API no es tan arriesgado porque estos tipos de API Gateway son "más delgadas", lo que significa que no implementan código C# personalizado, que podría evolucionar hacia un componente monolítico.

Este tipo de producto actúa más como un proxy inverso para las peticiones, donde también se puede filtrar las API de los microservicios internos y aplicar las políticas de autorización a las API publicadas en esta capa única.

Las métricas disponibles de un sistema de gestión de APIs ayudan a comprender cómo se utilizan y cómo funcionan. Permiten ver informes analíticos casi en tiempo real e identificar tendencias que

podrían afectar el negocio. Además, se pueden tener registros sobre la actividad de petición/respuesta para un mayor análisis, tanto en línea como histórico.

Con Azure API Management, se pueden proteger las API mediante claves, tokens o filtros de IP. Esto facilita la aplicación de cuotas y límites de velocidad flexibles y detallados, modificar la forma y el comportamiento de las API mediante políticas y mejorar el rendimiento con el almacenamiento en caché de las respuestas.

En esta guía y en la aplicación de referencia (eShopOnContainers), estamos limitando la arquitectura a una más sencilla y personalizada, para enfocarnos en contenedores simples sin usar productos PaaS como Azure API Management. Sin embargo, para aplicaciones grandes basadas en microservicios que se implementan en Azure, es altamente recomendable que revise y adopte Azure API Management como base para sus API Gateways.

Inconvenientes del patrón API Gateway

- El inconveniente más importante es que al implementar una *API Gateway*, se está acoplando ese nivel con los microservicios internos. Este acoplamiento podría presentar serias dificultades para la aplicación. Clemens Vasters, arquitecto del equipo de Azure Service Bus, se refiere a esta posible dificultad como "el nuevo ESB (Enterprise Service Bus)" en su sesión de "Messaging and Microservices" en GOTO 2016.
- El uso de una API Gateway crea un posible punto único de falla adicional.
- Una API Gateway puede introducir un mayor tiempo de respuesta debido a la llamada adicional a través de la red. Sin embargo, esta llamada adicional generalmente tiene menos impacto que tener un cliente que realice muchas peticiones directamente a los microservicios internos.
- Si no se escala correctamente, la API Gateway se puede convertir en un cuello de botella.
- Si una API Gateway incluye lógica personalizada y agregación de datos, puede tener un costo adicional de desarrollo y mantenimiento. Los desarrolladores deben actualizar la API Gateway para exponer los *endpoints* de cada microservicio. Además, los cambios de implementación en los microservicios internos pueden provocar cambios de código a nivel de la API Gateway. Sin embargo, si la API Gateway sólo está aplicando seguridad, inicio de sesión y control de versiones (como cuando se usa el Azure API Management), es posible que no se incurra en este coste adicional.
- Si sólo hay un equipo responsable por el desarrollo de la API Gateway, entonces eso se puede convertir en un cuello de botella. Por esto, resulta mejor tener varias API Gateway de grano fino que atiendan las distintas necesidades del cliente. También puede segregar internamente la API Gateway en múltiples áreas o capas, que sean propiedad de los diferentes equipos que trabajan en los microservicios internos.

Recursos adicionales

- **Charles Richardson. Pattern: API Gateway / Backend for Front-End**
  http://microservices.io/patterns/apigateway.html
- **Azure API Management**
  https://azure.microsoft.com/services/api-management/
- **Udi Dahan. Service Oriented Composition**
  http://udidahan.com/2014/07/30/service-oriented-composition-with-video/
- **Clemens Vasters. Messaging and Microservices at GOTO 2016** (video)
  https://www.youtube.com/watch?v=rXi5CLjIQ9k

Las comunicaciones en una arquitectura de microservicios

En una aplicación monolítica que se ejecuta en un proceso único, los componentes se invocan entre sí utilizando llamadas a métodos o funciones que están a nivel de lenguaje. Estos pueden estar fuertemente acoplados si se están creando objetos con código (por ejemplo, **`new ClassName()`**) o se pueden invocar de forma desacoplada, si se está usando Inyección de Dependencias, 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 las llamadas a métodos en llamadas RPC a los servicios generará una comunicación excesiva y poco eficiente, que no funcionará bien en entornos distribuidos. Los desafíos de diseñar un sistema distribuido de manera adecuada son tan conocidos que incluso existe un estándar de Las falacias de la computación distribuida, que enumera los supuestos que los desarrolladores suelen hacer al pasar de diseños monolíticos a diseños distribuidos.

No hay una solución, sino varias. Una opción es aislar los microservicios del negocio tanto como sea posible. A continuación, utilizar la comunicación asíncrona entre los microservicios internos y reemplazar la comunicación detallada, típica de la comunicación entre objetos, con una comunicación más agregada. Esto se puede hacer agrupando llamadas y devolviendo al cliente los datos que consolidan los resultados de varias llamadas internas.

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 es típicamente 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 "*endpoints* inteligentes y conexiones tontas". Este lema promueve un diseño lo más desacoplado posible entre los microservicios y lo más cohesivo posible dentro de un mismo microservicio. Como se explicó anteriormente, cada microservicio posee sus propios datos y lógica de dominio. Pero los microservicios que componen una aplicación de principio a fin, simplemente se organizan 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 de negocio centralizados.

Los dos protocolos utilizados comúnmente son petición/respuesta HTTP con APIs de recursos (principalmente para consultas) y mensajería asíncrona para realizar actualizaciones entre microservicios. Esto se explica con más detalle en las siguientes secciones.

Tipos de comunicación

Los clientes y los servicios pueden tener distintos tipos de comunicación, cada una de los cuales apunta a un escenario y objetivos diferentes. Inicialmente, esos tipos de comunicaciones se pueden clasificar en dos ejes o dimensiones.

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

- **Protocolo síncrono**. HTTP es un protocolo síncrono. El cliente envía una petición (*request*) y espera una respuesta (response) del servicio. Esto es independiente de la ejecución del código del cliente, que podría ser síncrona (el hilo está bloqueado) o asíncrona (el hilo no está bloqueado y la respuesta finalmente llegará a un *callback* – una función que recibe el

resultado). El punto importante aquí es que el protocolo (HTTP/HTTPS) es síncrono y el código del cliente sólo puede continuar su tarea cuando recibe la respuesta del servidor HTTP.

- **Protocolo asíncrono**. Otros protocolos como AMQP (un protocolo admitido por muchos sistemas operativos y entornos de nube) usan mensajes asíncronos. El código del cliente o el remitente del mensaje generalmente no espera una respuesta. Simplemente envía el mensaje como cuando se envía un mensaje a una cola de RabbitMQ o cualquier otro intermediario de mensajes.

El segundo eje define si la comunicación tiene uno o múltiples receptores:

- **Receptor individual**. Cada petición debe ser procesada por exactamente un receptor o servicio. Un ejemplo de esta comunicación es el patrón Comando.
- **Receptores múltiples**. Cada petición se puede procesar por muchos o ningún receptor. Este tipo de comunicación debe ser asíncrona. Un ejemplo es el mecanismo de publicación/suscripción utilizado en patrones como la arquitectura orientada a eventos. Esto se basa en una interfaz de bus de eventos o agente de mensajes (*message broker*) al propagar actualizaciones de datos entre múltiples microservicios a través de eventos; generalmente se implementa a través de un bus de servicio o artefacto similar como Azure Service Bus mediante el uso de los temas y suscripciones del servicio.

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

Es bueno conocer estas dos dimensiones, para tener claridad sobre los posibles mecanismos de comunicación, sin embargo, estás no son las preocupaciones importantes al desarrollar microservicios. Ni la naturaleza asíncrona de la ejecución ni del protocolo seleccionado, son los puntos importantes al integrar microservicios. Lo importante es poder integrar sus microservicios de forma asíncrona, manteniendo la independencia entre ellos, como se explica en la siguiente sección.

La integración asíncrona refuerza la autonomía de los microservicios

Como se mencionó anteriormente, el punto más importante al desarrollar una aplicación basada en microservicios es la integración de éstos. La idea principal es hacer lo posible para minimizar la comunicación entre los microservicios internos. Pero, por supuesto, en muchos casos se deben integrar de algún modo. Cuando se necesita esto, la regla clave es que la comunicación sea asíncrona. Eso no significa que se tenga que usar un protocolo específico (por ejemplo, mensajería asíncrona versus HTTP síncrono). Simplemente significa que la comunicación entre los microservicios se debe hacer sólo mediante la propagación de datos de forma asíncrona, haciendo todo lo posible para no depender de otros microservicios internos como parte de la petición/respuesta HTTP del servicio inicial.

Si es posible, nunca dependa de la comunicación síncrona (petición/respuesta) entre múltiples microservicios, ni siquiera para las consultas. El objetivo de cada microservicio es ser autónomo y estar disponible para el cliente, incluso si los otros servicios que forman parte de la totalidad de la aplicación están inactivos o funcionando mal. Si es necesario hacer una llamada desde un microservicio a otro (como realizar una petición HTTP para una consulta de datos) para poder proporcionar una respuesta a la aplicación cliente, entonces estaremos frente a una arquitectura que no será resiliente, cuando fallen algunos microservicios.

Además, tener dependencias HTTP entre microservicios, como cuando se crean cadenas largas de peticiones/respuestas HTTP, como se muestra en la primera parte de la Figura 4-15, no sólo hace que los microservicios no sean autónomos, sino que su rendimiento se vea afectado tan pronto como uno de los servicios en esa cadena no esté funcionando bien.

Cuantas más dependencias síncronas entre microservicios existan, tales como consultas u operaciones relacionadas, peor será el tiempo de respuesta general para las aplicaciones cliente.

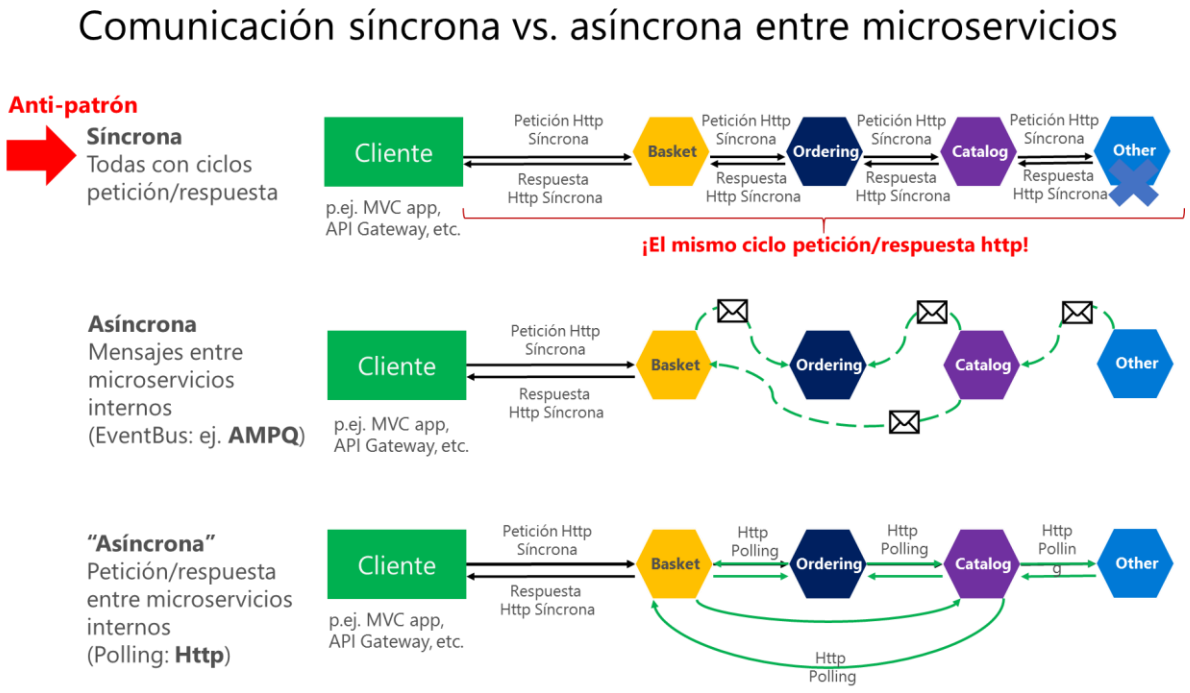


Figura 4-15. *Patrones y anti-patrones en la comunicación entre microservicios*

Si un microservicio necesita iniciar una acción adicional en otro microservicio, si es posible, no lo haga de forma síncrona como parte de la petición/respuesta original. En cambio, hágalo de forma asíncrona (utilizando mensajes asíncronos o eventos de integración, colas, etc.).

Finalmente, y es aquí donde surgen la mayoría de los problemas al construir microservicios, si el microservicio inicial necesita datos que son propiedad de otros, no se debe confiar en realizar peticiones síncronas para obtenerlos. En su lugar, se deben replicar o propagar esos datos, pero sólo los atributos necesarios, a la base de datos del servicio inicial (el que recibe la petición inicial), mediante el uso de consistencia eventual (generalmente mediante 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, el duplicar algunos datos en varios microservicios no es un diseño incorrecto. Por el contrario, eso puede traducir los datos al lenguaje específico o a los términos de ese dominio adicional o *Bounded Context*. Por ejemplo, en la aplicación eShopOnContainers se tiene un microservicio llamado Identity.API que está a cargo de la mayoría de los datos del usuario con una entidad llamada User. Sin embargo, cuando es necesario guardar datos del usuario en del microservicio de Pedidos, se hace como una entidad diferente llamada Comprador. La entidad Comprador comparte la misma identificación con la entidad Usuario original, pero puede tener sólo los pocos atributos necesarios para el dominio Pedidos y no el perfil completo del usuario.

Para tener una consistencia eventual, se puede utilizar cualquier protocolo para comunicar y propagar datos de forma asíncrona entre microservicios. Como se mencionó anteriormente, se podrían usar eventos de integración con bus de eventos o un *message broker* o, incluso, se podría usar HTTP *polling* hacia otros servicios. En realidad, no importa. La importante es no crear dependencias síncronas entre los microservicios.

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

Estilos de comunicación

Existen muchos protocolos y opciones que pueden usar, según el tipo de comunicación que se requiera. Si se está usando un mecanismo basado en petición/respuesta síncrona, el protocolo HTTP y el enfoque REST son los más comunes, especialmente si los servicios están publicados fuera del *host* Docker o del *cluster* de microservicios. Para la comunicación interna entre servicios (dentro del servidor Docker o *cluster* de microservicios), también pueden utilizar mecanismos de comunicación binarios, como la comunicación remota de Service Fabric o WCF utilizando el protocolo TCP. Por otro lado, también se pueden usar mecanismos asíncronos basados en mensajes como AMQP.

También hay múltiples formatos de mensaje como JSON o XML, o incluso formatos binarios, que pueden ser más eficientes. Si el formato binario seleccionado no es un estándar, probablemente no sea una buena idea publicar los servicios con ese formato. Se puede usar un formato no estándar para la comunicación interna entre los microservicios, si trabaja dentro de un *host* Docker o *cluster* de microservicio (orquestadores Docker o Azure Service Fabric), o para aplicaciones cliente propietarias que hablen con los microservicios.

La comunicación petición/respuesta con HTTP y REST

Cuando un cliente usa la comunicación petición/respuesta, envía una petición a un servicio, luego el servicio procesa la procesa y envía una respuesta. La comunicación petición/respuesta es

especialmente adecuada para consultar datos de una interfaz de usuario en tiempo real desde las 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.

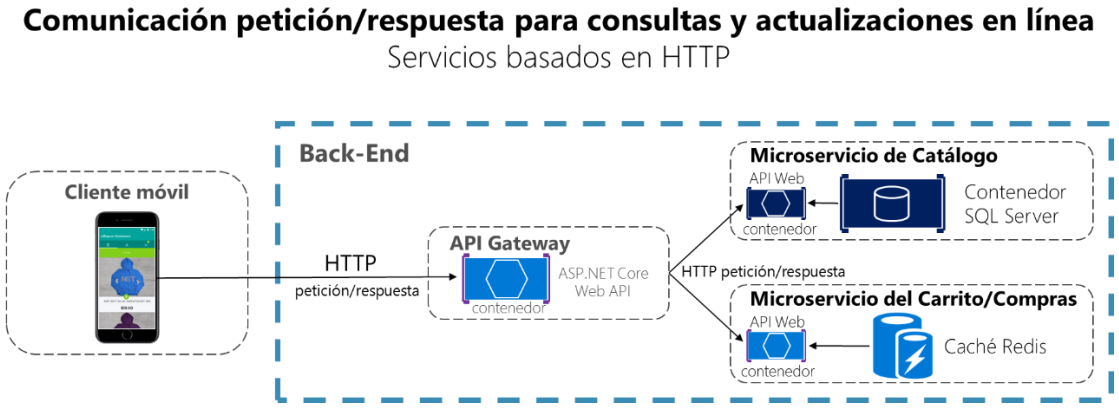


***Figura 4-16**. Usando comunicación por petición/respuesta HTTP (síncrona o asíncrona)*

Cuando un cliente usa la comunicación de petición/respuesta, supone que la respuesta llegará en un tiempo corto, generalmente menos de un segundo o unos pocos segundos como máximo. Para las respuestas con demora, se debe implementar una comunicación asíncrona basada en patrones de mensajes y tecnologías de mensajería, que es un enfoque diferente que explicamos en la siguiente sección.

Un estilo arquitectónico popular para la comunicación petición/respuesta es REST. Este enfoque se basa en el protocolo HTTP y está estrechamente relacionado con él, comprende los verbos HTTP como GET, POST y PUT. REST es el enfoque de comunicación que se utiliza con más frecuencia al crear servicios. Se pueden implementar servicios REST al trabajar con ASP.NET Core Web API.

Hay una ventaja adicional al usar los servicios HTTP REST como el lenguaje de definición de la interfaz. Por ejemplo, si se utilizan los metadatos de Swagger para describir la API del servicio, se pueden usar herramientas que generen *stubs* para los clientes, que pueden descubrir y consumir directamente los servicios.

Recursos adicionales

- **Martin Fowler. Richardson Maturity Model.** A description of the REST model.
  http://martinfowler.com/articles/richardsonMaturityModel.html
- **Swagger.** The official site.
  http://swagger.io/

Comunicación "push" y en tiempo real basada en HTTP

Otra posibilidad (generalmente para usos diferentes de REST) es una comunicación en tiempo real y uno a muchos con *frameworks* 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 el servidor puede enviar el contenido a los clientes conectados, a medida que los datos estén disponibles, en lugar de que el servidor espere las peticiones de nuevos datos por los clientes.

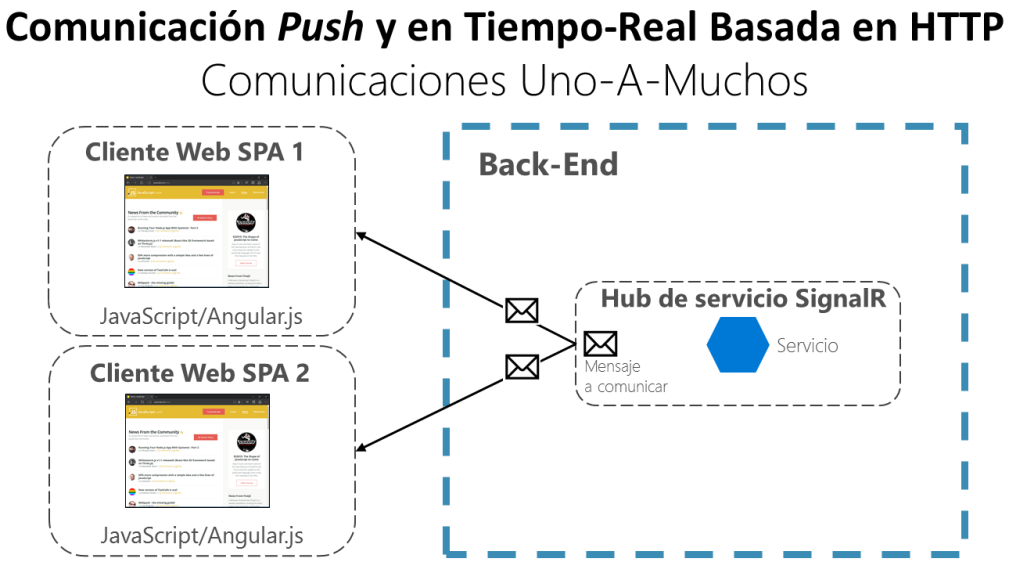


***Figura 4-17**. Comunicación asíncrona por mensaje en tiempo real uno-a-muchos*

Dado que la comunicación se realiza en tiempo real, las aplicaciones cliente muestran los cambios casi instantáneamente. Esto generalmente es manejado por un protocolo como WebSockets, usando muchas conexiones, una por cliente. Un ejemplo típico es cuando un servicio comunica un cambio en el puntaje de un juego de deportes a muchas aplicaciones web cliente simultáneamente.

Comunicación basada en mensajes asíncronos

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

Al usar mensajería, los procesos se comunican intercambiando mensajes de forma asíncrona. Un cliente realiza un comando o una petición a un servicio enviándole un mensaje. Si el servicio necesita responder, envía un mensaje diferente al cliente. Como se trata de una comunicación basada en mensajes, el cliente supone que la respuesta no se recibirá de inmediato y que es posible que no haya respuesta.

Un mensaje está compuesto por un encabezado (metadatos como identificación o información de seguridad) y un cuerpo. Los mensajes se envían generalmente a través de protocolos asíncronos como AMQP.

La infraestructura preferida para este tipo de comunicación en la comunidad de microservicios es un *message broker* liviano, que es distinto de los grandes *brokers* y orquestadores utilizados en SOA. En un bróker ligero de mensajería, la infraestructura suele ser "tonta", actuando sólo como un intermediario, con implementaciones simples como RabbitMQ o un bus de servicio escalable en la nube como Azure Service Bus. En este escenario, la mayoría del procesamiento "inteligente" aún vive en los *endpoints* que producen y consumen mensajes, es decir, en los microservicios.

Otra regla que se debe seguir, en la medida de lo posible, es usar sólo mensajes asíncronos entre los servicios internos y usar comunicación síncrona (como HTTP) sólo desde las aplicaciones cliente a los servicios de interfaz de usuario (*API Gateways* más el primer nivel de microservicios).

Hay dos tipos de mensajería asíncrona: comunicación basada en mensajes de receptor único o en mensajes de receptores múltiples. En las siguientes secciones entraremos en detalles sobre ellos.

Comunicaciones de receptor único

La comunicación asincrónica basada en mensajes con un receptor único, significa que hay una comunicación punto a punto que envía un mensaje a exactamente uno de los consumidores que está leyendo desde el canal y que el mensaje se procesa sólo 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 la red u otras fallas, el cliente debe poder volver a intentar el envío de mensajes y el servidor debe implementar la lógica necesaria para que una operación sea idempotente, para que cada mensaje sólo se procese una vez.

La comunicación basada en mensajes de un receptor único, 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 se comience a enviar comunicaciones basadas en mensajes (ya sea con comandos o eventos), se debe evitar mezclarlas con la comunicación HTTP síncrona.

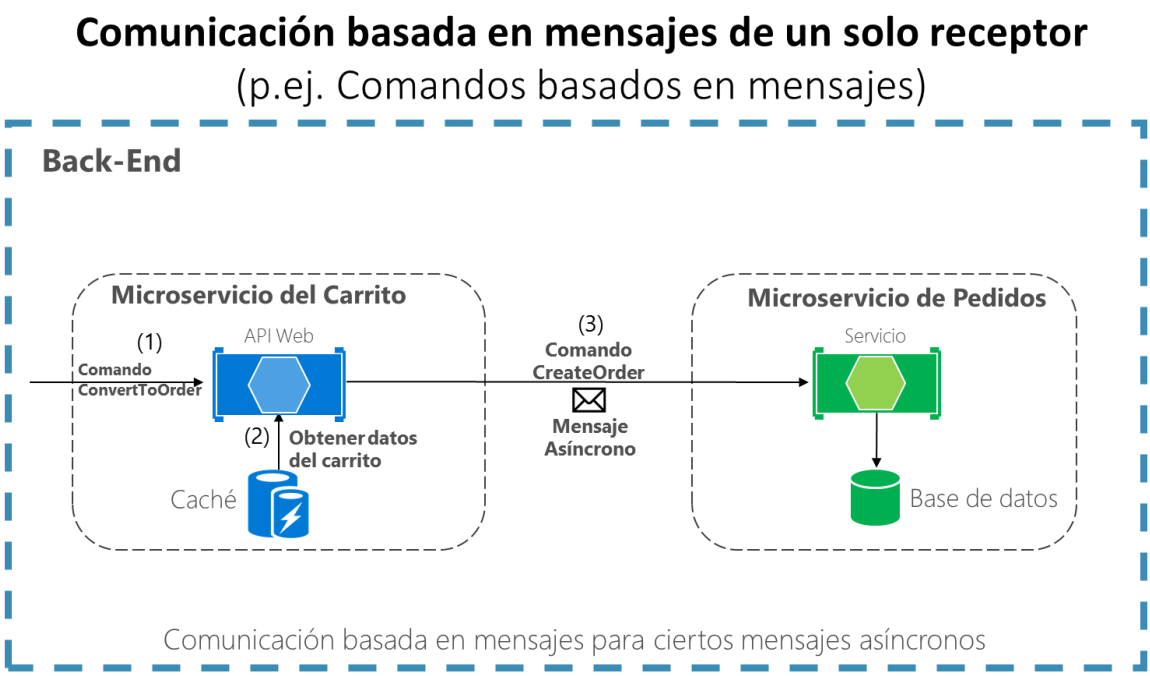


***Figura 4-18**. Un microservicio recibiendo un mensaje asíncrono*

Tenga en cuenta que cuando los comandos provienen de aplicaciones cliente, se pueden implementar como comandos síncronos HTTP. Se deben usar comandos basados en mensajes cuando se necesite mayor escalabilidad o cuando ya se esté trabajando en un proceso del negocio basado en mensajes.

Comunicaciones basadas en mensajes a receptores múltiples

Para lograr un enfoque más flexible, también se puede usar un mecanismo de publicación/suscripción para que un mensaje del remitente esté disponible para los microservicios subscriptores o para aplicaciones externas. Esto le ayuda a seguir el principio open/closed en el servicio de envío. De esta forma, se pueden agregar suscriptores adicionales en el futuro sin la necesidad de modificar el servicio del remitente.

Cuando utiliza una comunicación por publicación/suscripción, es posible que esté utilizando una interfaz tipo bus de eventos para publicar eventos hacia cualquier suscriptor.

Comunicación asíncrona basada en eventos

Cuando se utiliza una comunicación asíncrona basada en eventos, un microservicio publica un evento de integración cuando ocurre algo dentro de su dominio y otro microservicio debe enterarse de ello, como un cambio de precio en un microservicio de catálogo de productos. Los microservicios adicionales se suscriben a los eventos para que puedan recibirlos de forma asíncrona. Cuando eso sucede, los receptores pueden actualizar sus propias entidades de dominio, lo que puede provocar que se publiquen más eventos de integración. Este sistema de publicación/suscripción generalmente se realiza mediante una implementación de un bus de eventos. El bus de eventos se puede diseñar como una abstracción o interfaz, con la API que se necesita para publicar eventos y suscribirse o anular la suscripción a eventos. El bus de eventos también puede tener una o más implementaciones basadas en cualquier bróker entre procesos y mensajería, como una cola de mensajería o un bus de servicio que admite comunicación asíncrona y un modelo de publicación/suscripción.

Se recomienda que sea completamente claro para el usuario final, si un sistema utiliza consistencia eventual manejada por eventos de integración. El sistema no debe usar un enfoque que imite los eventos de integración, usando SignalR o *polling* desde el cliente. Tanto los usuarios finales como el dueño del negocio deben aceptar explícitamente la consistencia eventual en el sistema y darse cuenta de que, en muchos casos, esto no genera ningún problema para el negocio, siempre que esto sea explícito. Esto es importante porque los usuarios podrían esperar ver algunos resultados de inmediato y esto puede no ocurrir con la consistencia eventual.

Como se señaló anteriormente en la sección de Retos y soluciones para la administración de datos distribuidos, se pueden usar eventos de integración para implementar tareas del negocio que abarcan múltiples microservicios. Por lo tanto, tendrá una consistencia eventual 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 plantea la siguiente acción dentro de la misma tarea del negocio, considerada como un todo.

Un punto importante es que se puede necesitar comunicación con múltiples microservicios que están suscritos al mismo evento. Para hacerlo, se pueden usar los mensajes de publicación/suscripción basados en 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 *Bounded Contexts* en DDD, o la forma en que propagan las actualizaciones desde la base de datos de escritura hacia la base de datos de consulta en el patrón arquitectónico Command and Query Responsibility Segregation (CQRS). El objetivo es tener una consistencia eventual entre múltiples fuentes de datos en un sistema distribuido.

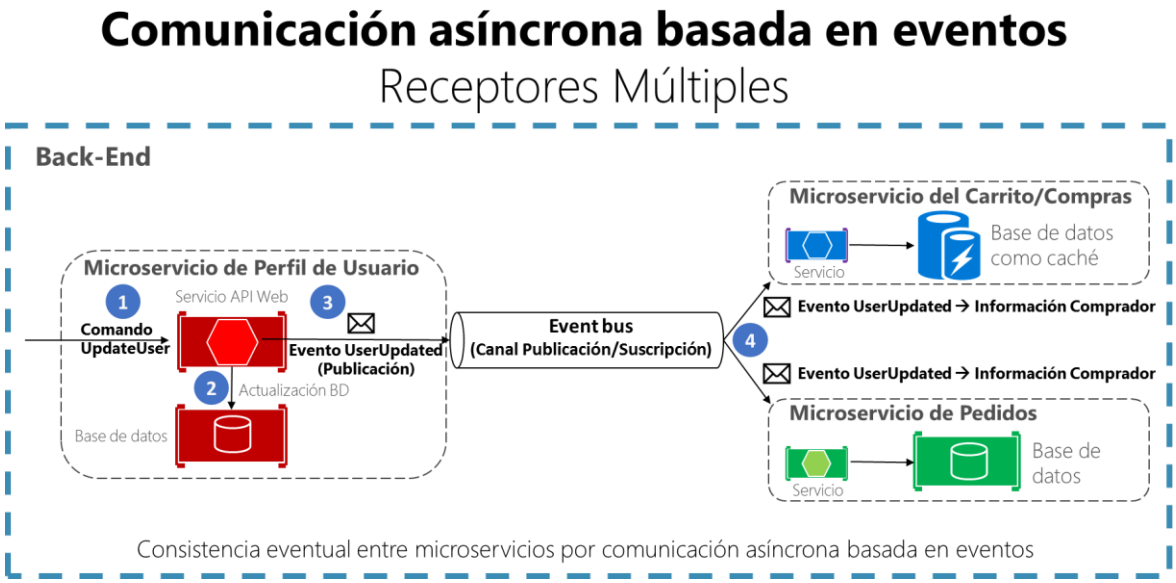


***Figura 4-19**. Comunicación asíncrona por mensaje basada en eventos*

La implementación determinará qué protocolo usar para las comunicaciones basadas en mensajes y basadas en eventos. AMQP puede ayudar a lograr una comunicación confiable a través de una cola.

Cuando se usa un bus de eventos, se puede usar un nivel de abstracción (como una interfaz de bus de eventos) basado en una implementación usando la API de un *message broker* como RabbitMQ o un

bus de servicio como Azure Service Bus with Topics. También se puede usar un bus de servicio de nivel superior como NServiceBus, MassTransit o Brighter para coordinar el bus de eventos y el sistema de publicación/suscripción.

Un comentario sobre las tecnologías de mensajería para sistemas de producción

Hay diferentes niveles en las tecnologías de mensajería disponibles para implementar un bus de eventos abstracto. Por ejemplo, productos como RabbitMQ y Azure Service Bus se ubican en un nivel inferior al de otros productos como, NServiceBus, MassTransit o Brighter, que pueden funcionar sobre RabbitMQ y Azure Service Bus. La elección depende de la cantidad de características avanzadas en el nivel de aplicación y la escalabilidad que necesita para su aplicación. Para implementar sólo un bus de eventos como prueba de concepto para su entorno de desarrollo, como lo hemos hecho en el ejemplo de eShopOnContainers, podría ser suficiente una implementación simple sobre RabbitMQ ejecutándose en un contenedor Docker.

Sin embargo, para los sistemas de misión crítica y de producción que necesitan alta escalabilidad, se debería evaluar el Azure Service Bus. Para abstracciones de mayor nivel y características que facilitan el desarrollo de aplicaciones distribuidas, recomendamos evaluar otros buses de servicio comerciales y *open source*, como NServiceBus, MassTransit y Brighter. Por supuesto, también se pueden construir funciones propias de bus de servicio usando tecnologías de menor nivel de abstracción como RabbitMQ y Docker. Pero ese trabajo de fontanería puede resultar demasiado costoso para una aplicación empresarial a la medida.

Publicación resiliente a un bus de servicios

Un desafío al implementar una arquitectura basada en eventos entre múltiples microservicios, es cómo actualizar atómicamente el estado en el microservicio original y publicar de forma flexible su evento de integración relacionado en el bus de eventos, basado de alguna manera en transacciones. Estas son algunas formas de lograrlo, aunque también podría haber otros enfoques.

- Usar una cola transaccional (basada en DTC) como MSMQ. (Sin embargo, este es un enfoque anticuado).
- Usando transaction log mining
- Usando el patrón Event Sourcing.
- Uso del patrón de Bandeja de salida: una tabla de base de datos transaccional que funcione como una cola de mensajes, que servirá de base para crear y publicar el evento con un componente creador de eventos.

Otros Temas que se deben considerar cuando se usa comunicación asíncrona, son la idempotencia del mensaje y la deduplicación del mensaje. 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

- **Event Driven Messaging**
  http://soapatterns.org/design_patterns/event_driven_messaging
- **Publish/Subscribe Channel**
  http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
- **Udi Dahan. Clarified CQRS**
  http://udidahan.com/2009/12/09/clarified-cqrs/

- **Patrón Command and Query Responsibility Segregation (CQRS)**
  https://docs.microsoft.com/azure/architecture/patterns/cqrs
- **Communicating Between Bounded Contexts**
  https://msdn.microsoft.com/library/jj591572.aspx
- **Eventual consistency**
  https://en.wikipedia.org/wiki/Eventual_consistency
- **Jimmy Bogard. Refactoring Towards Resilience: Evaluating Coupling**
  https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/

Creando, versionando y evolucionando APIs y contratos de microservicios

Una API es un contrato entre el servicio y sus clientes. Un microservicio podrá evolucionar de forma independiente sólo si no rompe el contrato de su API y es por esto que el contrato es tan importante. Si cambia el contrato, tendrá un impacto en sus aplicaciones cliente o en su API Gateway.

La naturaleza de la definición de la API depende del protocolo que esté utilizando. Por ejemplo, si usa mensajes (como AMQP), la API está definida por los tipos de mensajes. Si está utilizando servicios HTTP y RESTful, la API está definida por las URL y los formatos JSON de petición y respuesta.

Sin embargo, incluso si define cuidadosamente a su contrato inicial, una API de servicio deberá evolucionar con el tiempo. Cuando eso sucede y especialmente si la API es pública y consumida por múltiples aplicaciones cliente, normalmente no se puede obligar a todos los clientes a actualizarse al nuevo contrato API. Por lo general, se necesita desplegar incrementalmente las nuevas versiones de un servicio, de manera que las versiones antiguas y nuevas de un contrato de servicio se ejecuten simultáneamente. Por lo tanto, es importante tener una estrategia para el control de versiones de la API del servicio.

Cuando los cambios de API son pequeños, por ejemplo, si agregan atributos o parámetros, los clientes que usan una API anterior deberían poder cambiar y trabajar con la nueva versión del servicio. Es posible que pueda proporcionar valores predeterminados para cualquier atributo faltante que se requiera y los clientes podrían ignorar cualquier atributo de respuesta adicional.

Sin embargo, a veces es necesario realizar cambios importantes e incompatibles a una API de servicio. Como es posible que no pueda obligar a las aplicaciones o servicios clientes a actualizarse de inmediato a la nueva versión, un servicio debe admitir versiones anteriores de la API por algún tiempo. Si está utilizando un mecanismo basado en HTTP, como REST, un enfoque es insertar 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 en la misma instancia de servicio o implementar instancias diferentes y que cada una maneje una versión de la API. Una buena solución para esto es el patrón Mediator (por ejemplo, la librería MediatR) para desacoplar las diferentes versiones de implementación en manejadores independientes.

Finalmente, si está utilizando una arquitectura REST, Hypermedia es la mejor solución para versionar sus servicios y permitir API evolutivas.

Recursos adicionales

- **Scott Hanselman. ASP.NET Core RESTful Web API versioning made easy**
  http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx

- **Versioning a RESTful web API**
  https://docs.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api
- **Roy Fielding. Versioning, Hypermedia, and REST**
  https://www.infoq.com/articles/roy-fielding-on-versioning

Direccionabilidad de los microservicios y el registro de servicios

Cada microservicio tiene un nombre único (URL) que se usa para resolver su ubicación. Un microservicio debe ser direccionable, es decir, debe tener una dirección para poder llegar a él, donde sea que se esté ejecutando. Si tiene que pensar en qué computadora está ejecutando un microservicio en particular, las cosas pueden ir mal rápidamente. De la misma manera que el DNS resuelve una URL a una computadora en particular, un microservicio necesita 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. Esto implica que hay una interacción entre cómo se implementa su servicio y cómo se descubre, porque es necesario que haya un registro de servicios. En el mismo sentido, cuando una computadora falla, el registro de servicios 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 necesita estar altamente disponible y actualizado. Los clientes pueden almacenar en caché las ubicaciones de red obtenidas del registro de servicio. Sin embargo, esa información finalmente se desactualiza y los clientes ya no pueden descubrir instancias de servicio. En consecuencia, un registro de servicio consta de un *cluster* de servidores que utiliza un protocolo de replicación para mantener la consistencia.

El servicio de descubrimiento de microservicios está incorporado en algunos entornos de despliegue (llamados *clusters*, que se tratarán en una sección posterior). Por ejemplo, en un entorno de Azure Container Service, Kubernetes y DC/OS con Marathon, se puede gestionar el registro y la eliminación de las instancias del servicio. También ejecutan un proxy en cada *host* del *cluster* que funciona como enrutador de descubrimiento del lado del servidor. Otro ejemplo es el Azure Service Fabric, que también proporciona un registro de servicios a través de su servicio de nombres.

Tenga en cuenta que existe cierto solapamiento entre el registro de servicio y el patrón API Gateway, lo que también ayuda a resolver este problema. Por ejemplo, el Service Fabric Reverse Proxy es una implementación de API Gateway que se basa en el Service Fabric Naming Service y que ayuda a resolver la dirección a los servicios internos.

Recursos adicionales

- **Chris Richardson. Pattern: Service registry**
  http://microservices.io/patterns/service-registry.html,xu
- **Auth0. The Service Registry**
  https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/
- **Gabriel Schenker. Service discovery**
  https://lostechies.com/gabrielschenker/2016/01/27/service-discovery/

Creando una interfaz de usuario compuesta basada en microservicios

La arquitectura de microservicios a menudo comienza con el manejo de los datos y la lógica del lado del servidor. Sin embargo, un enfoque más avanzado es diseñar la interfaz de usuario (UI) basada también en microservicios. Eso significa tener una interfaz de usuario compuesta producida por los microservicios, en lugar de tener microservicios en el servidor y una aplicación cliente monolítica que consuma los microservicios. Con este enfoque, se pueden construir microservicios completos, que incluyan tanto la lógica como la presentación visual.

La figura 4-20 muestra el enfoque más sencillo, que consiste simplemente en consumir microservicios desde una aplicación cliente monolítica. Por supuesto, podría tener un servicio ASP.NET MVC de por medio, generando el HTML y JavaScript. La figura es una simplificación donde se destaca que tiene una interfaz de usuario única y monolítica, que consume los microservicios, que sólo se enfocan en la lógica y los datos y no en la interfaz de usuario (HTML y JavaScript).

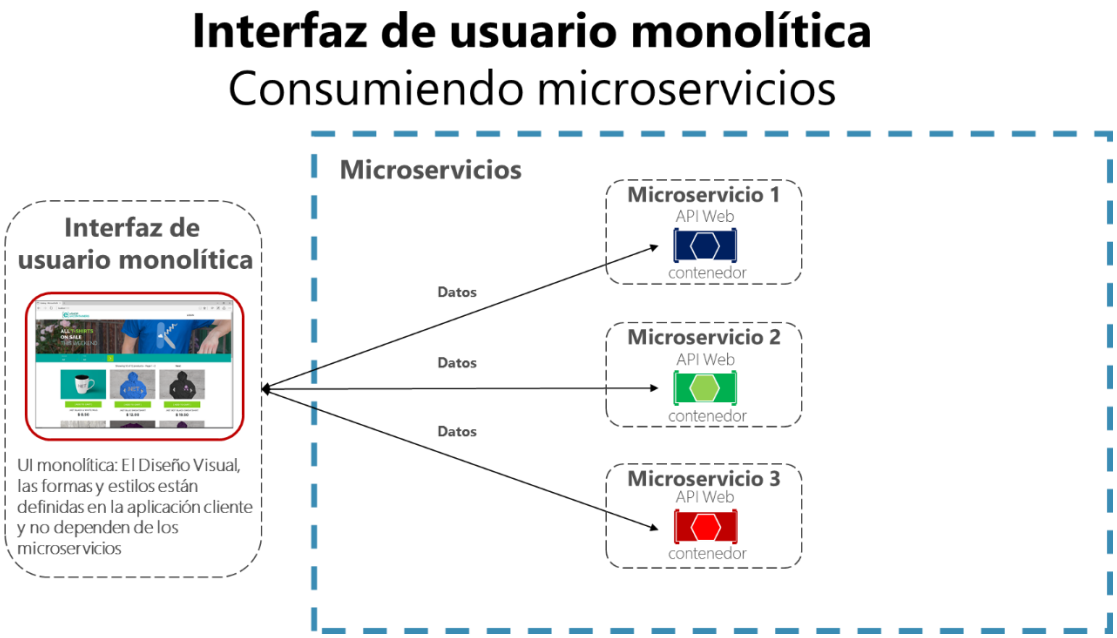


***Figura 4-20**. Una interfaz de usuario monolítica consumiendo microservicios del back-end*

En contraste, una interfaz de usuario compuesta es, precisamente, generada y compuesta por los mismos microservicios. Algunos de los microservicios controlan el aspecto visual de áreas específicas de la interfaz de usuario. La diferencia clave es que tiene componentes de interfaz de usuario de cliente (por ejemplo, clases de TypeScript) basados en plantillas y el *ViewModel* para esas plantillas proviene de cada microservicio.

Cuando arranca la aplicación cliente, cada uno de los componentes de la interfaz de usuario (clases de TypeScript, por ejemplo) se registran con un microservicio de infraestructura capaz de proporcionar los *ViewModels* para un escenario determinado. Si el microservicio cambia la presentación, la interfaz de usuario cambia también.

La Figura 4-21 muestra una versión de este enfoque de interfaz de usuario compuesta. Esto es una vista simplificada, porque es posible que otros microservicios estén consolidando partes más

detalladas, dependiendo de las tecnologías que se estén usando, por ejemplo, si está creando una aplicación web tradicional (ASP.NET MVC) o una SPA (Single Page Application).

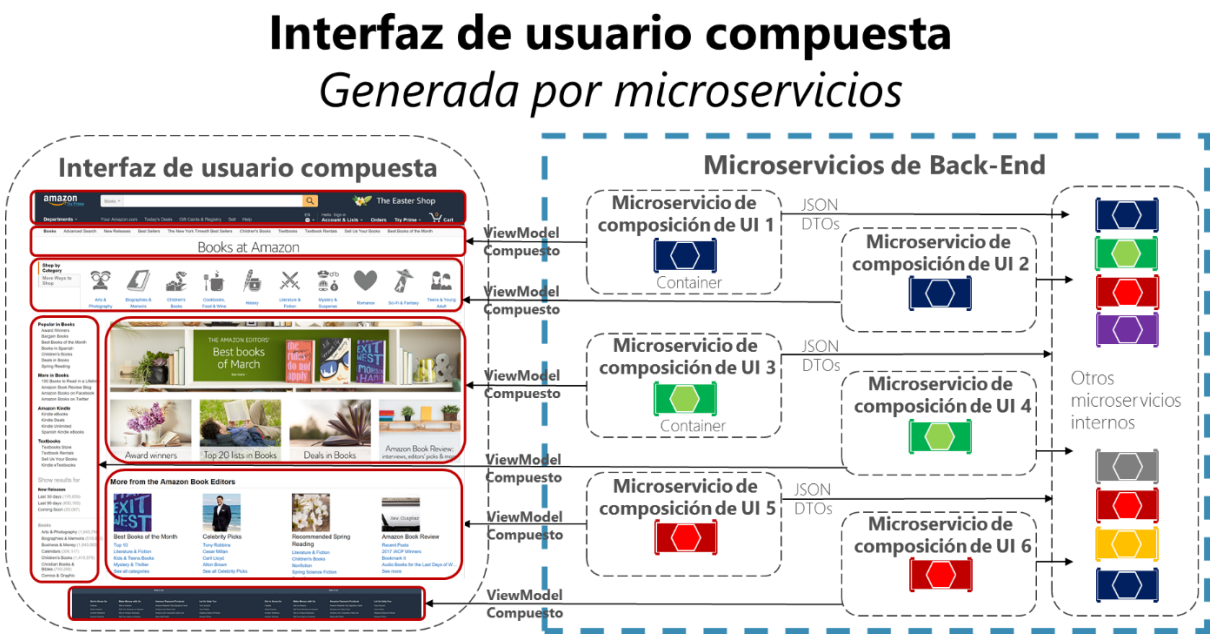


Figura 4-21. *Ejemplo de una UI compuesta definida por microservicios de back-end*

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

Un enfoque de interfaz de usuario compuesta, manejado por microservicios, puede ser más o menos desafiante, dependiendo de las técnicas de *front-end* que se utilicen. Por ejemplo, no usará las mismas técnicas para construir una aplicación web tradicional, para crear una SPA o una aplicación móvil nativa (como cuando se desarrollan aplicaciones Xamarin, que puede ser todavía más desafiante para este enfoque).

La aplicación de referencia eShopOnContainers utiliza el enfoque de interfaz de usuario monolítica por múltiples razones. Primero, es una introducción a microservicios y contenedores. Una interfaz de usuario compuesta no sólo es más avanzada, sino que también tiene una mayor complejidad al diseñarla y desarrollarla. En segundo lugar, eShopOnContainers también proporciona una aplicación móvil nativa basada en Xamarin, lo que la haría más compleja en el lado del cliente (en C#).

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

Recursos adicionales

- **Composite UI using ASP.NET (Particular's Workshop)**
  https://github.com/Particular/Workshop/tree/master/demos/asp-net-core
- **Ruben Oostinga. The Monolithic Frontend in the Microservices Architecture**
  http://blog.xebia.com/the-monolithic-frontend-in-the-microservices-architecture/
- **Mauro Servienti. The secret of better UI composition**
  https://particular.net/blog/secret-of-better-ui-composition

- **Viktor Farcic. Including Front-End Web Components Into Microservices**
  https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/
- **Managing Frontend in the Microservices Architecture**
  http://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

Resiliencia 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 se escribe involucra el manejo de excepciones y ahí es también donde se gasta más tiempo en las pruebas. El problema es más complicado que sólo escribir el código para manejar las fallas. ¿Qué sucede cuando la máquina donde se ejecuta el microservicio falla? No sólo se necesita detectar la falla del microservicio (un problema difícil en sí mismo), sino que también se necesita algo para reiniciarlo.

Un microservicio debe ser resiliente a las fallas y ser capaz de reiniciarse, a menudo en otra máquina por un tema de disponibilidad. Esta resiliencia también debe incluir el estado que se guardó en nombre del microservicio, desde dónde puede el microservicio recuperar este estado y si el microservicio se puede reiniciar con éxito. En otras palabras, es necesario que haya resiliencia en la capacidad de cómputo (el proceso debe poder reiniciarse en cualquier momento), así como también resiliencia en el estado o los datos, es decir manteniendo la consistencia e integridad de los datos.

Los problemas de resiliencia se combinan, y amplifican, en otros escenarios, como cuando ocurren fallas durante una actualización de la aplicación. El microservicio, trabajando con el sistema desplegado, necesita determinar si puede avanzar hacia la versión más nueva o, en su lugar, regresar a una versión anterior para mantener un estado consistente. Se deben tener en cuenta cuestiones como si hay suficientes máquinas disponibles para seguir avanzando y cómo recuperar versiones anteriores del microservicio. Esto requiere que el microservicio emita información de su estado de salud, para que la aplicación general y el orquestador puedan tomar estas decisiones.

Además, la resiliencia está relacionada con el comportamiento esperado de los sistemas basados en la nube. Como se mencionó anteriormente, un sistema basado en la nube debe manejar fallas y debe intentar recuperarse automáticamente de ellas. Por ejemplo, en caso de fallas en la red o en el contenedor, las aplicaciones o servicios cliente deben tener una estrategia para volver a intentar el envío de mensajes o peticiones, ya que en muchos casos las fallas en la nube son parciales. La sección Implementación de aplicaciones resilientes en esta guía, aborda cómo manejar fallas parciales. Describe técnicas como reintentos con retroceso exponencial o el patrón Interruptor Automático en .NET Core, mediante el uso de librerías como Polly, que ofrece una gran variedad de políticas para manejar este tema.

Gestión del funcionamiento y diagnósticos en microservicios

Puede parecer obvio, pero a menudo se pasa por alto, que un microservicio debe informar sobre su estado de salud y eventos de diagnóstico. De lo contrario, hay poca información desde una perspectiva de operaciones. La correlación de eventos de diagnóstico a través de un conjunto de servicios independientes y el manejo de las variaciones del reloj de la máquina, para dar sentido al orden de los eventos es todo un desafío. De la misma manera como se interactúa con un microservicio, en base a protocolos y formatos acordados, es necesario estandarizar cómo registrar eventos de diagnóstico y de salud, que finalmente terminarán en un almacén de eventos para consulta y visualización. En un enfoque de microservicios, es clave estandarizar en un formato único

de registro. Es necesario que haya un enfoque coherente para consultar los eventos de diagnóstico de la aplicación.

Control del estado de salud

El estado de salud es diferente de los diagnósticos. La salud se refiere a que el microservicio informe su estado actual para tomar las medidas adecuadas. Un buen ejemplo es cuando trabaja con los mecanismos de actualización y despliegue para mantener la disponibilidad. Aunque un servicio actualmente no esté saludable debido a un bloqueo del proceso o reinicio de la máquina, el servicio aún podría estar operativo y lo último que necesita es empeorar esto realizando una actualización. El mejor enfoque es hacer primero una investigación o esperar un tiempo para que se recupere el microservicio. Los eventos de salud de un microservicio ayudan a tomar decisiones informadas y, de hecho, ayudan a crear servicios de capaces de recuperarse por sí mismos.

En la sección Implementando controles del estado de salud en servicios ASP.NET Core de esta guía, explicamos cómo usar la nueva librería ASP.NET HealthChecks (que probablemente se incluya en ASP.NET Core 2.1) en sus microservicios, para que puedan informar su estado a un servicio de monitorización para tomar las medidas adecuadas cuando sea necesario.

Utilizando los *streams* de diagnósticos y log de eventos

Los logs de eventos proporcionan información sobre cómo se está ejecutando una aplicación o servicio, incluidas excepciones, advertencias y mensajes informativos simples. Por lo general, cada registro está en formato de texto con una línea por evento, aunque las excepciones también muestran a menudo el seguimiento de la pila en varias líneas.

En aplicaciones monolíticas basadas en servidor, simplemente se pueden escribir registros en un fichero en el disco y luego analizarlo con cualquier herramienta. Como la ejecución de la aplicación está limitada a un servidor o máquina virtual únicos, generalmente no es demasiado complejo analizar el flujo de eventos. Sin embargo, en una aplicación distribuida en la que se ejecutan múltiples servicios en muchos nodos de un *cluster*, poder correlacionar los eventos distribuidos es todo un desafío.

Una aplicación basada en microservicios no debe tratar de almacenar la secuencia de salida de eventos o ficheros de registro por sí mismo, ni siquiera tratar de administrar el enrutamiento de los eventos a un lugar central. Esto debe ser transparente, lo que significa que cada proceso debe simplemente escribir su flujo de eventos a una salida estándar, que será recolectado por la infraestructura del entorno donde se está ejecutando. Un ejemplo de estos enrutadores de flujos de eventos es Microsoft.Diagnostic.EventFlow, que recopila los flujos de eventos de múltiples fuentes y las publica en sistemas de salida. Estos pueden incluir salidas estándar simples para un entorno de desarrollo o sistemas en la nube como Application Insights, OMS (para aplicaciones *on-premises*) y Azure Diagnostics. También hay buenas plataformas y herramientas de análisis de registros de terceros que pueden buscar, alertar, monitorizar los registros e informar, incluso en tiempo real, como Splunk.

Orquestadores para manejar información de salud y diagnósticos

Cuando se crea una aplicación basada en microservicios, es necesario manejar la complejidad. Por supuesto, que un microservicio único es fácil de manejar, pero docenas o cientos de tipos y miles de instancias de microservicios representan un problema complejo. No se trata sólo de construir una arquitectura de microservicio; también se necesita alta disponibilidad, capacidad de direccionamiento,

resiliencia, control del estado de salud y diagnósticos si se espera mantener un sistema estable y consistente.

***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 muy difíciles de resolver por uno mismo. Los equipos de desarrollo deben enfocarse en resolver problemas del negocio y crear aplicaciones personalizadas con enfoques basados en microservicios. No deberían enfocarse en resolver problemas complejos de infraestructura. Si lo hicieran, el costo de cualquier aplicación basada en microservicios sería enorme. Por eso, existen plataformas orientadas a microservicios, denominadas orquestadores o *clusters* de microservicio, que se enfocan en resolver los problemas difíciles, en cuanto a ejecutar un servicio y utilizar los recursos de infraestructura de manera eficiente. Esto reduce la complejidad de crear aplicaciones que utilizan un enfoque de microservicios.

Distintos orquestadores pueden sonar similares, pero los diagnósticos y controles de salud ofrecidos por cada uno de ellos difieren en características y estado de madurez, a veces, dependiendo de la plataforma del sistema operativo, como se explica en la siguiente sección.

Recursos adicionales

- **The Twelve-Factor App. XI. Logs: Treat logs as event streams**
  https://12factor.net/logs
- **Microsoft Diagnostic EventFlow Library.** GitHub repo.
  https://github.com/Azure/diagnostics-eventflow
- **¿Qué es Diagnósticos de Azure?**
  https://docs.microsoft.com/azure/azure-diagnostics
- **Connect Windows computers to the Log Analytics service in Azure**
  https://docs.microsoft.com/azure/log-analytics/log-analytics-windows-agents
- **Logging What You Mean: Using the Semantic Logging Application Block**
  https://msdn.microsoft.com/library/dn440729(v=pandp.60).aspx
- **Splunk.** Official site.
  http://www.splunk.com
- **EventSource Class**. API for events tracing for Windows (ETW)
  https://msdn.microsoft.com/library/system.diagnostics.tracing.eventsource(v=vs.110).aspx

Orquestando microservicios y aplicaciones multi-contenedores, para alta escalabilidad y disponibilidad

Es esencial usar orquestadores en el entorno de producción, si las aplicaciones se basan en microservicios o simplemente se dividen en varios contenedores. Como se presentó anteriormente, en un enfoque basado en microservicios, cada microservicio posee su modelo y datos para que sea autónomo desde el punto de vista del desarrollo y despliegue. Pero incluso con una aplicación más tradicional que se compone de servicios múltiples, como SOA, también tendrá múltiples contenedores o servicios que componen la aplicación del negocio y que deben desplegarse como un sistema distribuido. Este tipo de sistemas son complejos de escalar y administrar, por lo tanto y sin lugar a dudas, es necesario un orquestador si desea tener una aplicación multi-contenedores y escalable en producción.

La Figura 4-23 muestra el despliegue en un *cluster* de una aplicación compuesta de múltiples microservicios en contenedores.

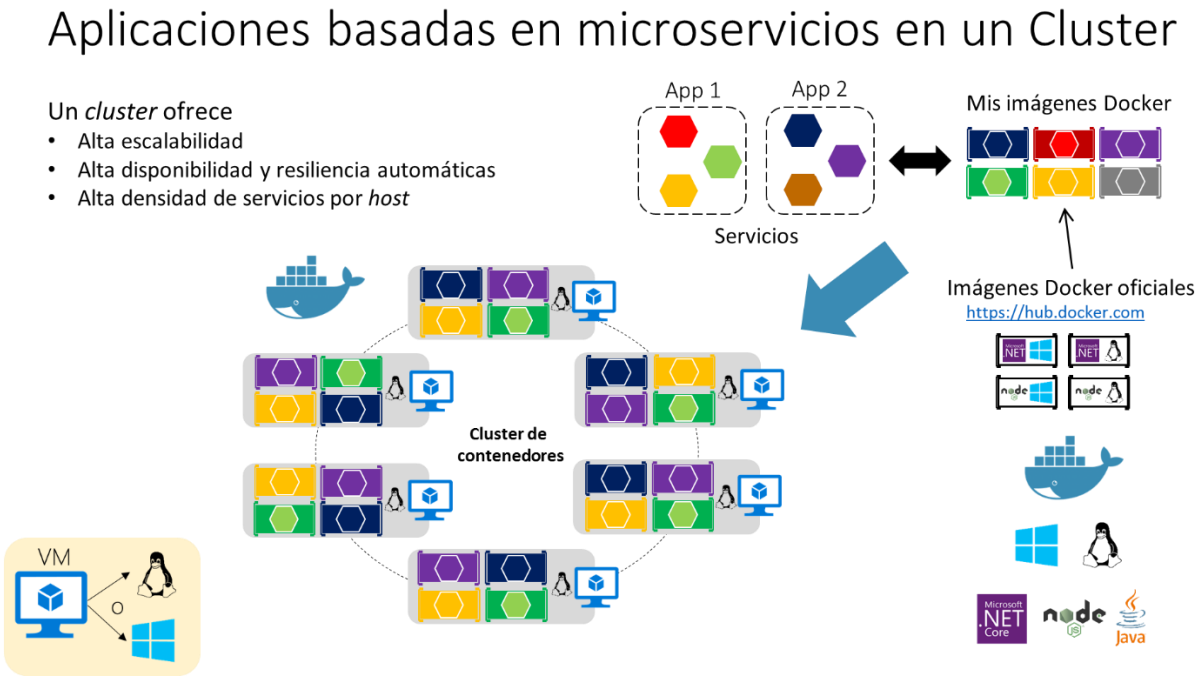


Figura 4-23. *Un cluster de contenedores*

Este parece un enfoque lógico. Pero, ¿cómo se maneja el balanceo de carga, el enrutamiento y la orquestación de estas aplicaciones compuestas?

El Docker Engine básico en *hosts* Docker individuales, cumple con las necesidades de gestión de instancias de una sola imagen en un *host*, pero esto no es suficiente cuando se trata de administrar varios contenedores implementados en múltiples hosts para aplicaciones más complejas y distribuidas. En la mayoría de los casos, se necesita una plataforma de administración que inicie automáticamente los contenedores, los escale creando varias instancias de una imagen y los

suspenda o apague cuando sea necesario e incluso, 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 muy simples y avanzar hacia aplicaciones empresariales más grandes con microservicios, es necesario usar plataformas de orquestación y *clustering*.

Desde el punto de vista de la arquitectura y el desarrollo, si se están creando aplicaciones empresariales grandes, compuestas por aplicaciones basadas en microservicios, es importante comprender las siguientes plataformas y productos que soportan escenarios avanzados:

***Clusters* y orquestadores**. Cuando se necesitan escalar aplicaciones en muchos *hosts* Docker, como cuando se trata de una aplicación grande basada en microservicios, es crítico poder administrar todos esos *hosts* como un *cluster* único, abstrayéndose la complejidad de la plataforma subyacente. Eso es lo que proporcionan los orquestadores y *clusters* de contenedores. Algunos ejemplos de orquestadores son Azure Service Fabric, Kubernetes, Docker Swarm y Mesosphere DC/OS. Los últimos tres orquestadores, *open source*, están disponibles en Azure a través del Azure Container Service.

Programadores. La programación significa tener la capacidad de que un administrador inicie contenedores en un *cluster*, así que los programadores también proporcionan una interfaz de usuario. Un programador de *cluster* tiene varias responsabilidades: usar los recursos del *cluster* de manera eficiente, establecer las restricciones indicadas por el usuario, balancear la carga de contenedores de manera eficiente entre los nodos o *hosts*, ser robusto contra errores y, a su vez, proporcionar alta disponibilidad.

Los conceptos de *cluster* y programador están estrechamente relacionados, por lo que los productos ofrecidos por diferentes proveedores suelen proporcionar ambas capacidades. La siguiente lista muestra las opciones de plataforma y software más importantes, para *clusters* y programadores. Estos orquestadores generalmente se ofrecen en nubes públicas como Azure.

Plataformas de software para *clustering*, orquestación y programación de contenedores	
Kubernetes 	Kubernetes es un producto *open source* que ofrece una funcionalidad que abarca desde la infraestructura del *cluster* y la programación de contenedores hasta las capacidades de orquestación. Permite automatizar el despliegue, el escalado y las operaciones de contenedores de aplicaciones en *clusters* de *hosts*. Kubernetes proporciona una infraestructura centrada en contenedores que los agrupa en unidades lógicas, para facilitar la administración y el descubrimiento. Kubernetes es un producto maduro en Linux, pero lo es menos en Windows.

Plataformas de software para *clustering*, orquestación y programación de contenedores	
Docker Swarm 	Docker Swarm maneja el *clustering* y programación de contenedores Docker. Al usar Swarm, puede convertir un grupo de *hosts* Docker en un *host* Docker virtual único. Los clientes pueden hacer peticiones de API a Swarm de la misma forma que lo hacen a los *hosts*, lo que significa que Swarm facilita el escalado de las aplicaciones a múltiples *hosts*. Docker Swarm es un producto de Docker, la compañía. Docker v1.12 o posterior puede ejecutar el modo nativo y modo Swarm incorporado.
Mesosphere DC/OS 	Mesosphere Enterprise DC/OS (basado en Apache Mesos) es una plataforma para ejecutar contenedores y aplicaciones distribuidas en entornos de producción. DC/OS es un producto maduro en Linux, pero menos maduro en Windows. DC/OS funciona al abstraer una colección de los recursos disponibles en el *cluster* y ponerlos a disposición de los componentes que se encuentran en la plataforma. Marathon se usa generalmente como un programador integrado con DC/OS.
Azure Service Fabric 	Service Fabric es una plataforma de microservicios de Microsoft para crear aplicaciones. Es un orquestador de servicios y crea *clusters* de equipos. Service Fabric puede desplegar servicios como contenedores o como procesos simples. Incluso puede mezclar servicios en procesos con servicios en contenedores dentro de la misma aplicación y *cluster*. Service Fabric proporciona varios modelos de programación prescriptivos, adicionales y opcionales, tales como Reliable Services y Reliable Actors. Service Fabric es un producto maduro en Windows (tiene años evolucionando en Windows), pero menos maduro en Linux. Tanto los contenedores de Linux como los de Windows son compatibles con Service Fabric desde 2017.

Usando orquestadores de contenedores en Microsoft Azure

Varios proveedores de servicios en la nube ofrecen soporte para orquestación de contenedores y *clusters* Docker, incluidos Microsoft Azure, Amazon EC2 Container Service y Google Container Engine. Microsoft Azure ofrece soporte para orquestación y *clusters* Docker a través del Azure Container Service (ACS), como se explica en la sección siguiente.

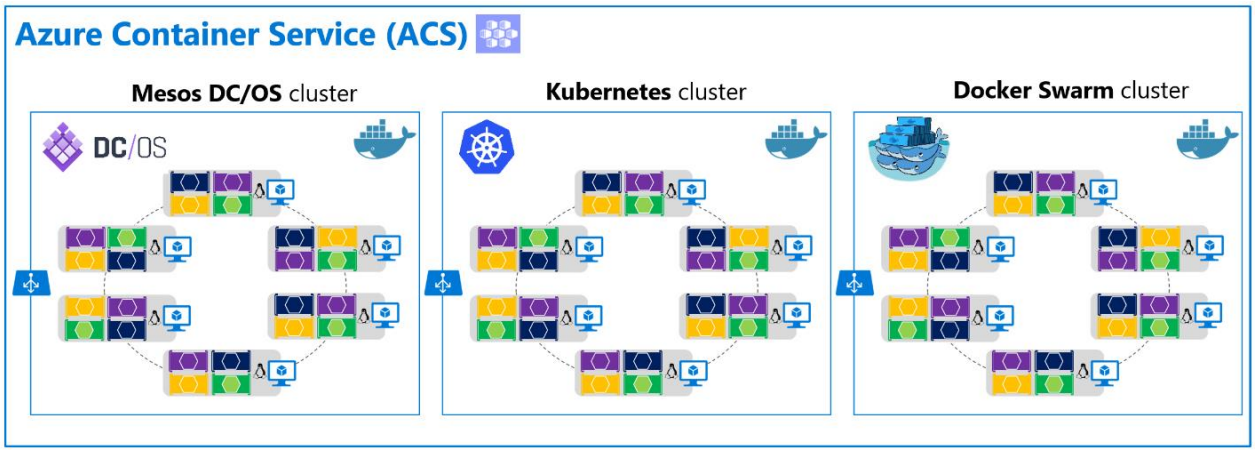
Otra opción es utilizar Microsoft Azure Service Fabric (una plataforma de microservicios), que también es compatible con Docker en Linux y contenedores Windows. Service Fabric se ejecuta en Azure o en cualquier otra nube y también se puede ejecutar en las instalaciones internas (*on-premises*).

Usando el Azure Container Service

Un *cluster* Docker agrupa varios *hosts* Docker y los expone como un *host* Docker virtual único, para poder desplegar varios contenedores. El *cluster* manejará toda la fontanería compleja de gestión, como la escalabilidad, estado de salud, etc. La Figura 4-23 muestra cómo se asigna un *cluster* Docker para aplicaciones compuestas a Azure Container Service (ACS).

ACS proporciona una forma de simplificar la creación, configuración y administración de un *cluster* de máquinas virtuales que están preconfiguradas para ejecutar aplicaciones contenerizadas. Con una configuración optimizada de las herramientas de programación y orquestación populares *open source*, ACS permite utilizar las habilidades existentes o recurrir a un cuerpo de expertos de la comunidad, cada vez más grande, para implementar y gestionar aplicaciones basadas en contenedores en Microsoft Azure.

Azure Container Service optimiza la configuración de las herramientas y tecnologías populares *open source*, para *clusters* Docker, específicamente para Azure. Se puede obtener una solución abierta que ofrece portabilidad tanto para sus contenedores como para la configuración de las aplicaciones. Basta con seleccionar el tamaño, la cantidad de hosts y las herramientas del orquestador y el ACS maneja todo lo demás.



***Figur 4-24**. Opciones para manejar clusters en Azure Container Service*

El ACS aprovecha las imágenes de Docker para garantizar que los contenedores de su aplicación sean totalmente portátiles. Permite la elección de plataformas de orquestación *open source* como DC/OS (con Apache Mesos), Kubernetes (creado originalmente por Google) y Docker Swarm, para garantizar que estas aplicaciones se puedan escalar a miles o incluso a decenas de miles de contenedores.

El servicio Azure Container le permite aprovechar las características de nivel empresarial de Azure, manteniendo la portabilidad de las aplicaciones, incluso a nivel de las capas de orquestación.

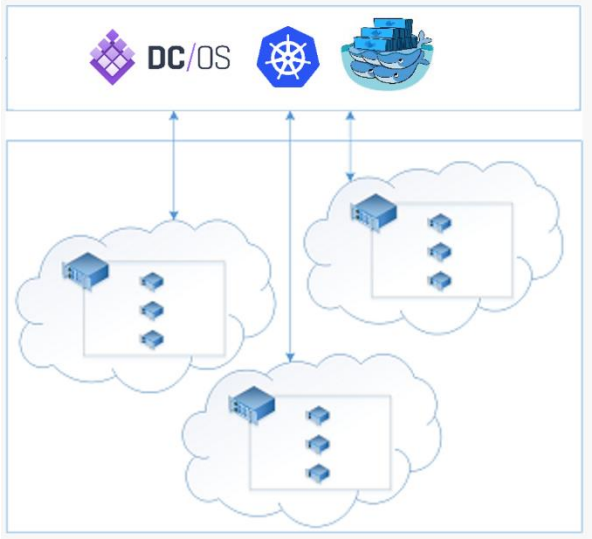


Figura 4-25. *Orquestadores en ACS*

Como se muestra en la Figura 4-25, Azure Container Service es simplemente la infraestructura provista por Azure para implementar DC/OS, Kubernetes o Docker Swarm, pero ACS no implementa ningún orquestador adicional. Por lo tanto, ACS no es un orquestador como tal, sino sólo una infraestructura que aprovecha los orquestadores *open source* existentes.

El objetivo de Azure Container Service, desde un punto de vista de uso, es proporcionar un entorno de alojamiento de contenedores usando herramientas y tecnologías populares *open source*. Con este fin, expone los *endpoints* estándar del API para el orquestador seleccionado. Al usar estos *endpoints*, se puede aprovechar cualquier software que pueda comunicarse con ellos. Por ejemplo, en el caso de Docker Swarm, puede optar por usa la interfaz de línea de comandos Docker (CLI) y para DC/OS, también puede usar su CLI.

Arrancando con el Azure Container Service

Para comenzar a usar Azure Container Service, se debe desplegar un *cluster* de Azure Container Service desde el Azure Portal, usando una plantilla de Azure Resource Manager o la CLI. Las plantillas disponibles incluyen Docker Swarm, Kubernetes, y DC/OS. Las plantillas de inicio rápido se pueden modificar para incluir una configuración adicional o avanzada de Azure. Para obtener más información sobre el despliegue de un *cluster* de Azure Container Service, vea Implementar un *cluster* de Azure Container Service.

No se incurre en costes adicionales por el software instalado por defecto como parte de ACS. Todas las opciones predeterminadas se implementan con software *open source*.

Actualmente, ACS está disponible para las máquinas virtuales Linux de las series A, D, DS, G y GS en Azure. Sólo se le cobrará por las instancias de cómputo que elija, así como también por los demás recursos de infraestructura subyacentes consumidos, como almacenamiento y redes. No hay cargos adicionales para ACS.

Actualmente Azure Container Services se está orientando principalmente hacia Kubernetes y se maneja como AKS. ACS, que todavía está disponible, maneja Docker Swarm and DC/OC, además de Kubernetes, como ya se ha mencionado.

Recursos adicionales

- **Introduction to Docker container hosting solutions with Azure Container Service**
  https://azure.microsoft.com/documentation/articles/container-service-intro/
- **Docker Swarm overview**
  https://docs.docker.com/swarm/overview/
- **Swarm mode overview**
  https://docs.docker.com/engine/swarm/
- **Mesosphere DC/OS Overview**
  https://docs.mesosphere.com/1.10/overview/
- **Kubernetes.** The official site.
  http://kubernetes.io/

Usando Azure Service Fabric

El Azure Service Fabric surgió de la transición de Microsoft, de entregar productos de caja, que por lo general eran de estilo monolítico, a la entrega de servicios. La experiencia de crear y operar servicios a grandes escalas, como Azure SQL Database, Azure Cosmos DB, Azure Service Bus o el *back-end* de Cortana, le dio forma a Service Fabric. La plataforma evolucionó con el tiempo a medida que más y más servicios la adoptaron. Es importante destacar que Service Fabric se tuvo que ejecutar no sólo en Azure sino también en despliegues independientes de Windows Server.

El objetivo de Service Fabric es resolver los problemas difíciles que surgen al construir y ejecutar servicios y usar los recursos de infraestructura de forma eficiente, de modo que los equipos de trabajo puedan resolver los problemas del negocio usando un enfoque de microservicios.

Service Fabric cubre dos aspectos generales, para facilitar la creación de aplicaciones que utilizan un enfoque de microservicios:

- Una plataforma que ofrece servicios del sistema para desplegar, escalar, actualizar, detectar y reiniciar servicios fallidos, descubrir la ubicación de los servicios, administrar el estado y supervisar el estado de salud. Estos servicios del sistema, en efecto, permiten muchas de las características descritas anteriormente de los microservicios.
- Las APIs o *frameworks* de programación, para facilitar la creación de aplicaciones como microservicios: Reliable Actors y Reliable Services. Por supuesto, se puede usar cualquier código para construir un microservicio, pero estas API hacen que el trabajo sea más sencillo y se integran con la plataforma a un nivel más profundo. De esta manera puede obtener información de diagnóstico y del estado de salud o puede aprovechar la administración confiable del estado.

El Service Fabric es independiente de cómo crea un servicio y puede usar cualquier tecnología. Sin embargo, proporciona APIs de programación integradas que facilitan la creación de microservicios.

Como se muestra en la Figura 4-26, se pueden crear y ejecutar microservicios en Service Fabric, ya sea con procesos simples o con contenedores Docker. También es posible mezclar microservicios basados en contenedores con microservicios basados en procesos dentro del mismo *cluster* Service Fabric.

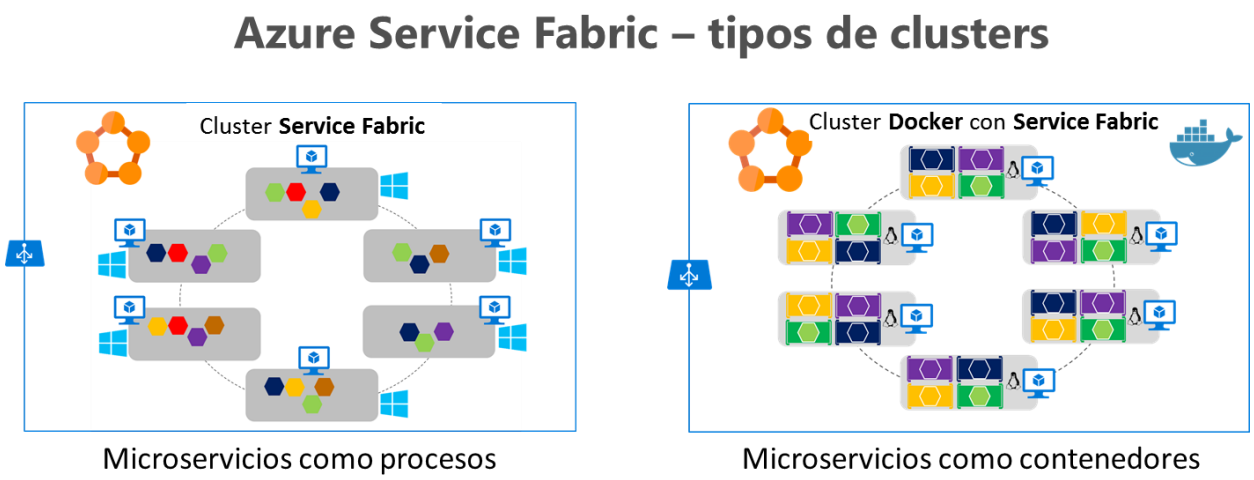


***Figura 4-26**. Desplegando microservicios como procesos o como contenedores en Azure Service Fabric*

Los *cluster*s de Service Fabric basados en hosts Linux y Windows pueden ejecutar contenedores Docker Linux y contenedores de Windows, respectivamente.

Para obtener información actualizada sobre el soporte de contenedores en Azure Service Fabric, consulte Service Fabric and containers.

Service Fabric es un buen ejemplo de una plataforma en la que puede definir una arquitectura lógica (microservicios del negocio o *Bounded Contexts*) diferente de la implementación física, como se presentó en la sección Arquitectura lógica frente a arquitectura física. Por ejemplo, si implementan Reliable Services en Azure Service Fabric, que se presentan en la sección Microservicios sin estado vs microservicios con estado más adelante, se un microservicio conceptual del negocio con múltiples servicios físicos.

Como se muestra en la Figura 4-27 y viéndolo desde una perspectiva de microservicio lógico o del negocio, cuando se implementa un Service Fabric Stateful Reliable Service, generalmente se deben implementar dos niveles de servicios. El primero es el servicio de *back-end* confiable con estado, que maneja múltiples particiones (cada partición es un servicio con estado). El segundo es el servicio de *front-end*, o servicio de *gateway*, que se encarga del enrutamiento y la consolidación de datos entre múltiples particiones o instancias de servicios con estado. Ese servicio de *gateway* también maneja la comunicación del lado del cliente, con bucles de reintento que acceden al servicio de *back-end*. Se llama un servicio de *gateway* si implementa un servicio personalizado, por otro lado, puede usar el proxy inverso incluido en Service Fabric.

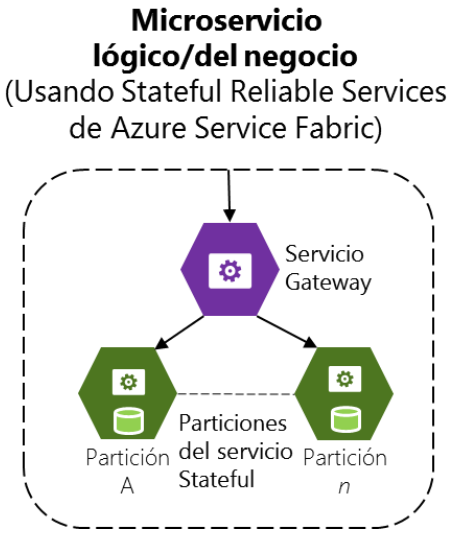


***Figura 4-27**. Microservicio del negocio con varias instancias del servicio stateful y un gateway personalizado como front-end*

En cualquier caso, cuando utilizan los Service Fabric Stateful Reliable Services, también se tiene un microservicio lógico o del negocio (*Boundex Context*) que generalmente está compuesto de múltiples servicios físicos. Cada uno de ellos, así como el servicio de *gateway* y el de partición se podrían implementar como servicios ASP.NET Web API, tal como se muestra en la figura 4-27.

En Service Fabric, puede agrupar y desplegar servicios como una Service Fabric Application, que es la unidad de empaquetado y despliegue para el orquestador o *cluster*. Por lo tanto, la Service Fabric Application se podría asignar a este *Bounded Context* o microservicio de negocio autónomo, para desplegar estos servicios de forma autónoma.

Los contenedores y Service Fabric

También se pueden desplegar servicios en imágenes de contenedor dentro de un *cluster* de Service Fabric. Como se muestra en la Figura 4-28, la mayoría de las veces solo habrá un contenedor por servicio.

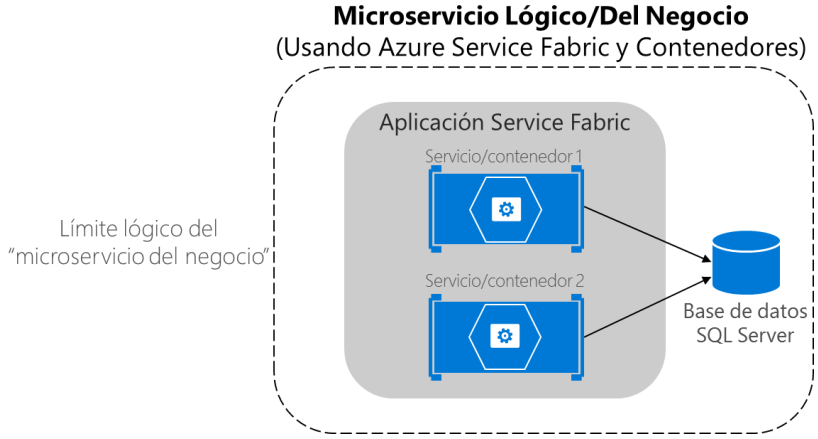


***Figure 4-28**. Microservicio del negocio con varios servicios (contenedores) en Service Fabric*

Sin embargo, los llamados contenedores "sidecar" (dos contenedores que se deben desplegar juntos como parte de un servicio lógico) también se pueden manejar en Service Fabric. Lo importante es que un microservicio del negocio sea el límite lógico alrededor de varios elementos con alta cohesión. En muchos casos, podría tratarse de un servicio único con un modelo de datos también único, pero también podría tener varios servicios físicos.

No se pueden desplegar Service Fabric Reliable Stateful Services en contenedores (al menos hasta mediados de 2017), sólo puede desplegar servicios sin estado y servicios de actores (*actor services*) en contenedores. Sin embargo, hay que tener presente que sí se pueden mezclar servicios en procesos simples con servicios en contenedores en la Service Fabric Application, como se muestra en la Figura 4-29.

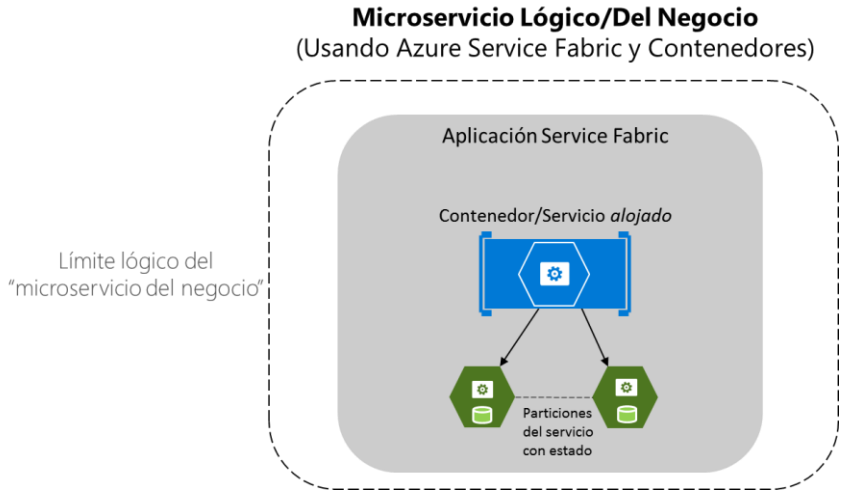


Figura 4-29. *Microservicio del negocio mapeado a una aplicación Service Fabric con contenedores y servicios con estado*

Para obtener más información sobre el soporte de contenedores en Azure Service Fabric, consulte Service Fabric and containers.

Microservicios sin estado (*stateless*) versus microservicios con estado (*stateful*)

Como se mencionó anteriormente, cada microservicio (*Bounded Context* lógico) debe poseer su modelo de dominio (datos y lógica). En el caso de microservicios sin estado, las bases de datos serán externas, empleando bases de datos relacionales como SQL Server o no relacionales como NoSQL, MongoDB o Azure Cosmos DB.

Pero los mismos servicios también pueden tener estado en Service Fabric, lo que significa que hay datos se encuentran dentro del microservicio. Esta información podría existir no solo en el mismo servidor, sino dentro del proceso de microservicio, en la memoria y persistido en discos duros y replicado en otros nodos. La figura 4-30 muestra los diferentes enfoques.

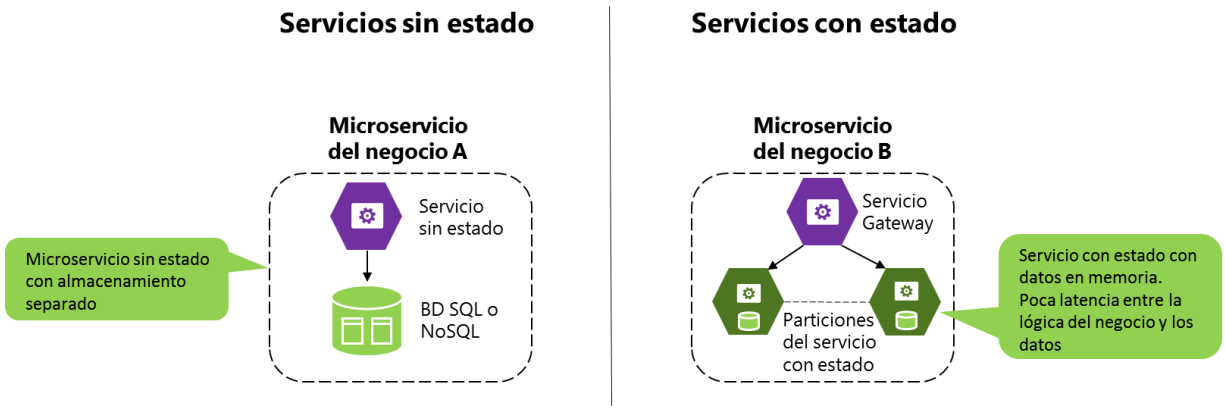


***Figura 4-30**. Microservicios con estado versus sin estado*

Un enfoque sin estado es perfectamente válido y es más fácil de implementar que los microservicios con estado, ya que el enfoque es similar a los patrones conocidos tradicionales. Pero los microservicios sin estado implican una latencia entre el proceso y las fuentes de datos. También están involucradas más partes móviles cuando intenta mejorar el rendimiento con caché y colas adicionales. El resultado es que se puede terminar con arquitecturas complejas con demasiados niveles.

Por el contrario, los Reliable Services pueden sobresalir en escenarios avanzados, porque no hay latencia entre la lógica del dominio y los datos. El procesamiento intensivo de datos, los *back-ends* de juego, las bases de datos como servicio y otros escenarios de baja latencia se benefician de los servicios con estado, que habilitan el manejo de un estado local para lograr un acceso más rápido.

Los servicios sin estado y con estado son complementarios. Por ejemplo, se puede ver en el diagrama de la derecha, en la Figura 4-30, que un servicio con estado podría dividirse en múltiples particiones. Para acceder a esas particiones, se podría usar un servicio sin estado que actúe como un *gateway* que sepa cómo abordar cada partición en función de las claves de la partición.

Sin embargo, los servicios con estado tienen inconvenientes. Se requiere manejar un nivel de complejidad mayor para que se puedan escalar. La funcionalidad que generalmente sería implementada por sistemas de bases de datos externos, se debe manejar en el servicio para tareas tales como la replicación de datos y el particionamiento de datos. Sin embargo, esta es una de las áreas en las que un orquestador como Azure Service Fabric con sus stateful reliable services puede ayudar más, al simplificar el desarrollo y el ciclo de vida de los microservicios con estado que utilizan la Reliable Services API y los Reliable Actors.

Otros *frameworks* de microservicios que permiten servicios con estado, soportan el patrón Actor y que mejoran la tolerancia a errores y la latencia entre la lógica del negocio y los datos son Microsoft Orleans de Microsoft Research y Akka.NET. Ambos *frameworks* están mejorando actualmente su soporte para Docker.

Tenga en cuenta que los contenedores Docker no tienen estado. Si desea implementar un servicio con estado, necesita uno de los *framework* prescriptivos y de nivel superior mencionados anteriormente.

Proceso de desarrollo de aplicaciones basadas en Docker

Visión

Desarrollar aplicaciones .NET contenerizadas de la manera que prefiera, ya sea orientado hacia un IDE con Visual Studio y herramientas de Visual Studio para Docker u orientado a CLI/Editor con Docker CLI y Visual Studio Code.

Entorno de desarrollo para aplicaciones Docker

Opciones de la herramienta de desarrollo: IDE o editor

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

Visual Studio (para Windows). Para desarrollar aplicaciones basadas en Docker con Visual Studio, se recomienda usar Visual Studio 2017 o versiones posteriores, que ya vienen con herramientas integradas para Docker. Las herramientas para Docker le permiten desarrollar, ejecutar y validar las aplicaciones directamente en el entorno Docker de destino. Se puede presionar F5 para ejecutar y depurar la aplicación (contenedor único o contenedores múltiples) directamente en un *host* Docker, o presione CTRL + F5 para editar y actualizar su aplicación sin tener que reconstruir el contenedor. Esta es la opción de desarrollo más poderosa para las aplicaciones basadas en Docker.

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

Visual Studio Code y CLI de Docker. Para trabajar con un editor liviano y multiplataforma, está Visual Studio Code (VS Code), que permita cualquier lenguaje de desarrollo y la CLI de Docker. Este es un enfoque de desarrollo multiplataforma para Mac, Linux y Windows.

Al instalar la herramienta Docker Community Edition (CE), se puede usar la CLI de Docker para crear aplicaciones tanto para Windows como para Linux.

Recursos adicionales

- **Visual Studio Tools para Docker**
  https://docs.microsoft.com/aspnet/core/publishing/visual-studio-tools-for-docker
- **Visual Studio Code**. Sitio oficial de descargas.
  https://code.visualstudio.com/download
- **Docker Community Edition (CE) for Mac and Windows**
  https://www.docker.com/community-editions

Lenguajes y *frameworks* .NET para contenedores Docker

Como se mencionó en secciones anteriores, se puede usar .NET Framework, .NET Core o el proyecto Mono *open source* para desarrollar aplicaciones .NET en contenedores Docker. Se Puede desarrollar en C#, F# o Visual Basic cuando se apunta a Contenedores Linux o Windows, dependiendo de cuál framework.NET Framework se utilice. Para obtener más detalles sobre los lenguajes .NET, consulte el artículo de blog The .NET Language Strategy.

Flujo de trabajo para el desarrollo de aplicaciones Docker

El ciclo de vida de desarrollo de la aplicación comienza en la máquina de cada desarrollador, donde se codifica la aplicación con el lenguaje preferido y se prueba localmente. Independientemente del lenguaje, el framework y la plataforma elegida, con este flujo de trabajo, el desarrollador siempre desarrolla y prueba contenedores Docker, pero lo hace de forma local.

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

- El sistema operativo seleccionado (por ejemplo, una distribución de Linux, Windows Nano Server o Windows Server Core).
- Los ficheros agregados por el desarrollador (binarios de la aplicación, etc.).
- Información de configuración (configuraciones del entorno y dependencias).

Flujo de trabajo para el desarrollo de aplicaciones basadas en contenedores Docker

Esta sección describe el flujo de trabajo del *bucle interno* para el desarrollo de aplicaciones basadas en contenedores Docker. El *bucle interno* significa que no tiene en cuenta el flujo de trabajo de DevOps más amplio (que puede incluir hasta el despliegue en producción) y sólo se centra en el trabajo de desarrollo realizado en la máquina del desarrollador. No se incluyen los pasos iniciales para configurar el entorno, ya que se realizan sólo una vez.

Una aplicación se compone de los servicios propios más las librerías adicionales (dependencias). Los siguientes son los pasos básicos que por lo general son necesarios para construir una aplicación Docker, como se ilustra en la Figura 5-1.

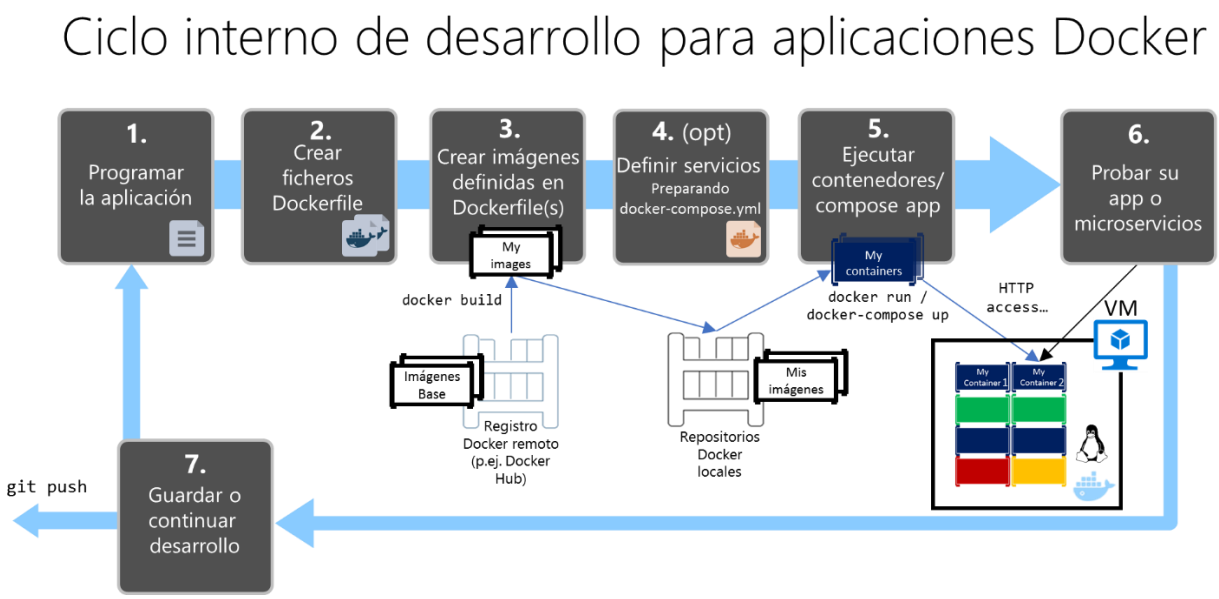


Figura 5-1. *Flujo de trabajo, paso-a-paso para desarrollar aplicaciones Docker contenerizadas*

En esta guía se detalla el proceso y se explica cada paso importante, enfocándose en un entorno con Visual Studio.

Cuando se utiliza un entorno de desarrollo con editor/CLI (por ejemplo, Visual Studio Code más CLI de Docker en macOS o Windows), en general necesita conocer cada paso con más detalle que si se usa Visual Studio. Para obtener más información sobre cómo trabajar en un entorno con CLI, consulte el eBook Containerized Docker Application lifecycle with Microsoft Platforms and Tools.

Cuando se usa Visual Studio 2015 o Visual Studio 2017, muchos de esos pasos son manejados por el IDE, lo que mejora drásticamente la productividad. Esto es especialmente cierto cuando usa Visual Studio 2017 y se apunta a aplicaciones de múltiples contenedores. Por ejemplo, con sólo un clic del mouse, Visual Studio agrega los ficheros **`Dockerfile`** y **`docker-compose.yml`** a los proyectos con la configuración de la aplicación. Cuando se ejecuta la aplicación en Visual Studio, se crea la imagen Docker y se ejecuta directamente en Docker la aplicación multi-contenedores. Incluso permite depurar varios contenedores a la vez. Estas características aumentarán significativamente la velocidad de desarrollo.

Sin embargo, el que Visual Studio haga que esos pasos sean automáticos, no significa que no necesite saber qué está sucediendo debajo con Docker. Por esta razón, a continuación, detallamos cada paso.

Paso 1. Programar la versión inicial de la aplicación o servicio

Desarrollar una aplicación Docker es similar desarrollar cualquier otra aplicación. La diferencia es que, al desarrollar con Docker, se está desplegando y probando la aplicación o servicios dentro de contenedores Docker en su entorno local (ya sea una configuración de máquina virtual de Linux por Docker o directamente Windows, si usa contenedores de Windows).

Preparación de un entorno local con Visual Studio

Para comenzar, se debe tener instalado el Docker Community Edition (CE) for Windows, como se explica en las siguientes instrucciones:

> Get started with Docker CE for Windows

Además, debe estar instalado Visual Studio 2017. Esto es preferible a Visual Studio 2015 con el complemento Visual Studio Tools for Docker, porque Visual Studio 2017 tiene un soporte más avanzado para Docker, por ejemplo, para depuración en contenedores. Visual Studio 2017 incluye las herramientas para Docker cuando se selecciona la carga de trabajo de .NET Core multiplataforma durante la instalación, como se muestra en la Figura 5-2.

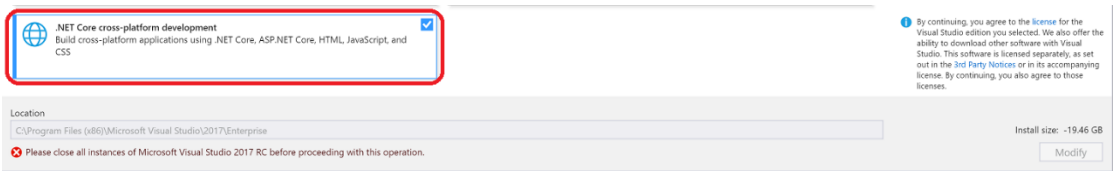


***Figura 5-2**. Seleccionado la carga de trabajo **.NET Core and Docker** al configurar Visual Studio 2017*

Puede comenzar a programar su aplicación en .NET (normalmente en .NET Core si planea usar contenedores) incluso antes de habilitar Docker en la aplicación para despliegue y pruebas. Sin embargo, se recomienda empezar a trabajar en Docker lo antes posible, porque ese será el entorno real y eso ayuda a descubrir cualquier problema lo antes posible. Visual Studio promueve esto, porque hace que trabajar con Docker sea tan fácil que resulte casi transparente, esto se puede apreciar en especial al depurar aplicaciones de varios contenedores desde Visual Studio.

Recursos adicionales

- **Get started with Docker CE for Windows**
  https://docs.docker.com/docker-for-windows/
- **Visual Studio 2017**
  https://www.visualstudio.com/vs/

Paso 2. Crear el fichero Dockerfile para una imagen base de .NET existente

Se necesita un **`Dockerfile`** para cada imagen personalizada que se deba construir, también necesita un **`Dockerfile`** para cada contenedor que se desplegará, ya sea que despliegue automáticamente desde Visual Studio o manualmente mediante la Docker CLI (comandos **`docker run`** y **`docker-compose`**). Si la aplicación contiene un servicio único personalizado, se necesita sólo un **`Dockerfile`**. Si la aplicación contiene múltiples servicios (como en una arquitectura de microservicios), se necesita un **`Dockerfile`** para cada servicio.

El fichero **`Dockerfile`** se coloca en la carpeta raíz de la aplicación o servicio. Contiene los comandos que le dicen a Docker cómo configurar y ejecutar la aplicación o servicio en un contenedor. Se puede

crear manualmente un fichero **`Dockerfile`** en un editor y agregarlo al proyecto junto con sus dependencias .NET.

Con Visual Studio y sus herramientas para Docker, esta tarea requiere sólo unos pocos clics del mouse. Cuando crea un nuevo proyecto en Visual Studio 2017, hay una opción llamada **Habilitar el soporte para Docker**, como se muestra en la Figura 5-3.

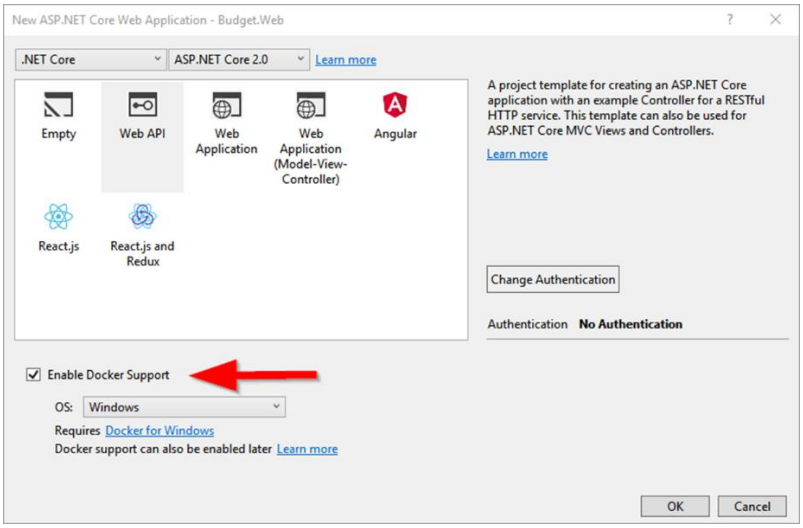


***Figura 5-3**. Habilitando soporte para Docker al crear un nuevo proyecto en Visual Studio 2017*

También puede habilitar el soporte Docker en un proyecto existente haciendo clic derecho en su fichero de proyecto en Visual Studio y seleccionando la opción **Add > Docker Support**, como se muestra en la Figura 5-4.

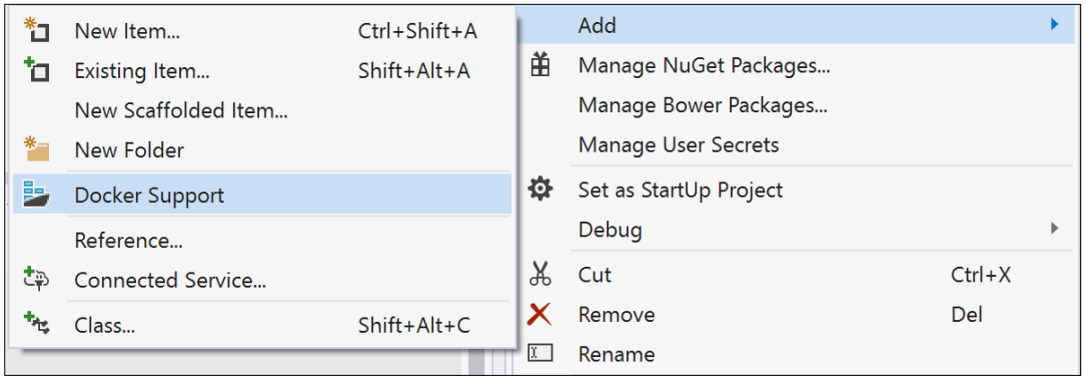


***Figure 5-4**. Habilitando soporte para Docker en un proyecto ASP.NET Core existente en Visual Studio 2017*

Al hacer esto en un proyecto (como una aplicación web ASP.NET o un servicio API web) se agrega el fichero **`Dockerfile`** al proyecto con la configuración necesaria. También se agrega un fichero **`docker-compose.yml`** para toda la solución. En las siguientes secciones, describimos la información que se incluye en cada uno de esos ficheros. Aunque Visual Studio se puede encargar de este trabajo, es útil entender qué entra en un **`Dockerfile`**.

Opción A: Creando un Proyecto con una imagen oficial Docker para .NET existente

Por lo general, se crea una imagen personalizada para su contenedor sobre una imagen base que puede obtener de un repositorio oficial en el registro de Docker Hub. Eso es precisamente lo que sucede por debajo cuando se habilita el soporte para Docker en Visual Studio. El fichero **`Dockerfile`** usará una imagen **`aspnetcore`** existente.

Anteriormente explicamos qué imágenes y repositorios Docker se pueden usar, según el framework y el sistema operativo seleccionado. Por ejemplo, para usar ASP.NET Core (Linux o Windows), la imagen a usar es **`microsoft/aspnetcore: 2.0`**. Por lo tanto, sólo necesita especificar cuál imagen base Docker usará para el contenedor. Para hacerlo, agregue **`FROM microsoft/aspnetcore: 2.0`** a su **`Dockerfile`**. Esto será realizado automáticamente por Visual Studio, pero si se tuviera que actualizar la versión, este valor que se cambia.

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

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

```
FROM microsoft/aspnetcore:2.0
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", " MySingleContainerWebApp.dll "]
```

En este caso, el contenedor se basa en la versión 2.0 de la imagen oficial de ASP.NET Core Docker (multi-arch para Linux y Windows). Esta es la configuración **`FROM microsoft/aspnetcore: 2.0`**. (Para obtener más detalles sobre esta imagen base, consulte las páginas ASP.NET Core Docker Image y .NET Core Docker Image.) En el fichero **`Dockerfile`**, también debe indicar a Docker el puerto TCP que se utilizará en tiempo de ejecución (en este caso, el puerto 80, como se indica con la configuración **`EXPOSE`**).

Se pueden especificar opciones de configuración adicionales en el fichero **`Dockerfile`**, según el lenguaje y el framework que esté utilizando. Por ejemplo, la línea **`ENTRYPOINT`** con **`["dotnet", "MySingleContainerWebApp.dll"]`** le dice a Docker que ejecute una aplicación .NET Core. Si se estuviese utilizando el SDK y la CLI .NET (dotnet CLI) para compilar y ejecutar la aplicación .NET, esta configuración sería diferente. La conclusión es que la línea **`ENTRYPOINT`** y otras opciones serán diferentes según el lenguaje y la plataforma que seleccionada para la aplicación.

Recursos adicionales

- **Building Docker Images for .NET Core Applications**
  https://docs.microsoft.com/dotnet/articles/core/docker/building-net-docker-images
- **Create a base image**. En la documentación oficial de Docker
  https://docs.docker.com/engine/userguide/eng-image/baseimages/

Usando repositorios de imágenes multi-arch

Un repositorio puede contener variantes por plataforma, como una imagen Linux y otra Windows. Esto permite a los proveedores como Microsoft (creadores de imágenes base) crear un solo

repositorio para cubrir múltiples plataformas (es decir, Linux y Windows). Por ejemplo, el repositorio microsoft/aspnetcore disponible en el registro de Docker Hub proporciona soporte para Linux y Windows Nano Server utilizando el mismo nombre de repositorio.

Se puede especificar una etiqueta que apunte a una plataforma explícita como en los siguientes casos:

`microsoft/aspnetcore:2.0.0-jessie`	.NET Core 2.0 runtime-only on Linux
`microsoft/dotnet: 2.0.0-nanoserver`	.NET Core 2.0 runtime-only on Windows Nano Server

Pero, y esto es nuevo desde mediados de 2017, si especifica el mismo nombre de imagen, incluso con la misma etiqueta, las nuevas imágenes multi-arch (como la imagen mencionada de aspnetcore) usarán la versión Linux o Windows dependiendo en el sistema operativo del *host* Docker donde se esté desplegando, como se muestra en el siguiente ejemplo:

`microsoft/aspnetcore:2.0`	.NET Core 2.0 runtime-only on Linux or Windows Nano Server depending on the Docker *host* OS

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

Opción B: Creando una imagen base desde cero

Se puede crear una imagen base Docker desde cero. Este escenario no se recomienda para alguien que está comenzando con Docker, pero si desea usar la versión específica de su propia imagen base, puede hacerlo.

Recursos adicionales

- **Multi-arch .NET Core images:** https://github.com/dotnet/announcements/issues/14
- **Create a base image**. Official Docker documentation. https://docs.docker.com/engine/userguide/eng-image/baseimages/

Paso 3.Crear imágenes Docker personalizadas e incluir la aplicación o servicio en ellas

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

Hay que tener en cuenta que las imágenes Docker se crean automáticamente en Visual Studio. Los siguientes pasos sólo son necesarios para el entorno de trabajo con editor/CLI y se explican para aclarar qué sucede por debajo.

Durante el desarrollo se necesita programar y probar localmente hasta que se guarde el cambio o característica completa en sistema de control de versiones (por ejemplo, a GitHub). Esto significa que se deben crear las imágenes de Docker e implementar contenedores en un *host* Docker local (VM de Windows o Linux) y ejecutar, probar y depurar contra esos contenedores locales.

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

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

***Figura 5-5**. Creando una imagen Docker personalizada*

Opcionalmente, en lugar de ejecutar directamente Docker Build desde la carpeta del proyecto, se puede generar primero una carpeta de despliegue con las librería y binarios .NET necesarios ejecutando **`dotnet publish`** y luego usar el comando **`docker build`**.

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. Se puede repetir este paso para cada imagen personalizada que se necesite crear para una aplicación Docker compuesta.

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

Se pueden encontrar las imágenes existentes en el repositorio local con el comando **`docker images`**, como se muestra en la Figura 5-6.

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

***Figura 5-6**. Consultando las imágenes existentes con el comando* `docker images`

Creando imágenes Docker con Visual Studio

Cuando se usa Visual Studio para crear un proyecto con soporte para Docker, no es necesario crear la imagen explícitamente. En cambio, la imagen se crea automáticamente al presionar F5 y se ejecuta la aplicación o servicio contenerizado. Este paso es automático en Visual Studio y no lo verá, pero es importante que sepa lo que está sucediendo por debajo.

Paso 4.Definir los servicios en docker-compose.yml al construir aplicaciones multi-contenedor

El fichero docker-compose.yml permite definir un conjunto de servicios relacionados que se desplegarán como una aplicación compuesta, con comandos de despliegue.

Para usar un fichero **`docker-compose.yml`**, debe crearlo en la carpeta raíz de la solución principal, con un contenido similar al del siguiente ejemplo:

```
version: '3'
services:
  webmvc:
    image: eshop/web
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
    ports:
      - "80:80"
    depends_on:
      - catalog.api
      - ordering.api
  catalog.api:
    image: eshop/catalog.api
    environment:
      - ConnectionString=Server=sql.data;Port=1433;Database=CatalogDB;…
    ports:
      - "81:80"
    depends_on:
      - postgres.data
  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=sql.data;Database=OrderingDb;…
    ports:
      - "82:80"
    extra_hosts:
      - "CESARDLBOOKVHD:10.0.75.1"
    depends_on:
      - sql.data
  sql.data:
    image: mssql-server-linux:latest
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
```

Tenga en cuenta que este fichero **`docker-compose.yml`** es una versión simplificada y combinada (con varios contenedores). Contiene datos estáticos de configuración para cada contenedor (como el nombre de la imagen personalizada), que siempre aplican, además de la información de configuración que puede depender del entorno de despliegue, como la cadena de conexión. En secciones posteriores, veremos cómo puede dividir la configuración de **`docker-compose.yml`** en varios ficheros **`docker-compose`** y sobreescribir valores según el entorno y el tipo de ejecución (depuración o publicación).

En el fichero **`docker-compose.yml`** del ejemplo, se definen cinco servicios: el servicio **`webmvc`** (una aplicación web), dos microservicios (**`ordering.api`** y **`basket.api`**) y un contenedor de fuente de datos, **`sql.data`**, basado en SQL Server para Linux que se ejecuta como un contenedor. Cada servicio se implementará como un contenedor, por lo que se requiere una imagen Docker para cada uno.

El fichero **`docker-compose.yml`** especifica no solo qué contenedores se están utilizando, sino cómo se configuran individualmente. Por ejemplo, la definición del contenedor **`webmvc`** en el fichero .yml:

- Utiliza una imagen **`eshop/web:latest`**. Sin embargo, también se podría configurar para que se construya la imagen como parte de la ejecución del **`docker-compose`**, con una configuración adicional basada en una sección **`build:`** en el fichero **`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 **`depends_on`**. Esto hace que el servicio espere hasta que se inicien esos servicios.

Revisaremos nuevamente el fichero **`docker-compose.yml`** más adelante, cuando veamos cómo implementar microservicios y aplicaciones multi-contenedor.

Trabajando con docker-compose.yml en Visual Studio 2017

Cuando se agrega soporte Docker a un proyecto en una solución de Visual Studio, como se muestra en la Figura 5-7, Visual Studio agrega un fichero **`Dockerfile`** al proyecto y agrega una sección de servicios (un proyecto) en la solución con los ficheros **`docker-compose.yml`**. Esto facilita comenzar a componer una solución de contenedores múltiples. A continuación, puede abrir los ficheros **`docker-compose.yml`** y actualizarlos con funciones adicionales.

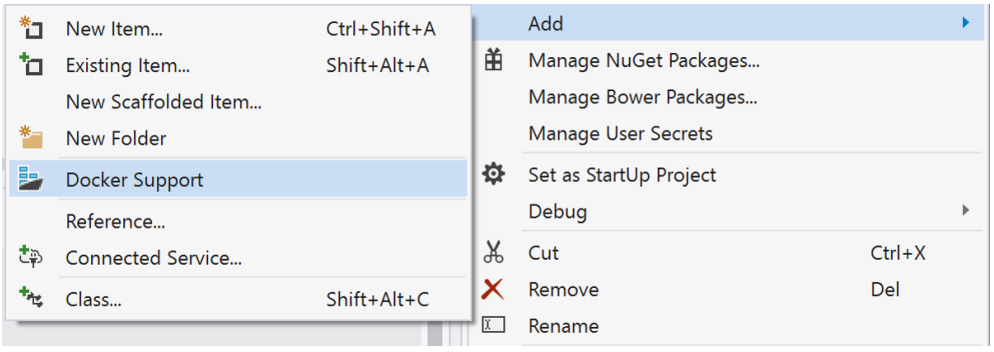


***Figura 5-7**. Añadiendo soporte para Docker en Visual Studio 2017 con click-derecho en un proyecto ASP.NET Core*

Después de agregar soporte para Docker a la solución en Visual Studio, también se agrega un nuevo nodo (en el fichero de proyecto **`docker-compose.dcproj`**) en el Solution Explorer, que contiene los ficheros **`docker-compose.yml`** agregados, como se muestra en la Figura 5- 8.

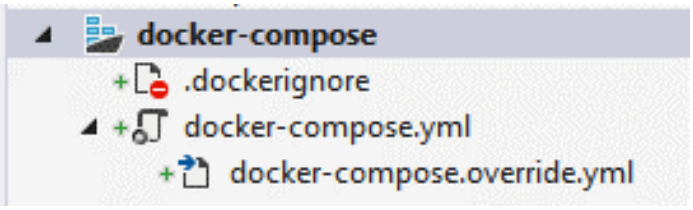


***Figura 5-8**. El nodo **docker-compose** añadido al Solution Explorer en Visual Studio 2017*

Se puede desplegar una aplicación multi-contenedor con un fichero único **`docker-compose.yml`** usando el comando **`docker-compose up`**. Sin embargo, Visual Studio agrega un grupo de ellos para que sea más fácil reemplazar los valores según el entorno (desarrollo versus producción) y el tipo de ejecución (ejecución versus depuración). Esta facilidad se explicará en secciones posteriores.

Paso 5.Construir y ejecutar la aplicación Docker

Si la aplicación sólo tiene un contenedor único, se puede ejecutar desplegándolo en un *host* Docker (máquina virtual o servidor físico). Sin embargo, si la aplicación contiene múltiples servicios, se puede desplegar como una aplicación compuesta, ya sea utilizando un único comando CLI (**`docker-compose up`**) o con Visual Studio, que utilizará ese comando por debajo. Veamos las diferentes opciones.

Opción A: Ejecutar un contenedor único con Docker CLI

Se puede ejecutar un contenedor Docker usando el comando **`docker run`**, como se muestra en la Figura 5-9:

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

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

***Figura 5-9**. Ejecutando un contenedor Docker con el comando* `docker run`

En este caso, el comando vincula el puerto interno 5000 del contenedor al puerto 80 del *host*. Esto significa que el *host* está escuchando en el puerto 80 y reenviando al puerto 5000 en el contenedor.

Opción B: Ejecutando una aplicación multi-contenedor

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

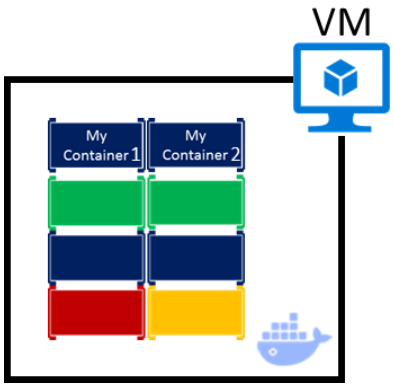


Figura 5-10. *Máquina virtual con contenedores Docker desplegados*

Ejecutando una aplicación multi-contenedor con el Docker CLI

Para ejecutar una aplicación multi-contenedor con Docker CLI, se puede ejecutar el comando **`docker-compose up`**. Este comando usa el fichero **`docker-compose.yml`**, que está a nivel de la solución, para desplegar una aplicación multi-contenedor. La Figura 5-11 muestra los resultados al ejecutar el comando desde el directorio principal de la solución, que contiene el fichero **`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. *Ejemplo de resultados cuando se ejecuta el comando* `docker-compose up`

Cuando se ejecuta el comando **`docker-compose up`**, la aplicación y sus contenedores relacionados se despliegan en el *host* Docker, como se ilustra en la representación de máquina virtual en la figura 5-10.

Ejecutando y depurando una aplicación multi-contenedor con Visual Studio

No podría ser más sencillo ejecutar una aplicación multi-contenedor con Visual Studio 2017. No sólo se puede ejecutar la aplicación multi-contenedor, sino que también se pueden depurar todos sus contenedores directamente desde Visual Studio estableciendo *break-points* comunes y corrientes.

Como se mencionó anteriormente, cuando se agrega soporte para Docker a un proyecto, dentro de una solución, ese proyecto se configura en el fichero global **`docker-compose.yml`** (a nivel de solución), que permite ejecutar o depurar toda la solución a la vez. Visual Studio iniciará un contenedor para cada proyecto que tenga habilitada la compatibilidad con Docker y realizará todos los pasos internos necesarios (**`dotnet publish`**, **`docker build`**, etc.).

El punto importante aquí es que, como se muestra en la Figura 5-12, en Visual Studio 2017 hay un comando Docker adicional para la tecla F5. Esta opción permite ejecutar o depurar una aplicación de varios contenedores, ejecutando todos los contenedores definidos en los ficheros **`docker-compose.yml`** a nivel de la solución. La capacidad de depurar soluciones de varios contenedores significa que se pueden establecer varios *break-points*, uno en cada proyecto diferente (contenedor) y,

al depurar desde Visual Studio, se detendrá en puntos definidos en los diferentes proyectos y ejecutándose en los contenedores diferentes.

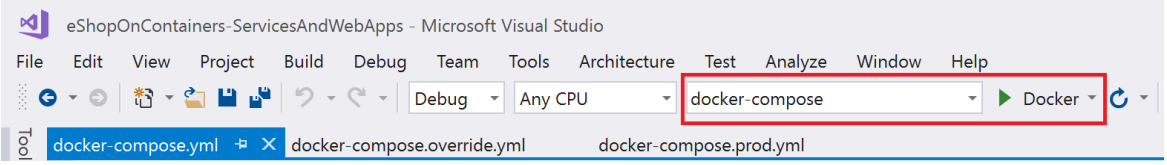


***Figura 5-12**. Ejecutando aplicaciones multi-contenedores con Visual Studio 2017*

Recursos adicionales

- **Deploy an ASP.NET container to a remote Docker host**
  https://azure.microsoft.com/documentation/articles/vs-azure-tools-docker-hosting-web-apps-in-docker/

Un comentario sobre pruebas y despliegue 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 el entorno de desarrollo. Pero no se debe usar este enfoque cuando el objetivo es desplegar en *clusters* Docker y orquestadores como Docker Swarm, Mesosphere DC/OS o Kubernetes. Si está utilizando un *cluster* como Docker en modo Swarm (disponible en Docker CE para Windows y Mac desde la versión 1.12), se debe desplegar y probar con comandos adicionales como **`docker service create`** para servicios individuales. Si está desplegando una aplicación multi-contenedor, se deben usar los comandos **`docker compose bundle`** y **`docker deploy myBundleFile`** para implementar la aplicación compuesta como un *stack*. Para obtener más información, consulte el artículo Introducing Experimental Distributed Application Bundles en el blog de Docker.

Para DC/OS y Kubernetes también se usarían comandos y *scripts* de despliegue diferentes.

Paso 6. Probar la aplicación Docker en el *host* Docker local

Este paso variará dependiendo de lo que haga la aplicación. En una aplicación web .NET Core simple, que se despliega como un solo contenedor o servicio, se puede acceder al servicio abriendo un navegador en el *host* Docker y navegando hacia el sitio como se muestra en la Figura 5-13. (Si la configuración en el fichero **`Dockerfile`** mapea el contenedor a un puerto en el *host* que no sea 80, se debe incluir la publicación de *host* en la URL).

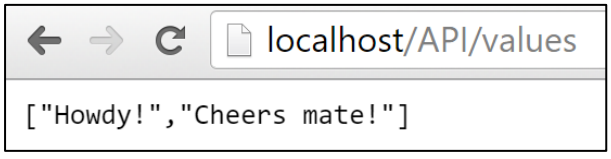


***Figura 5-13**. Ejemplo de pruebas locales de la aplicación Docker usando localhost*

Si localhost no está apuntando a la IP del *host* Docker (es el valor por defecto al usar Docker CE), para navegar a su servicio, use la dirección IP del adaptador de red de su máquina.

Tenga en cuenta que esta URL en el navegador usa el puerto 80 para el ejemplo. Sin embargo, internamente, las peticiones se redireccionan al puerto 5000, porque así fue como se desplegó con el comando **`docker run`**, como se explicó en un paso anterior.

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

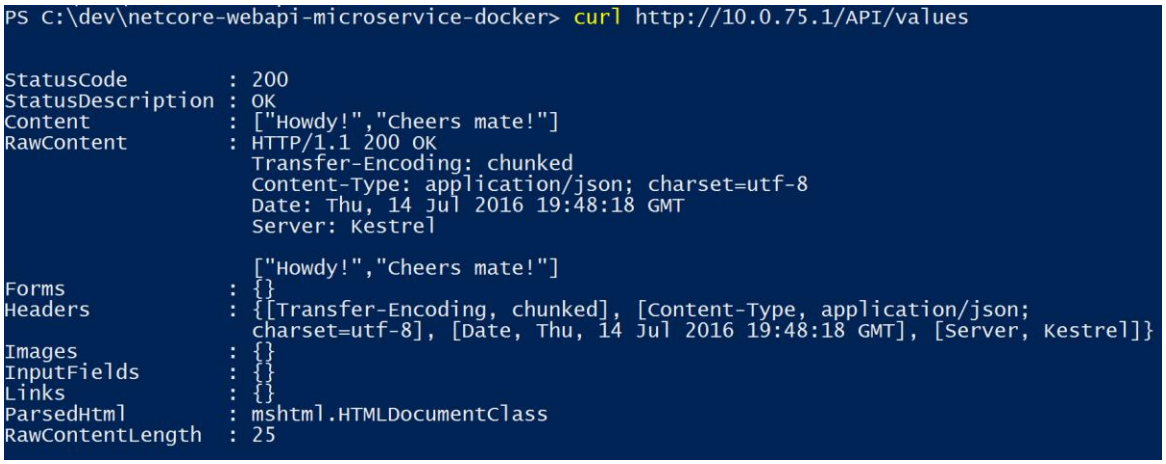


***Figura 5-14**. Ejemplo de pruebas locales de la aplicación Docker usando curl*

Probando y depurando contenedores con Visual Studio 2017

Al ejecutar y depurar contenedores con Visual Studio 2017, se puede depurar la aplicación .NET de la misma forma como lo haría al ejecutarla sin contenedores.

Probando y depurando sin Visual Studio

Si se está desarrollando con el enfoque editor/CLI, la depuración de contenedores es más difícil y tendrá que hacerlo generando trazas.

Recursos adicionales

- **Depuración de aplicaciones en un contenedor de Docker local**
  https://azure.microsoft.com/documentation/articles/vs-azure-tools-docker-edit-and-refresh/
- **Steve Lasker. Build, Debug, Deploy ASP.NET Core Apps with Docker.** Video.
  https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T115

Flujo de trabajo simplificado para desarrollar contenedores con Visual Studio

El flujo de trabajo al usar Visual Studio es mucho más simple y eficiente que con el enfoque editor/CLI. La mayoría de los pasos requeridos por Docker, relacionados con los ficheros **`Dockerfile`** y **`docker-compose.yml`**, están ocultos o simplificados por Visual Studio, como se muestra en la Figura 5-15.

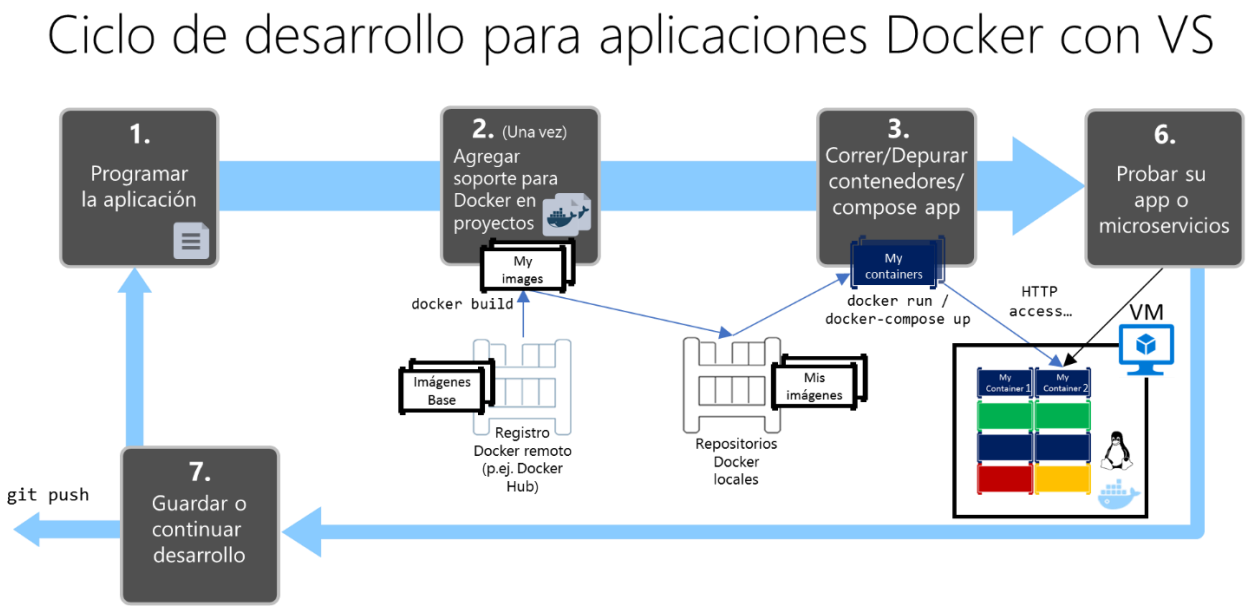


***Figura 5-15**. Proceso simplificado cuando se usa Visual Studio*

Además, sólo necesita realizar el paso 2 (agregar soporte Docker a sus proyectos) una vez. Por lo tanto, el flujo de trabajo es similar a las tareas habituales cuando usa .NET para cualquier otro desarrollo. Sin embargo, es necesario saber qué sucede por debajo (el proceso de creación de imágenes, qué imágenes base está utilizando, despliegue de contenedores, etc.) y, a veces, también necesitará editar el fichero **`Dockerfile`** o **`docker-compose.yml`**, para personalizar los comportamientos. Pero la mayoría del trabajo se simplifica enormemente al usar Visual Studio, lo que le hace mucho más productivo.

Recursos adicionales

- **Steve Lasker. .NET Docker Development with Visual Studio 2017**
  https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T111
- **Jeffrey T. Fritz. Put a .NET Core App in a Container with the new Docker Tools for Visual Studio**
  https://blogs.msdn.microsoft.com/webdev/2016/11/16/new-docker-tools-for-visual-studio/

Usando comandos PowerShell en un fichero Dockerfile para configurar Contenedores de Windows

Los Contenedores de Windows le permiten convertir sus aplicaciones de Windows existentes en imágenes Docker y desplegarlas con las mismas herramientas que el resto del ecosistema de Docker. Para usar Contenedores de Windows, incluya los comandos de PowerShell en el fichero **`Dockerfile`**, como se muestra en el siguiente ejemplo:

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

En este caso, estamos utilizando una imagen base de Windows Server Core (la configuración FROM) e instalando IIS con un comando de PowerShell (la configuración RUN). De manera similar, también podría usar los comandos de PowerShell para configurar componentes adicionales como ASP.NET 4.x, .NET 4.6 o cualquier otro software de Windows. Por ejemplo, el siguiente comando en un **`Dockerfile`** configura ASP.NET 4.5:

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

Recursos adicionales

- **Microsoft/aspnet-docker.**
  Ejemplos de comandos Powershell en dockerfiles para incluir características de Windows.
  https://github.com/Microsoft/aspnet-docker/blob/master/4.7.1-windowsservercore-ltsc2016/runtime/Dockerfile

SECCIÓN 6

Diseñando y Desarrollando Aplicaciones .NET Multi-Contenedor y Basadas en Microservicios

Visión

Desarrollar aplicaciones contenerizadas de microservicios significa que está creando aplicaciones multi-contenedor. Sin embargo, una aplicación de multi-contenedores también podría ser más simple, por ejemplo, una aplicación de tres capas y se podría construir sin usar una arquitectura de microservicios.

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

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

Diseñando una aplicación orientada a microservicios

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

Especificaciones de la aplicación

La aplicación hipotética maneja las peticiones ejecutando la lógica de negocios, accediendo a bases de datos y luego devolviendo respuestas HTML, JSON o XML. Diremos que la aplicación debe soportar una variedad de clientes, incluidos los navegadores de escritorio que ejecutan *Single Page Applications* (SPA), aplicaciones web tradicionales, aplicaciones web móviles y aplicaciones móviles nativas. La aplicación también podría exponer una API para que la consumieran terceros. También debería ser capaz de integrar sus microservicios o aplicaciones externas de forma asíncrona, por lo que ese enfoque ayudará a la resiliencia de los microservicios en el caso de fallas parciales.

La aplicación tendrá estos tipos de componentes:

- **Componentes de presentación**. Son responsables de manejar la interfaz de usuario y consumir servicios remotos.
- **Lógica del dominio o del negocio**. Es la lógica de dominio de la aplicación.
- **Lógica de acceso a la base de datos**. Son los componentes responsables de acceder a las bases de datos (SQL o NoSQL).
- **Lógica de integración de aplicaciones**. Es la lógica que permite la comunicación entre los componentes que implementan la lógica del negocio. Esto incluye un canal de mensajería, basado principalmente en *brokers* de mensajes.

La aplicación deberá tener alta escalabilidad y permitir que los subsistemas verticales escalen de forma autónoma, porque ciertos subsistemas requerirán más escalabilidad que otros.

Se debe poder desplegar la aplicación en múltiples entornos de infraestructura (múltiples nubes públicas y locales) y, preferiblemente, debe ser multiplataforma, capaz de pasar de Linux a Windows (o viceversa) fácilmente.

Contexto del equipo de desarrollo

También suponemos lo siguiente en cuanto al proceso de desarrollo

- Tiene múltiples equipos de desarrollo enfocándose en diferentes áreas de negocio de la aplicación.
- Los nuevos miembros del equipo deben ser productivos rápidamente y la aplicación debe ser fácil de entender y modificar.
- La aplicación se utilizará durante un largo plazo, en que irá evolucionando y reglas del negocio estarán cambiando constantemente.
- Necesita que sea fácilmente mantenible a largo plazo, lo que significa tener agilidad para implementar cambios en el futuro y poder actualizar los subsistemas con un impacto mínimo en el resto.
- Desea usar integración despliegue continuo de la aplicación.
- Desea aprovechar las tecnologías emergentes (*frameworks*, lenguajes de programación, etc.) mientras desarrolla la aplicación. No desea realizar migraciones completas de la aplicación cuando se migren a nuevas tecnologías, ya que eso tendría costes elevados e impacto en la estabilidad y confiabilidad de la aplicación.

Seleccionando una arquitectura

¿Cuál debería ser la arquitectura para desplegar de la aplicación? Las especificaciones para 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 colaborativos, donde cada microservicio sea un contenedor.

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

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

Los microservicios se desarrollan y despliegan como contenedores de forma independiente. Esto significa que un equipo puede desarrollar y desplegar un microservicio sin afectar a otros subsistemas.

Cada microservicio tiene su propia base de datos, lo que le permite estar totalmente desacoplado de otros microservicios. Cuando es necesario, la consistencia entre bases de datos de diferentes microservicios se logra usando eventos de integración de nivel de aplicación (a través de un bus de eventos lógicos), tal como se maneja en *Command and Query Responsibility Segregation* (CQRS). Por eso, las restricciones del negocio deben adecuarse a la consistencia eventual los microservicios y sus bases de datos.

eShopOnContainers: Una aplicación de referencia en .NET Core basada en microservicios y desplegada en contenedores

Para que se pueda enfocar en la arquitectura y las tecnologías, en lugar de pensar en un dominio de negocio hipotético que quizás no conozca, hemos seleccionado un dominio de negocio bien conocido, específicamente, 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 del negocio. El código fuente de la aplicación basada en contenedores está disponible en el repositorio eShopOnContainers en GitHub.

La aplicación está compuesta por varios subsistemas, que incluyen varias opciones de interfaz de usuario de la tienda (una aplicación web y una aplicación móvil nativa), junto con los microservicios y contenedores de servicios de *back-end* para todas las operaciones del lado del servidor. La Figura 6-1 muestra la arquitectura de la aplicación de referencia.

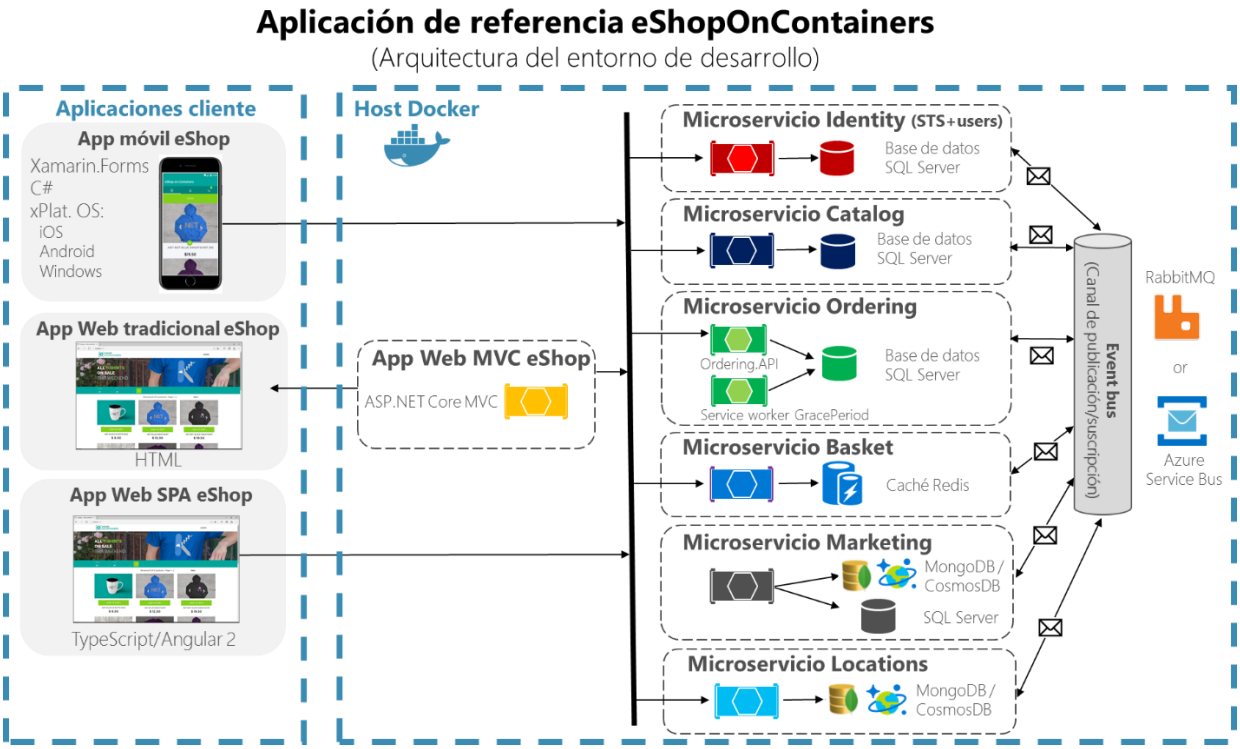


Figura 6-1. *La arquitectura del entorno de desarrollo para la aplicación de referencia eShopOnContainers*

Entorno de *hosting*. En la Figura 6-1, verá varios contenedores desplegados dentro de un solo *host* Docker. Ese sería el caso cuando se implementa en un *host* Docker único con el comando **`docker-compose up`**. Sin embargo, si está usando un orquestador o *cluster* de contenedores, cada contenedor podría estar ejecutándose en un *host* (nodo) diferente y cualquier nodo podría estar ejecutando cualquier cantidad de contenedores, como explicamos anteriormente en la sección de arquitectura.

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

- Comunicación directa de cliente a microservicio. Se usa para consultas y al aceptar comandos de actualización o transaccionales de las aplicaciones cliente.
- Comunicación asíncrona basada en eventos. Esto ocurre a través de un bus de eventos para propagar actualizaciones a través de microservicios o para integrar con aplicaciones externas. El bus de eventos se puede implementar con cualquier tecnología de infraestructura de mensajería como RabbitMQ, o utilizando buses de servicio de mayor 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 contenedores y también puede haber comunicación entre los microservicios. Como se mencionó, esta arquitectura inicial está usando comunicación directa de cliente a microservicio, lo que significa que una aplicación cliente puede realizar solicitudes a cada uno de los microservicios directamente. Cada microservicio tiene un *endpoint* público como **`https://servicename.applicationname.companyname`**. Si es necesario, cada microservicio puede usar un puerto TCP diferente. En producción, esa URL se correlacionaría con

el balanceador de carga de los microservicios, que distribuye las solicitudes entre las instancias disponibles del microservicio.

Nota importante sobre API *Gateway* versus comunicación directa en eShopOnContainers. Como se explicó en la sección de arquitectura de esta guía, la arquitectura de comunicación directa de cliente a microservicio puede tener inconvenientes cuando se construye una aplicación grande y compleja basada en microservicios. Pero puede ser suficientemente bueno para una aplicación pequeña, como eShopOnContainers, donde el objetivo es enfocarse en una aplicación basada en contenedores Docker más simple y no queríamos crear una API *Gateway* monolítica, que pudiese impactar la autonomía de desarrollo de los microservicios.

Pero, si va a diseñar una aplicación grande basada en microservicios con docenas de microservicios, le recomendamos encarecidamente que considere el patrón API *Gateway*, como explicamos en la sección de arquitectura.

Se podría refactorizar esta decisión arquitectónica de una vez, pensando en aplicaciones para producción y fachadas especialmente diseñadas para clientes remotos. Tener múltiples *API Gateways* personalizados, dependiendo del dispositivo cliente puede proporcionar beneficios, como se mencionó en la sección de arquitectura, pero, como también se mencionó, las *API Gateways* grandes y monolíticas que podrían matar la autonomía de desarrollo de sus microservicios.

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 que son SQL Server) están en el mismo contenedor. Esta decisión de diseño se hizo sólo para facilitar que un desarrollador obtenga el código de GitHub, lo clone y lo abra en Visual Studio o Visual Studio Code. Por otro lado, esto hace que sea fácil compilar las imágenes Docker personalizadas utilizando .NET Core CLI y Docker CLI y luego desplegarlas y ejecutarlas en un entorno de desarrollo Docker. De cualquier manera, el uso de contenedores para bases de datos permite a los desarrolladores crear e implementar en cuestión de minutos sin tener que preparar una base de datos externa o cualquier otra fuente de datos con dependencias duras en la infraestructura (tanto en la nube o como *on-premises*).

En un entorno de producción real, para lograr alta disponibilidad y escalabilidad, las bases de datos deberían estar en servidores dedicados en la nube o en servidores *on-premises*, pero no en contenedores.

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

Recursos adicionales

- **eShopOnContainers GitHub repo. Source code for the reference application**
  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 más fácil que un desarrollador pueda comprender y comenzar rápidamente con buena productividad.
- Los contenedores arrancan rápido, lo que hace que los desarrolladores sean más productivos.
- Un IDE como Visual Studio puede cargar rápidamente proyectos más pequeños, aumentando la productividad.
- Cada microservicio se puede diseñar, desarrollar y desplegar independientemente de otros, lo que proporciona agilidad, porque es más fácil desplegar frecuentemente nuevas versiones.

Es posible escalar áreas individuales de la aplicación. Por ejemplo, es posible que el servicio de catálogo o el servicio del carrito de compras se deban escalar, pero no el proceso de pedidos. Una infraestructura de microservicios será mucho más eficiente en el uso recursos, cuando se escala, que una arquitectura monolítica.

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

Los problemas están más aislados. Si hay un problema en un servicio, sólo ese servicio se ve afectado inicialmente (excepto cuando se usa un diseño incorrecto, con dependencias directas entre microservicios) y otros servicios pueden seguir atendiendo las solicitudes. Por el contrario, si un componente funciona mal en una arquitectura de despliegue monolítico puede colapsar todo el sistema, especialmente cuando involucra recursos del entorno, como una fuga de memoria. Además, cuando se resuelve un problema en un microservicio, puede desplegar sólo el microservicio afectado sin afectar el resto de la aplicación.

Puede usar las últimas tecnologías. Como puede comenzar a desarrollar servicios de forma independiente y ejecutarlos uno al lado del otro (gracias a los contenedores y .NET Core), puede comenzar a utilizar las últimas tecnologías y *frameworks* más rápidamente en lugar de quedarse atrapado en una *stack* o *framework* más antiguo para toda la aplicación.

Inconvenientes de una solución basada en microservicios

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

Aplicación distribuida. Hacer una aplicación distribuida agrega complejidad para los desarrolladores cuando diseñan y construyen los servicios. Por ejemplo, los desarrolladores deben implementar la comunicación entre servicios utilizando protocolos como HTTP o AMPQ, lo que agrega complejidad para las pruebas y el manejo de excepciones. También agrega latencia al sistema.

Complejidad de despliegue. Una aplicación que tiene docenas de tipos de microservicios y necesita alta escalabilidad (necesita poder crear muchas instancias por servicio y balancear esos servicios entre muchos *hosts*) implica un alto grado de complejidad de despliegue para la gestión de TI e infraestructura. Si no está utilizando una infraestructura orientada a microservicios (como un orquestador y programador), esa complejidad adicional puede requerir mucho más esfuerzo de desarrollo que la aplicación del negocio en sí misma.

Transacciones atómicas. generalmente no son posibles las transacciones atómicas entre múltiples microservicios. Los requisitos del negocio deben manejar la consistencia 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 que necesita inicialmente la nueva aplicación, será mayor que las de la aplicación monolítica original. Esto 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 sólo ciertas áreas de la aplicación, en comparación con los costes a largo plazo cuando se desarrollan aplicaciones monolíticas, el mayor uso de recursos suele representar un buen compromiso

Problemas con la comunicación directa de cliente a microservicio. Cuando la aplicación es grande, con docenas de microservicios, existen desafíos y limitaciones si la aplicación requiere comunicaciones directas de cliente a microservicio. Un problema es la posible falta de correspondencia entre las necesidades del cliente y las API expuestas por cada uno de los microservicios. En ciertos casos, es posible que la aplicación cliente tenga que realizar muchas solicitudes por separado para componer la interfaz de usuario, que puede ser ineficiente a través de Internet y no sería práctico en una red móvil. Por lo tanto, se deben minimizar las peticiones de la aplicación cliente al *back-end*.

Otro problema con las comunicaciones directas de cliente a microservicio es que algunos microservicios podrían estar usando protocolos que no son compatibles con la Web. Un servicio podría usar un protocolo binario, mientras que otro podría usar mensajes AMQP. Esos protocolos no son compatibles con un firewall y es mejor usarlos sólo 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 hace que sea difícil refactorizar los contratos para esos microservicios. Con el tiempo, los desarrolladores pueden querer cambiar la forma en que el sistema se divide en servicios. Por ejemplo, podrían 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 del cliente.

Como se mencionó en la sección de arquitectura, cuando diseñe y construya una aplicación compleja basada en microservicios, podría considerar el uso de *API Gateways* múltiples de grano fino en lugar de una comunicación simple y directa de cliente a microservicio.

Partición de los microservicios. Finalmente, sin importar qué enfoque elija para su arquitectura de microservicios, otro desafío es decidir cómo particionar una aplicación completa en múltiples microservicios. Como se señala en la sección de arquitectura de la guía, hay varias técnicas y enfoques que puede usar. Básicamente, debe identificar las áreas de la aplicación que están desacopladas de otras y que tienen un bajo número de dependencias fuertes. En muchos casos, esto se corresponde con la partición de los servicios por caso de uso. Por ejemplo, en nuestra aplicación de e-shop, tenemos un servicio de pedidos que es responsable de toda la lógica del negocio relacionada con el proceso de pedido. También tenemos el servicio de catálogo y el servicio de carrito de compras que implementa otras capacidades. Idealmente, cada servicio debe tener sólo un pequeño conjunto de responsabilidades. Esto es similar al principio de responsabilidad única (SRP) aplicado a las clases, que establece que una clase sólo debe tener un motivo para cambiar. Pero en este caso, se trata de microservicios, por lo que el alcance será mayor que el de una sola clase. Más que nada, un microservicio tiene que ser completamente autónomo, de principio a fin, incluida la responsabilidad de sus propias fuentes de datos.

Arquitectura externa versus arquitectura interna y patrones de diseño

La arquitectura externa es la arquitectura de microservicios compuesta por servicios múltiples, 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 patrones diferentes, para diferentes microservicios. Los microservicios incluso pueden usar tecnologías y lenguajes de programación diferentes. La Figura 6-2 ilustra esta diversidad.

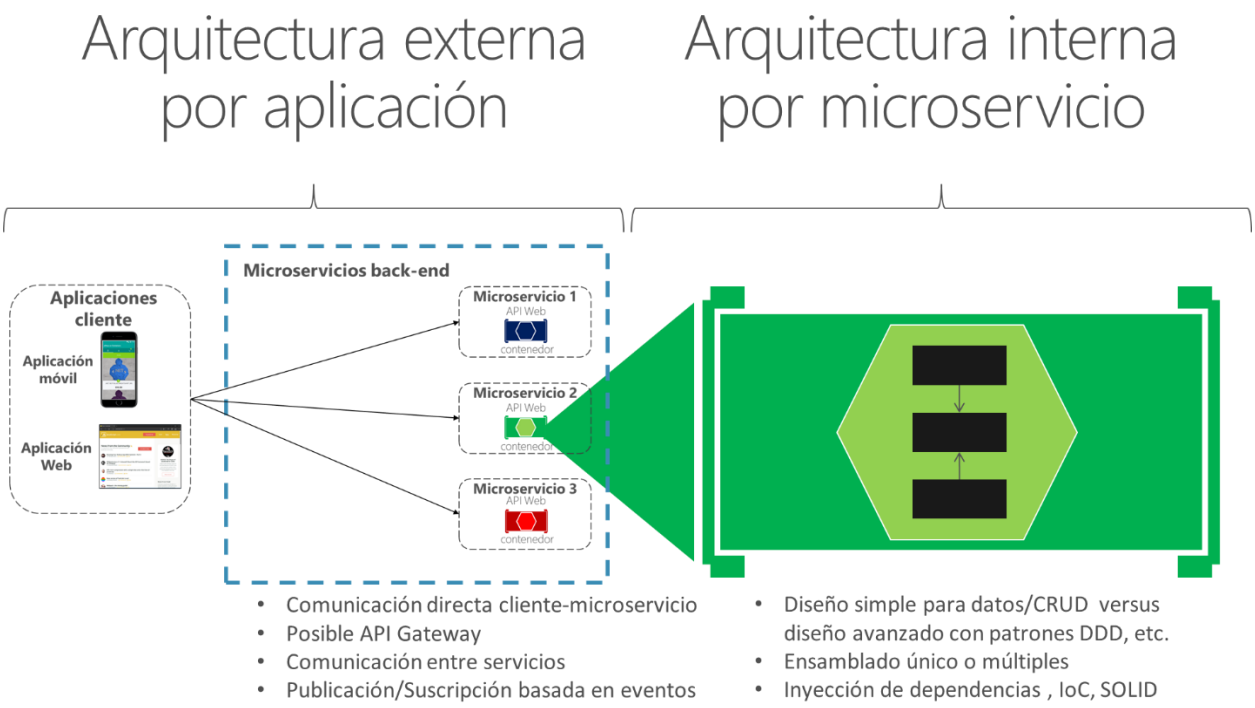


***Figura 6-2**. Diferencias entre arquitectura y diseño internos y externos*

Por ejemplo, en nuestro ejemplo de eShopOnContainers, los microservicios de catálogo, carrito de compras y perfil de usuario son simples (básicamente, subsistemas CRUD – Create, Read, Update, Delete – Crear, Leer, Actualizar, Eliminar). Por lo tanto, su arquitectura y diseño interno es sencillo. Sin embargo, es posible que tenga otros microservicios, como el microservicio de pedidos, que es más complejo y representa reglas del negocio en constante cambio, con un alto grado de complejidad del dominio. En casos como estos, es posible que desee implementar patrones más avanzados dentro de un microservicio particular, como los definidos con los enfoques de diseño orientados por el dominio (DDD), como lo estamos haciendo en el microservicio de pedidos de eShopOnContainers. (Revisaremos estos patrones de DDD posteriormente, en la sección que explica la implementación del microservicio de pedidos).

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 está apuntando a dominios de IA (Inteligencia Artificial) y de *machine learning* (aprendizaje de máquinas), en lugar de un lenguaje de programación 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 se tienen qué implementar usando patrones avanzados de DDD, porque eso sería exagerado. Del mismo modo, los microservicios complejos, con una lógica del negocio en cambio constante, no deberían implementarse como componentes CRUD, porque podría terminar con código de baja calidad.

El nuevo mundo: múltiples patrones arquitectónicos y servicios políglotas

Hay muchos patrones arquitectónicos usados por los arquitectos y desarrolladores de software. A continuación, se muestran algunos de ellos (mezcla de estilos y patrones de arquitectura):

- CRUD simple, un nivel, una capa.
- N-Capas tradicional.
- N-Capas Diseño Orientado por el Dominio (DDD).
- Clean Architecture (como la usada en eShopOnWeb)
- Command and Query Responsibility Segregation (CQRS).
- Event-Driven Architecture (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), F#, Node.js, Python, Java, C++, GoLang y más.

El punto importante es que ningún patrón o estilo de arquitectura, ni ninguna tecnología en particular, son adecuados para todas las situaciones. La Figura 6-3 muestra algunos enfoques y tecnologías (aunque no en un orden particular) que se podrían usar en diferentes microservicios.

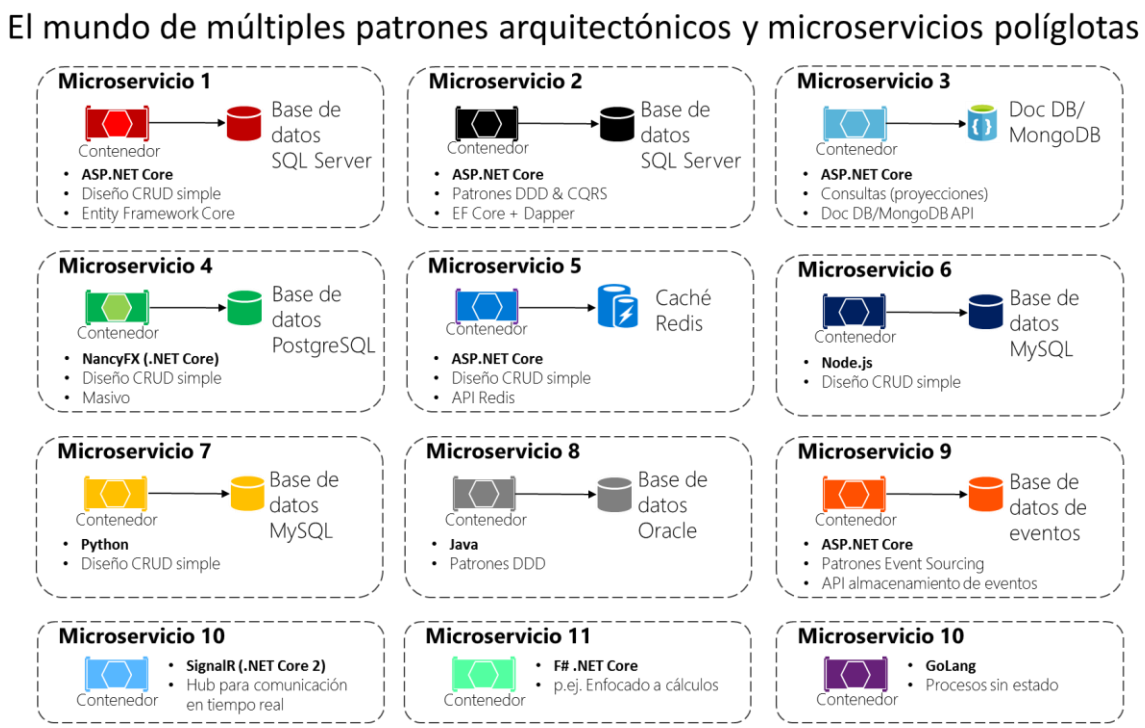


***Figura 6-3**. El mundo de múltiples patrones arquitectónicos y microservicios políglotas*

Como se muestra en la Figura 6-3, en aplicaciones compuestas por muchos microservicios (*Bounded Context* en terminología DDD, o simplemente "subsistemas" como microservicios autónomos), puede implementar cada microservicio de una forma diferente. Cada uno puede tener un patrón arquitectónico diferente y utilizar diferentes lenguajes y bases de datos, dependiendo de la naturaleza de la aplicación, los requisitos del negocio y las prioridades. En algunos casos, los microservicios pueden ser similares. Pero eso no es frecuente, ya que los límites y requisitos de cada subsistema suelen ser diferentes.

Por ejemplo, para una aplicación CRUD simple de mantenimiento, podría no tener sentido diseñar e implementar patrones DDD. Pero para su dominio o negocio principal, se podrían necesitar patrones más avanzados para abordar la complejidad del negocio y manejar reglas en cambio constante.

Especialmente cuando maneje aplicaciones grandes, compuestas por múltiples subsistemas, no debe aplicar una arquitectura única de nivel superior, basada en un patrón de arquitectura único. Por ejemplo, CQRS no tiene sentido como una arquitectura de nivel superior para una aplicación completa, pero podría ser útil para un conjunto específico de servicios.

No hay una solución universal o patrón de arquitectura correcto para todos los casos. No puede tener "un patrón de arquitectura que gobierne todo". Dependiendo de las prioridades de cada microservicio, debe elegir un enfoque más adecuado para cada uno, como se explica en las siguientes secciones.

Creando un microservicio CRUD simple para datos

Esta sección describe cómo crear un microservicio simple que realice operaciones de creación, lectura, actualización y eliminación (CRUD) en una fuente de datos.

Diseñando un microservicio CRUD simple

Desde el punto de vista del diseño, este tipo de microservicios contenerizados es muy simple. Tal vez el problema a resolver sea simple, o tal vez la implementación sea sólo una prueba de concepto.

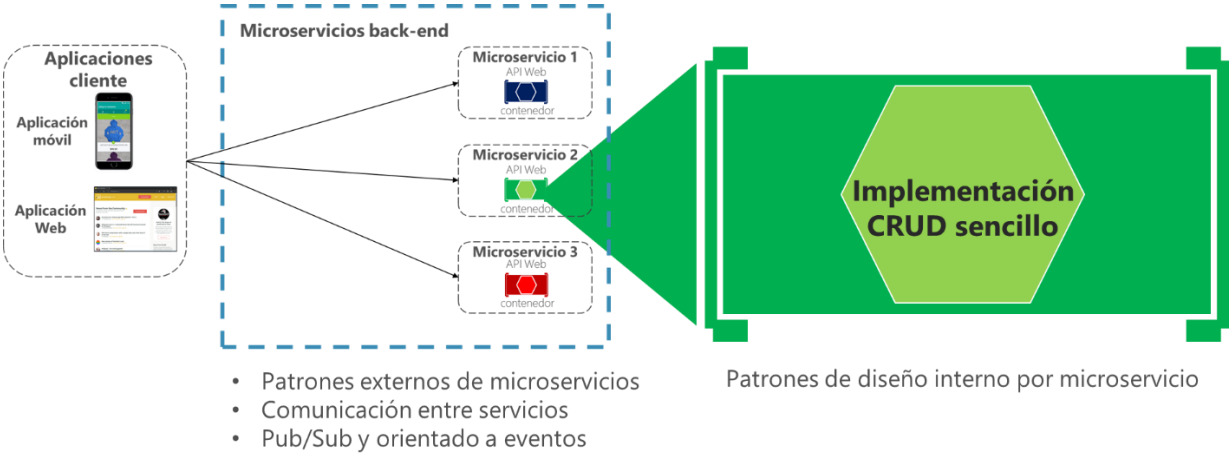


***Figura 6-4**. Diseño interno para microservicios CRUD sencillos*

Un ejemplo de este tipo es el microservicio de catálogo de la aplicación eShopOnContainers. Este tipo de servicio implementa toda su funcionalidad en un proyecto ASP.NET Web API único que incluye

clases para su modelo de datos, su lógica del negocio y su lógica 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 y prueba), pero también podría ser cualquier servidor SQL Server normal, como se muestra en la Figura 6-5.

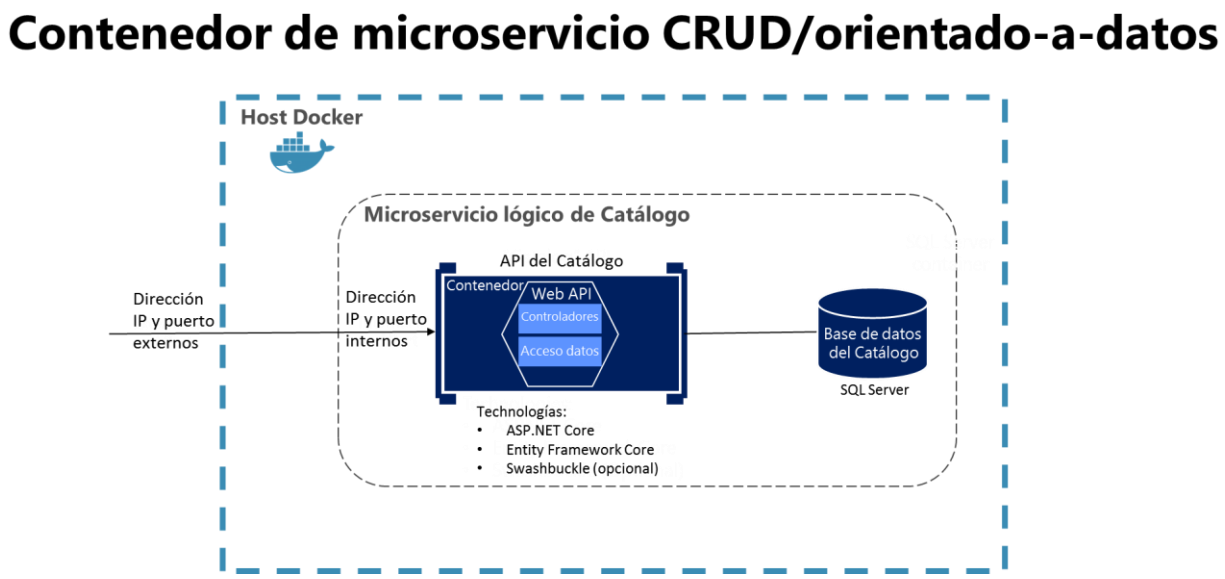


***Figura 6-5**. Diseño de un microservicio simple tipo CRUD/orientado a datos*

Cuando está desarrollando este tipo de servicio, solo necesita ASP.NET Core y una API de acceso a datos o 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 adecuado para entornos de desarrollo, porque puede tener todas sus dependencias en funcionamiento sin necesidad de preparar una base de datos en la nube o en las instalaciones internas. Esto es muy conveniente para ejecutar pruebas de integración. Sin embargo, para entornos de producción, no se recomienda hacerlo, ya que normalmente no se obtiene 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 DocumentDB.

Finalmente, al editar los ficheros 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.

Implementando un microservicio CRUD simple con ASP.NET Core

Para implementar un microservicio CRUD simple usando .NET Core y Visual Studio, se comienza creando un proyecto simple ASP.NET Core API Web (apuntando a .NET Core para que pueda correr en un *host* Docker de Linux), como se muestra en la Figura 6- 6.

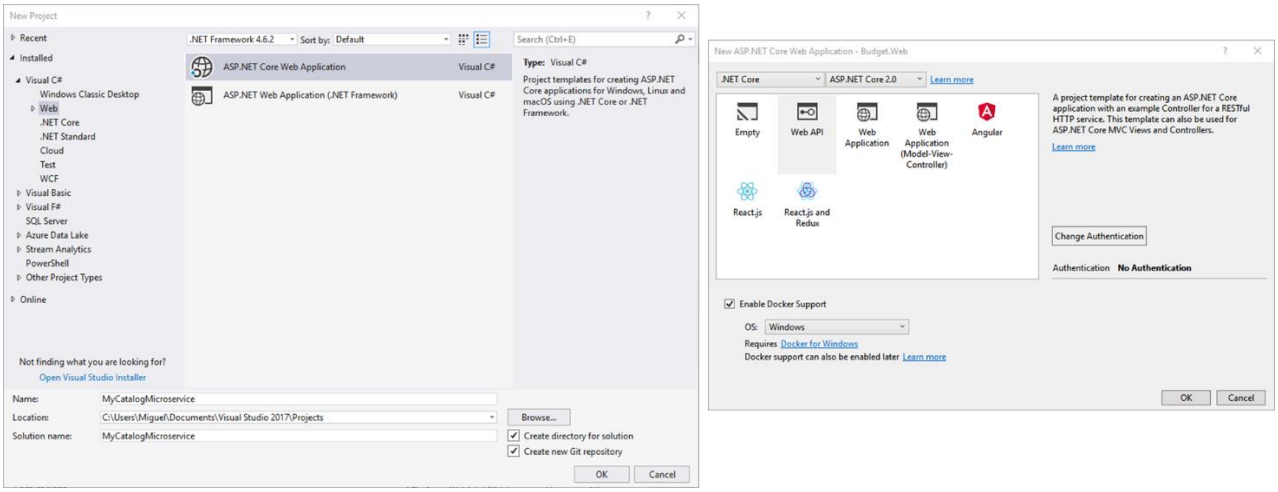


***Figura 6-6**. Creando un proyecto ASP.NET Core Web API en Visual Studio*

Después de crear el proyecto, puede implementar sus controladores MVC como lo haría en cualquier otro proyecto de API Web, utilizando la API de Entity Framework o cualquier otra. En un nuevo proyecto API Web, puede ver que la única dependencia que tiene en ese microservicio es ASP.NET Core. Internamente, dentro de la dependencia *Microsoft.AspNetCore.All*, se hace referencia a Entity Framework y muchos otros paquetes NuGet de .NET Core, como se muestra en la Figura 6-7.

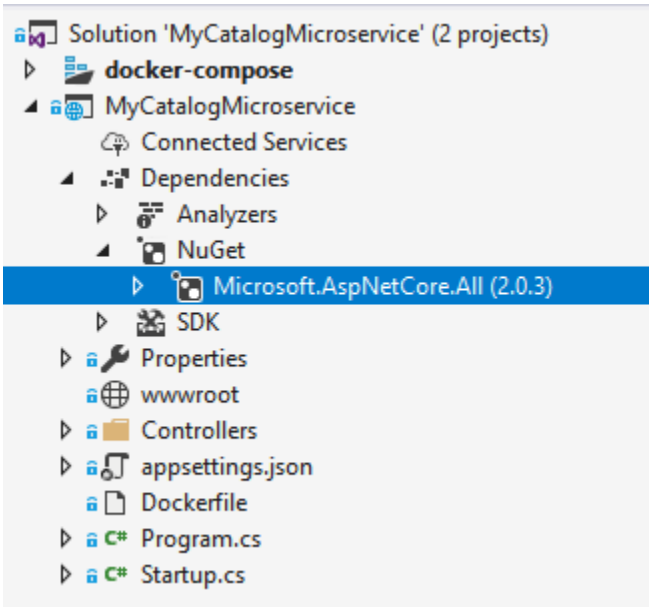


***Figura 6-7**. Dependencias en un microservicio API Web CRUD simple*

Implementando servicios CRUD API Web con Entity Framework Core

Entity Framework (EF) Core es una versión ligera, extensible y multiplataforma de la popular tecnología de acceso a datos de Entity Framework. EF Core es un mapeador objeto-relacional (ORM) que permite a los desarrolladores de .NET trabajar con una base de datos usando objetos .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 Docker de SQL Server para Linux. Sin embargo, la base de datos podría implementarse en cualquier Servidor SQL, como Windows local o Azure SQL DB. Lo único que tendría que 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 el uso de un modelo. Un modelo se compone de clases de entidades (del modelo de dominio) y un contexto derivado de una clase base de EF (DbContext), que representa una sesión con la base de datos, 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 migraciones EF para crear una base de datos a partir de su modelo, usando el enfoque *code first* (que facilita la evolución de la base de datos a medida que su modelo cambia con el tiempo). Para el microservicio de catálogo estamos utilizando el último enfoque. Puede ver un ejemplo de la clase **`CatalogItem`** en el siguiente ejemplo de código, que es una clase simple (POCO – Plain Old CLR Object), que corresponde a la entidad del modelo de dominio.

```
Public class CatalogItem
{
    public int Id { get; set; }
    public string Name { get; set; }
    public string Description { get; set; }
    public decimal Price { get; set; }
    public string PictureFileName { get; set; }
    public string PictureUri { get; set; }
    public int CatalogTypeId { get; set; }
    public CatalogType CatalogType { get; set; }
    public int CatalogBrandId { get; set; }
    public CatalogBrand CatalogBrand { get; set; }
    public int AvailableStock { get; set; }
    public int RestockThreshold { get; set; }
    public int MaxStockThreshold { get; set; }

    public bool OnReorder { get; set; }
    public CatalogItem() { }

    // Additional code ...
}
```

También necesita un **DbContext** que represente una sesión con la base de datos. Para el microservicio de catálogo, la clase **CatalogContext** deriva de la clase base **DbContext**, como se muestra en el siguiente ejemplo:

```
public class CatalogContext : DbContext
{
    public CatalogContext(DbContextOptions<CatalogContext> options) : base(options)
    {
    }
    public DbSet<CatalogItem> CatalogItems { get; set; }
    public DbSet<CatalogBrand> CatalogBrands { get; set; }

    public DbSet<CatalogType> CatalogTypes { get; set; }
    // Additional code ...
}
```

Puede tener implementaciones adicionales de **DbContext**. Por ejemplo, en el microservicio Catalog.API, hay un segundo **DbContext** llamado **CatalogContextSeed** donde se cargan automáticamente los datos de ejemplo la primera vez que intenta acceder a la base de datos. Este método es útil para datos de demostración.

Dentro del **DbContext**, utilice el método **OnModelCreating** para personalizar el mapeo objeto/base de datos de las entidades del modelo de dominio y otros puntos de extensibilidad de EF.

Consultando datos desde controladores API Web

Las instancias de las clases de entidades normalmente se recuperan de la base de datos utilizando Language Integrated Query (LINQ), como se muestra en el siguiente ejemplo:

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _catalogContext;
    private readonly CatalogSettings _settings;
    private readonly ICatalogIntegrationEventService
                                            _catalogIntegrationEventService;
    public CatalogController(CatalogContext context,
                             IOptionsSnapshot<CatalogSettings> settings,
                             ICatalogIntegrationEventService
                                              catalogIntegrationEventService)
    {
        _catalogContext = context ?? throw new
                                        ArgumentNullException(nameof(context));
        _catalogIntegrationEventService = catalogIntegrationEventService ??
throw new ArgumentNullException(nameof(catalogIntegrationEventService));
        _settings = settings.Value;
        ((DbContext)context).ChangeTracker.QueryTrackingBehavior =
                                                QueryTrackingBehavior.NoTracking;
    }
    // GET api/v1/[controller]/items[?pageSize=3&pageIndex=10]
    [HttpGet]
    [Route("[action]")]
    [ProducesResponseType(typeof(PaginatedItemsViewModel<CatalogItem>),
                          (int)HttpStatusCode.OK)]
    public async Task<IActionResult> Items([FromQuery]int pageSize = 10,
                                           [FromQuery]int pageIndex = 0)
    {
        var totalItems = await _catalogContext.CatalogItems
            .LongCountAsync();

        var itemsOnPage = await _catalogContext.CatalogItems
            .OrderBy(c => c.Name)
            .Skip(pageSize * pageIndex)
            .Take(pageSize)
            .ToListAsync();

        itemsOnPage = ChangeUriPlaceholder(itemsOnPage);
        var model = new PaginatedItemsViewModel<CatalogItem>(
            pageIndex, pageSize, totalItems, itemsOnPage);
```

```
        return Ok(model);
    }
    //...
}
```

Guardar datos

Los datos se crean, eliminan y modifican en la base de datos usando instancias de las clases de entidad. Podría agregar código como el siguiente ejemplo (con datos simulados, en este caso) a los controladores de su API web.

```
var catalogItem = new CatalogItem() {CatalogTypeId=2, CatalogBrandId=2,
                                     Name="Roslyn T-Shirt", Price = 12};
_context.Catalog.Add(catalogItem);
_context.SaveChanges();
```

Inyección de dependencias en ASP.NET Core y controladores API Web

En ASP.NET Core puede usar Inyección de Dependencias (DI - *Dependency Injection*) desde el principio. No es necesario configurar un contenedor de inversión de control (IoC) de terceros, aunque puede conectar su contenedor de IoC preferido a la infraestructura ASP.NET Core si lo desea. En este caso, significa que puede inyectar directamente el **`DBContext`** de EF requerido o repositorios adicionales a través del constructor del controlador.

En el ejemplo anterior de la clase **`CatalogController`**, estamos inyectando un objeto de tipo **`CatalogContext`** más otros objetos a través del constructor **`CatalogController()`**.

El registro de la clase **DbContext** en el contenedor IoC del servicio, es una configuración fundamental del proyecto de la API web. Normalmente se hace en la clase **Startup** llamando al método **services.AddDbContext<DbContext>()** dentro del método **ConfigureServices()**, como se muestra en el siguiente ejemplo:

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

      services.AddDbContext<CatalogContext>(options =>
    {
        options.UseSqlServer(Configuration["ConnectionString"],
                                sqlServerOptionsAction: sqlOptions =>
                                {
                                    sqlOptions.
                                              MigrationsAssembly(
                                                     typeof(Startup).
                                                          GetTypeInfo().
                                                                 Assembly.
                                                                   GetName().Name);
                                    //Configuring Connection Resiliency:

                                    sqlOptions.
                                            EnableRetryOnFailure(maxRetryCount: 5,
                                            maxRetryDelay: TimeSpan.FromSeconds(30),
                                            errorNumbersToAdd: null);
                                });
        // Changing default behavior when client evaluation occurs to throw.
        // Default in EFCore would be to log warning when client evaluation is done.
        options.ConfigureWarnings(warnings => warnings.Throw(
                                    RelationalEventId.QueryClientEvaluationWarning));
    });
    //...
}
```

Recursos adicionales

- **Querying Data**
  https://docs.microsoft.com/ef/core/querying/index
- **Saving Data**
  https://docs.microsoft.com/ef/core/saving/index

La cadena de conexión a la base de datos y las variables de entorno usadas por los contenedores Docker

Puede usar la configuración de ASP.NET Core y agregar una propiedad **ConnectionString** a su fichero **settings.json** como se muestra en el siguiente ejemplo:

```
{
  "ConnectionString": "Server=tcp:127.0.0.1,5433;Initial Catalog=
      Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=Pass@word",
  "ExternalCatalogBaseUrl": "http://localhost:5101",
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

El fichero **settings.json** puede tener valores predeterminados para la propiedad **ConnectionString** o para cualquier otra propiedad. Sin embargo, esas propiedades serán reemplazadas al usar Docker, por los valores de las variables de entorno que especifique en el fichero **docker-compose.override.yml**.

Desde los ficheros **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 fichero **docker-compose.override.yml** (la cadena de conexión y otras líneas largas se pliegan a la línea siguiente en este ejemplo, pero no debe hacerlo en su fichero, es decir, la cadena de conexión es una sola línea larga).

```
# docker-compose.override.yml

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

Los ficheros **docker-compose.yml** a nivel de solución no sólo son más flexibles que los ficheros de configuración en el nivel de proyecto o microservicio, sino que también son más seguros. Tenga en cuenta que las imágenes Docker que compila por microservicio no contienen los ficheros **docker-compose.yml**, sólo ficheros binarios y ficheros de configuración para cada microservicio, incluido el fichero **Dockerfile**. Pero los ficheros **docker-compose.yml** y **docker-compose.override.yml** no se incluyen en la imagen Docker, no se despliegan junto con su aplicación, sólo se usan en el momento del despliegue.

Por lo tanto, poner valores de variables de entorno en los ficheros **`docker-compose.yml`** (incluso sin encriptar los valores) es más seguro que ponerlos en los ficheros de configuración de .NET regulares que se despliegan con su código.

Finalmente, puede obtener el valor configurado 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 otras formas de almacenar esos datos secretos, como las cadenas de conexión. Por lo general, eso será administrado por el orquestador elegido, como puede hacer con la administración de secretos de Docker Swarm.

Implementar versionamiento en ASP.NET Web APIs

A medida que cambian los requisitos del negocio, se pueden agregar nuevas colecciones de recursos, las relaciones entre los recursos pueden cambiar y la estructura de los datos en los recursos podría modificarse. La actualización de una API Web para gestionar nuevos requisitos es un proceso relativamente sencillo, pero debe tener en cuenta los efectos que dichos 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 podrían ser creadas por organizaciones de terceros que operan de forma remota.

El versionamiento permite que una API Web indique las características y los recursos que expone. Una aplicación cliente puede enviar peticiones a una versión específica de una característica o recurso. Existen varios enfoques para implementar el versionamiento:

- En el URI
- En el *query string*
- En el header

El versionamiento en el *query string* o URI son los más simples de implementar. El versionamiento en los encabezados es un buen enfoque. Sin embargo, este último no es tan explícito y sencillo como los otros dos. Dado que el versionamiento de URL es el más simple y explícito, es el que se usa en la aplicación de ejemplo eShopOnContainers.

Con el versionamiento por URI, como en la aplicación de ejemplo eShopOnContainers, cada vez que modifique la API web o cambie el esquema de recursos, debe cambiar el número de versión al URI para cada recurso. Los URI existentes deben continuar funcionando como antes, devolviendo recursos que se ajusten al esquema que coincida con la versión solicitada.

Como se muestra en el siguiente ejemplo, la versión se puede establecer utilizando el atributo **`Route`** en el controlador de la API Web, lo que hace que la versión sea explícita en el URI (**`v1`** en este caso).

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
   // Implementation ...
```

Este mecanismo de versionamiento es simple y depende de que el servidor enrute la petición al *endpoint* apropiado. Sin embargo, para un versionamiento más sofisticado, además de ser el mejor al usar REST, debe usar enlaces e implementar HATEOAS (Hypertext as the Engine of Application State).

Recursos adicionales

- **Scott Hanselman. ASP.NET Core RESTful Web API versioning made easy**
  http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx
- **Versioning a RESTful web API**
  https://docs.microsoft.com/azure/architecture/best-practices/api-design#versioning-a-restful-web-api
- **Roy Fielding. Versioning, Hypermedia, and REST**
  https://www.infoq.com/articles/roy-fielding-on-versioning

Generando metadata de descripción de Swagger para su API Web ASP.NET Core

Swagger es un *framework open source* muy utilizado y respaldado por un gran ecosistema de herramientas que le ayuda a diseñar, crear, documentar y consumir sus API RESTful. Se está convirtiendo en el estándar para el dominio de descripción de metadatos de APIs. Debería incluir metadatos de descripción de Swagger con cualquier tipo de servicio de microservicio, ya sean orientados a datos o microservicios más avanzados orientados por el dominio (como se explica en la siguiente sección).

El corazón de Swagger es la especificación Swagger, que es metadata de la descripción de un API en un fichero JSON o YAML. La especificación crea el contrato RESTful para su API, que detalla todos sus recursos y operaciones en un formato legible tanto por humanos como por programas, para facilitar el desarrollo, descubrimiento e integración de la API.

La especificación Swagger 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 compañías como Spotify, Uber, Slack y Microsoft, visite el sitio de Swagger (http://swagger.io).

¿Por qué usar Swagger?

Las razones principales para generar metadatos de Swagger para sus API son las siguientes.

Facilita que otros productos consuman e integren automáticamente sus API. Docenas de productos y herramientas comerciales, y muchas librerías y *frameworks* soportan Swagger. Microsoft tiene productos y herramientas de alto nivel que pueden consumir automáticamente APIs basadas en Swagger, como las siguientes:

- AutoRest. Puede generar automáticamente clases cliente .NET para llamar a Swagger. Esta herramienta se puede utilizar desde la CLI y también se integra con Visual Studio para facilitar su uso a través del IDE.
- Microsoft Flow. Puede usar e integrar automáticamente su API en un flujo de trabajo Microsoft Flow de alto nivel, sin necesidad de conocimientos de programación.

- Microsoft PowerApps. Puede consumir automáticamente su API desde las aplicaciones móviles PowerApp creadas con PowerApps Studio, sin necesidad de conocimientos de programación.
- Azure App Service Logic Apps. Puede usar e integrar automáticamente su API en una Azure App Service Logic App, sin necesidad de conocimientos de programación.

Permite generar automáticamente documentación de la API. Cuando crea API RESTful de gran escala, como en aplicaciones complejas basadas en microservicios, necesita manejar muchos *endpoints* con diferentes modelos de datos utilizados en las *payloads* (datos incluídos en) las peticiones y respuestas. Tener la documentación adecuada y tener un explorador de API robusto, como el que se 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.

Cómo automatizar la generación de la metadata de Swagger para un API con el paquete NuGet Swashbuckle

Generar metadatos Swagger manualmente (en un fichero JSON o YAML) puede ser un trabajo tedioso. Sin embargo, puede automatizar el descubrimiento de la API de los servicios de ASP.NET Web API, usando el paquete NuGet Swashbuckle para generar dinámicamente los metadatos Swagger de la API.

Swashbuckle genera automáticamente los metadatos Swagger para sus proyectos ASP.NET Web API. Soporta los proyectos ASP.NET Core Web API y los ASP.NET Web API del *framework* tradicional y cualquier otra variante, como las Azure API App, las Azure Mobile App, los microservicios de Azure Service Fabric basados en ASP.NET. También soporta APIs Web simples implementada en contenedores, como en la aplicación de referencia.

Swashbuckle combina el API Explorer y Swagger en swagger-ui para proporcionar una experiencia completa en documentación y descubrimiento para los consumidores de su API. Además de su generador de metadatos Swagger, Swashbuckle también contiene una versión integrada de swagger-ui, que funcionará automáticamente una vez que se instale Swashbuckle.

Esto significa que puede complementar su API con una interfaz de usuario agradable, para ayudar a los desarrolladores a descubrir y usar su API. Requiere una cantidad muy pequeña de código y de mantenimiento porque se genera automáticamente, lo que le permite concentrarse en crear su API. El resultado del explorador de APIs se parece al de la Figura 6-8.

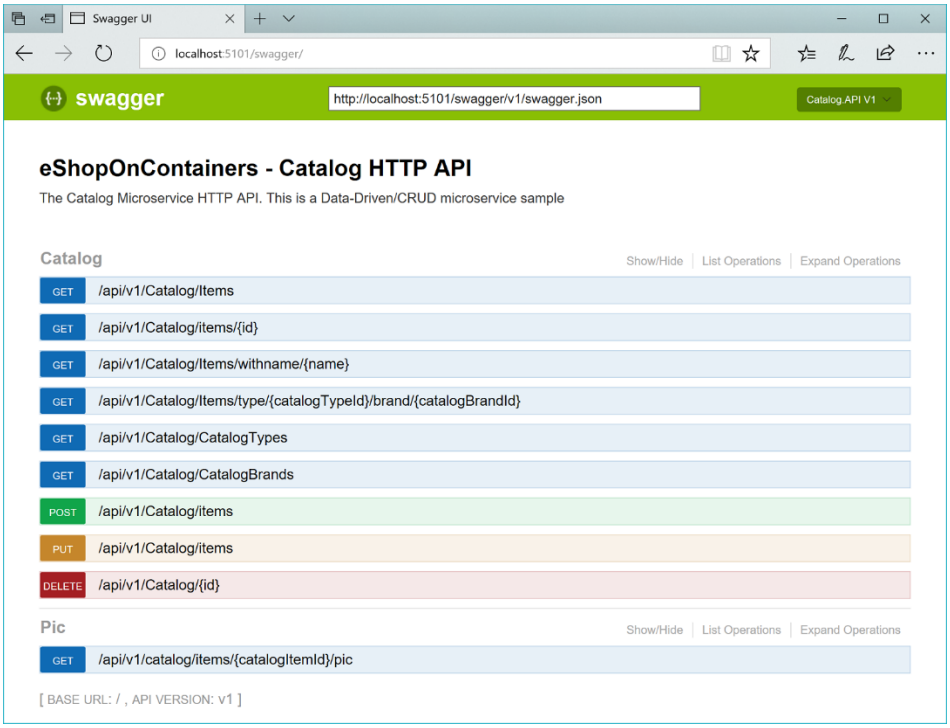


***Figura 6-8**. Explorador de APIs de Swashbuckle, basado en metadata Swagger del microservicio de catálogo de eShopOnContainers*

Sin embargo, el explorador de APIs no es lo más importante. Una vez que tiene una API web que se puede describir con los metadatos Swagger, su API se puede usar sin problemas desde las herramientas basadas en Swagger, incluidos los generadores de código de cliente que se pueden orientar a muchas plataformas. Por ejemplo, como se mencionó antes, AutoRest genera automáticamente clases de cliente .NET. Pero también están disponibles herramientas adicionales como swagger-codegen, que permiten la generación de código de librerías de cliente API, *stubs* de servidor y documentación automáticamente.

Actualmente, Swashbuckle consta de dos paquetes internos de NuGet bajo el metapaquete de alto nivel **`Swashbuckle.Swashbuckle.AspNetCoreSwaggerGen`** para aplicaciones ASP.NET Core.

Después de haber agregado la dependencia a estos paquetes NuGet en su proyecto de API Web, necesitará configurar Swagger en la clase **`Startup`**, como se muestra a continuación:

```
public class Startup
{
    public IConfigurationRoot Configuration { get; }

    // Other startup code...

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

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

       // Other ConfigureServices() code...
    }

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

        app.UseSwagger()
          .UseSwaggerUI(c =>
          {
              c.SwaggerEndpoint("/swagger/v1/swagger.json", "Catalog.API V1");

          });
    }
}
```

Una vez hecho esto, puede iniciar su aplicación y explorar los siguientes *endpoints* de JSON y la interfaz de usuario de Swagger, usando URLs como estas:

```
http://<your-root-url>/swagger/v1/swagger.json
http://<your-root-url>/swagger/
```

Anteriormente pudo ver la interfaz de usuario generada 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.

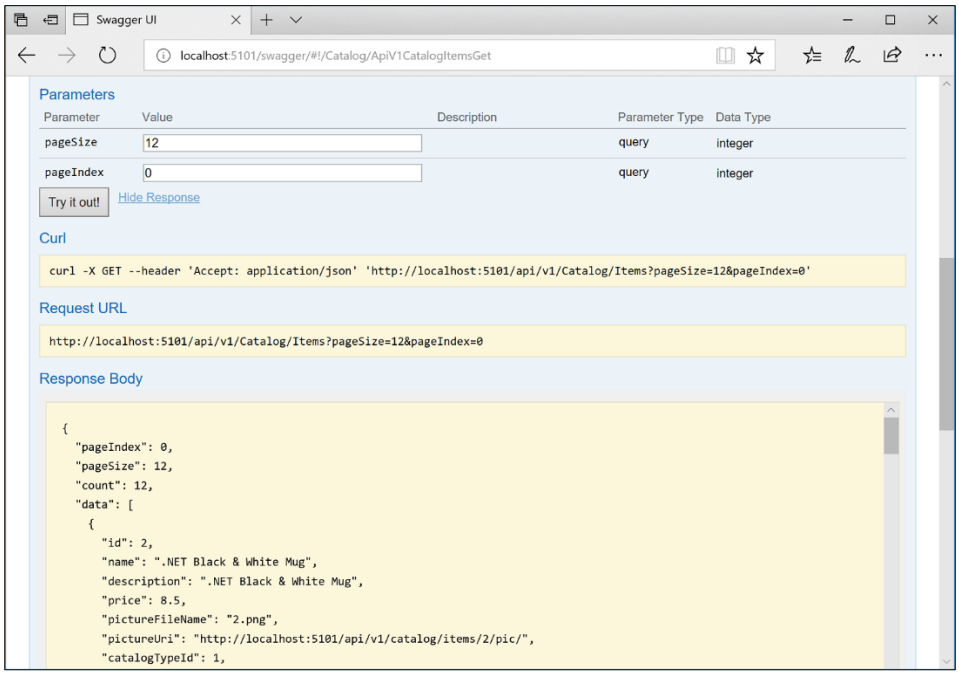


***Figura 6-9**. Probando el método Catalog/Item del API con Swashbuckle UI*

La figura 6-10 muestra los metadatos Swagger JSON generados a partir del microservicio eShopOnContainers (que es lo que las herramientas usan por debajo) cuando hace la petición a **`<su-url-raíz>/swagger/v1/swagger.json`** usando Postman.

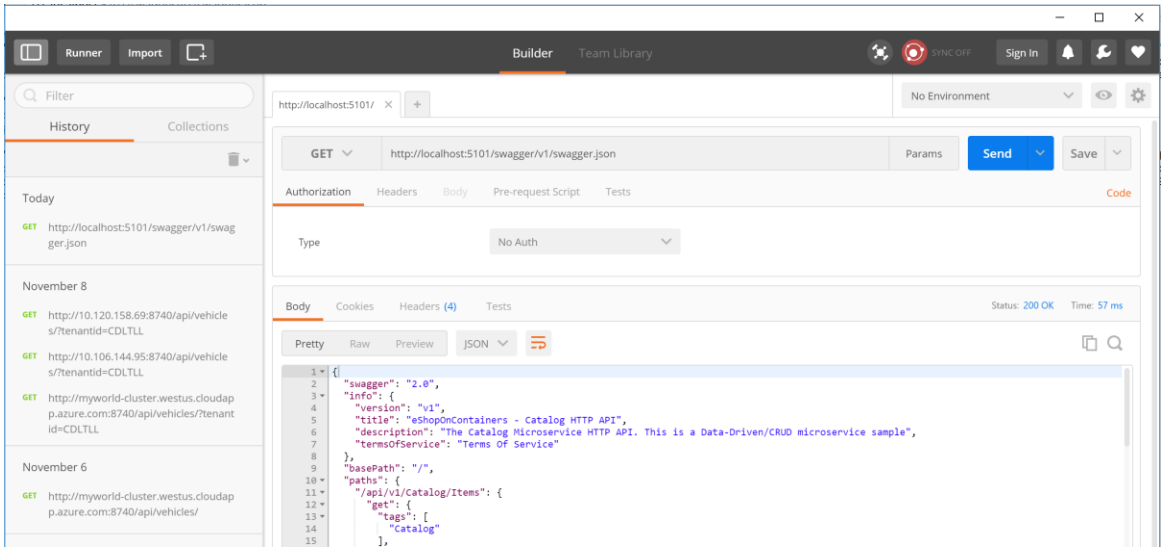


***Figura 6-10**. JSON de la metadara Swagger*

Es así de simple. Y como se genera automáticamente, los metadatos de Swagger se ampliarán cuando agregue más funcionalidades a su API.

Recursos adicionales

- **ASP.NET Web API Help Pages using Swagger**
  https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger

Definiendo su aplicación multi-contenedor con docker-compose.yml

En esta guía, se presentó el fichero **`docker-compose.yml`** en la sección Paso 4. Definir los servicios en docker-compose.yml al construir aplicaciones multi-contenedor. Sin embargo, hay otras formas de usar los ficheros **`docker-compose`** que vale la pena explorar con más detalle.

Por ejemplo, puede describir explícitamente cómo desea desplegar su aplicación de varios contenedores en el fichero **`docker-compose.yml`**. Opcionalmente, también puede describir cómo va a construir sus imágenes Docker personalizadas. (También se pueden construir imágenes Docker personalizadas con Docker CLI).

Básicamente, usted define cada uno de los contenedores que desea desplegar, más ciertas características para el despliegue de cada uno. Una vez que tenga un fichero con la descripción del despliegue multi-contenedor, puede implementar toda la solución en una sola acción orquestada por el comando de la CLI **`docker-compose up`**, o puede desplegarla de forma transparente desde Visual Studio. De lo contrario, necesitaría usar la CLI de Docker para desplegar contenedor por contenedor en varios pasos, mediante el uso del comando **`docker run`** desde la interfaz de comandos. Por lo tanto, cada servicio definido en **`docker-compose.yml`** debe especificar exactamente una imagen o *build*. Otros parámetros son opcionales y análogos a los correspondientes del comando **`docker run`**.

El siguiente código YAML es la definición de un posible fichero global **`docker-compose.yml`** único, para el ejemplo eShopOnContainers. Este no es el fichero real de **`docker-compose`** de eShopOnContainers, es sólo una versión simplificada y consolidada en un único fichero, aunque esta no es la mejor manera de trabajar con ficheros **`docker-compose`**, como se explicará más adelante.

```
version: '2'
services:
  webmvc:
    image: eshop/webmvc
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
      - BasketUrl=http://basket.api
    ports:
      - "5100:80"
    depends_on:
      - catalog.api
      - ordering.api
      - basket.api
  catalog.api:
    image: eshop/catalog.api
    environment:
      - ConnectionString=Server=sql.data;Initial Catalog=CatalogData;
                                                  User Id=sa;Password=your@password
    expose:
      - "80"
    ports:
      - "5101:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
```

```
    depends_on:
      - sql.data
  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=sql.data;Database=Services.OrderingDb;
                                                    User Id=sa;Password=your@password
    ports:
      - "5102:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
    depends_on:
      - sql.data
  basket.api:
    image: eshop/basket.api
    environment:
      - ConnectionString=sql.data
    ports:
      - "5103:80"
    depends_on:
      - sql.data
  sql.data:
    environment:
      - SA_PASSWORD=your@password
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"
  basket.data:
    image: redis
```

El parámetro o nodo raíz en este fichero es **`services`**. Bajo ese nodo se definen los servicios que desea desplegar y ejecutar con comando **`docker-compose up`** o cuando despliega desde Visual Studio, usando ese fichero **`docker-compose.yml`**. En este caso, el fichero **`docker-compose.yml`** tiene definidos varios servicios, como se describen en la siguiente tabla.

Nombre del servicio docker-compose.yml	Descripción
`webmvc`	Contenedor que incluye la aplicación ASP.NET Core MVC que consumen los microservicios del *back-end*.
`catalog.api`	Contenedor que incluye el microservicio ASP.NET Core Web API del Catálogo
`ordering.api`	Contenedor que incluye el microservicio ASP.NET Core Web API del Pedidos
`sql.data`	Contenedor donde corre SQL Server para Linux, con las bases de datos de todos los microservicios
`basket.api`	Contenedor que incluye el microservicio ASP.NET Core Web API del Carrito de Compras
`basket.data`	Contenedor donde corre el servicio de cache REDIS con la base de datos del Carrito de Compras

Un contenedor para un servicio API Web sencillo

Enfocándonos en el contenedor del microservicio catalog.api, podemos ver que tiene una definición sencilla:

```
catalog.api:
  image: eshop/catalog.api
  environment:
    - ConnectionString=Server=catalog.data;Initial Catalog=CatalogData;
                                                    User Id=sa;Password=your@password
  expose:
    - "80"
  ports:
    - "5101:80"

  #extra hosts can be used for standalone SQL Server or services at the dev PC
  extra_hosts:
    - "CESARDLSURFBOOK:10.0.75.1"

  depends_on:
    - sql.data
```

Este servicio contenerizado tiene la siguiente configuración básica:

- Se basa en la imagen personalizada **`eshop/catalog.api`**. Por simplicidad, no hay parámetro **`build:`** para identificador la compilación en el fichero. Esto significa que la imagen debe haber sido previamente creada (con **`docker build`**) o haber sido descargada (con **`docker pull`**) de cualquier registro de Docker.

- Define una variable de entorno llamada **`ConnectionString`** con la cadena de conexión que usará Entity Framework para acceder a la instancia de SQL Server, con la base de datos del catálogo. En este caso, el mismo contenedor de SQL Server tiene 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 la base de datos de cada microservicio.

- El nombre del servidor de SQL es **`sql.data`**, que es el mismo nombre del contenedor de SQL Server para Linux. Esto es conveniente, porque permite usar la resolución de nombres interna al *host* Docker, para resolver la dirección de red, por lo que no necesita conocer la IP interna de los contenedores a los que está accediendo desde otros.

  Como la cadena de conexión está definida por una variable de entorno, puede establecer 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 procesos de CI/CD en VSTS o su sistema DevOps preferido.

- Expone el puerto 80 para el acceso interno al servicio **`catalog.api`** dentro del *host* Docker. El *host* es realmente una máquina virtual Linux porque está basada en una imagen Docker para Linux, pero puede configurar el contenedor para que se ejecute en una imagen Windows.

- Envía el puerto 80 expuesto en el contenedor, al puerto 5101 en la máquina del *host* Docker (la máquina virtual Linux).

- Vincula el servicio web al servicio **`sql.data`** (el contenedor con la instancia de SQL Server sobre Linux). Cuando especifica esta dependencia, el contenedor **`catalog.api`** no se iniciará hasta que el contenedor **`sql.data`** haya arrancado, esto es importante porque **`catalog.api`** necesita tener la base de datos funcionando primero. Sin embargo, este tipo de dependencia de contenedor no es suficiente en muchos casos, porque Docker sólo verifica a nivel del contenedor. A veces, el servicio (en este caso, SQL Server) tarda en arrancar, por lo que es aconsejable implementar una lógica de reintento con retardo exponencial en los microservicios cliente. De esta forma, si una dependencia de un contenedor no está lista en un tiempo breve, la aplicación lo manejará y seguirá siendo resiliente.

- Está configurado para permitir el acceso a servidores externos: la configuración **`extra_hosts`** le permite acceder a servidores externos o máquinas fuera del *host* Docker (es decir, fuera de la máquina virtual Linux por defecto, que es un *host* Docker de desarrollo), como una instancia de SQL Server local en su máquina de desarrollo.

También hay otras configuraciones más avanzadas del **`docker-compose.yml`** que revisaremos más adelante.

Usando ficheros docker-compose para ejecutar en múltiples entornos

Los ficheros **`docker-compose.yml`** son ficheros de configuración y se pueden utilizar en muchas infraestructuras que entienden ese formato. La herramienta más sencilla es el comando **`docker-compose`**, pero otras como los orquestadores (por ejemplo, Docker Swarm) también entienden ese fichero.

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 **`docker-compose`** de la CLI, para crear ese entorno o usar Visual Studio, que usa **`docker-compose`** por debajo.

El fichero **`docker-compose.yml`** le permite configurar y documentar todas las dependencias de servicios 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 ficheros **`docker-compose.yml`** son interpretados por el motor Docker, pero también sirven para documentar la composición de su aplicación multi-contenedor.

Entornos de pruebas

Una parte importante de cualquier proceso de despliegue o integración continuos (CD/CI) son las pruebas unitarias y de integración. Estas pruebas automatizadas requieren un entorno aislado para que no se vean afectadas por los usuarios o cualquier otro cambio en los datos de la aplicación.

Con **docker-compose** puede crear y eliminar ese entorno aislado muy fácilmente, con unos pocos comandos desde la interfaz de comandos del sistema o con *scripts batch*, como los siguientes comandos:

```
docker-compose up -d
./run_unit_tests
docker-compose down
```

Entornos de producción

También puede usar Compose para desplegar en un motor Docker remoto. Un caso típico es implementar en una instancia única del *host* Docker (como un servidor o una máquina virtual de producción preparados con Docker Machine). Pero también podría ser un *cluster* Docker Swarm, porque los *clusters* también son compatibles con los ficheros **docker-compose.yml**.

Si está utilizando cualquier otro orquestador (Azure Service Fabric, Mesos DC/OS, Kubernetes, etc.), es posible que necesite agregar ajustes de configuración de metadatos y configuración como los de **docker-compose.yml**, pero en el formato requerido por el 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.

Usando múltiples ficheros docker-compose para manejar varios entornos

Al apuntar a diferentes entornos, debe usar varios ficheros *compose*. Esto le permite crear múltiples variantes de configuración dependiendo del entorno.

Reemplazando el fichero base docker-compose

Podría usar un fichero **docker-compose.yml** único, como en los ejemplos simplificados que se muestran en las secciones anteriores. Sin embargo, no se recomienda eso para la mayoría de las aplicaciones.

Docker Compose, por defecto, lee dos ficheros, un **docker-compose.yml** y un fichero opcional **docker-compose.override.yml**. Como se muestra en la Figura 6-11, cuando está usando Visual Studio y habilita la compatibilidad con Docker

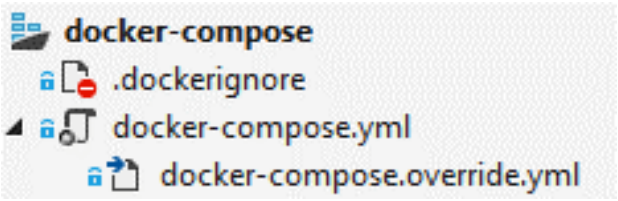


Figura 6-11. *Ficheros de docker-compose en Visual Studio 2017*

Puede editar los ficheros **docker-compose** con cualquier editor, como Visual Studio Code o Sublime, y ejecutar la aplicación con el comando **docker-compose up**.

Por convención, el fichero **`docker-compose.yml`** contiene su configuración base y otros parámetros estáticos. Eso significa que no debería cambiar la configuración del servicio según el entorno de donde se despliegue.

El fichero **`docker-compose.override.yml`**, como su nombre lo sugiere, contiene parámetros que reemplazan la configuración base, como los que dependen del entorno de despliegue. También puede tener múltiples ficheros de reemplazo con diferentes nombres. Los ficheros de reemplazo generalmente contienen información adicional que necesita la aplicación, pero específica para un entorno o para un despliegue.

Apuntando a múltiples entornos

Un caso de uso típico es cuando define múltiples ficheros de composición para apuntar a múltiples entornos, como producción, pre-producción (*staging*), integración continua (CI) o desarrollo. Para soportar estas diferencias, puede dividir su configuración de composición en varios ficheros, como se muestra en la figura 6-12.

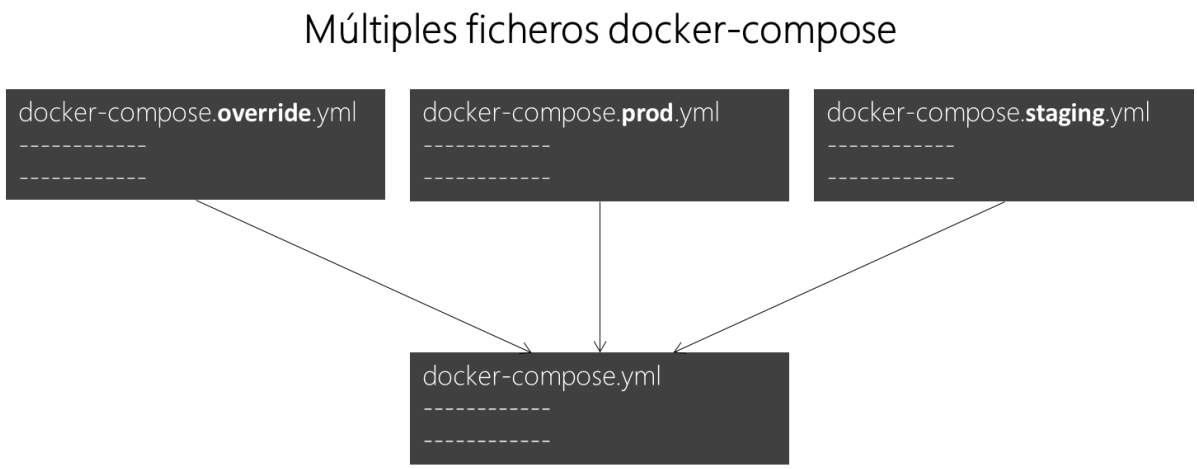


Figura 6-12. *Múltiples ficheros docker-compose pueden reemplazar valores del docker-compose.yml base*

Comience con el fichero base **`docker-compose.yml`**. Este fichero base debe contener la configuración básica o estática, que no cambia según el entorno. Por ejemplo, eShopOnContainers tiene el siguiente fichero **`docker-compose.yml`** (simplificado con menos servicios) como fichero base.

```
#docker-compose.yml (Base)
version: '3'
services:
  basket.api:
    image: eshop/basket.api:${TAG:-latest}
    build:
      context: ./src/Services/Basket/Basket.API
      dockerfile: Dockerfile
    depends_on:
      - basket.data
      - identity.api
      - rabbitmq

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

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

  webmvc:
    image: eshop/webmvc:${TAG:-latest}
    build:
      context: ./src/Web/WebMVC
      dockerfile: Dockerfile
    depends_on:
      - catalog.api
      - ordering.api
      - identity.api
      - basket.api
      - marketing.api

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

  nosql.data:
    image: mongo

  basket.data:
    image: redis

  rabbitmq:
    image: rabbitmq:3-management
```

Como ya dijimos, los parámetros en el fichero base **`docker-compose.yml`** no deberían cambiar por los diferentes entornos de despliegue.

Si, por ejemplo, se fija en la definición del servicio **`webmvc`**, puede ver cómo esa información es muy similar sin importar el entorno donde se despliegue. Usted tiene la siguiente información:

- El nombre del servicio: `webmvc`.
- La imagen particular del contenedor: `eshop/webmvc`.
- El comando para construir la imagen particular del contenedor, indicando cual fichero **`Dockerfile`** usar.
- Las dependencias con otros servicios, para que este contenedor no arranque hasta que las dependencias hayan arrancado.

Puede tener una configuración adicional, pero el punto importante es que en el fichero base **`docker-compose.yml`**, sólo debe establecer la configuración que es común a todos los entornos. Luego, en **`docker-compose.override.yml`** o ficheros similares para producción o *staging*, debe colocar la configuración específica para cada entorno.

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

```
#docker-compose.override.yml (Extended config for DEVELOPMENT env.)
version: '3'

services:
# Simplified number of services here:

  basket.api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_REDIS_BASKET_DB:-basket.data}
      - identityUrl=http://identity.api
      - IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - AzureServiceBusEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
      - UseLoadTest=${USE_LOADTEST:-False}

    ports:
      - "5103:80"

  catalog.api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_CATALOG_DB:-
Server=sql.data;Database=Microsoft.eShopOnContainers.Services.CatalogDb;User
Id=sa;Password=Pass@word}
```

```
      - PicBaseUrl=${ESHOP_AZURE_STORAGE_CATALOG_URL:-
http://localhost:5101/api/v1/catalog/items/[0]/pic/}
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - AzureStorageAccountName=${ESHOP_AZURE_STORAGE_CATALOG_NAME}
      - AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_CATALOG_KEY}
      - UseCustomizationData=True
      - AzureServiceBusEnabled=False
      - AzureStorageEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
    ports:
      - "5101:80"

  marketing.api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_MARKETING_DB:-
Server=sql.data;Database=Microsoft.eShopOnContainers.Services.MarketingDb;User
Id=sa;Password=Pass@word}
      - MongoConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosql.data}
      - MongoDatabase=MarketingDb
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - identityUrl=http://identity.api
      - IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
      - CampaignDetailFunctionUri=${ESHOP_AZUREFUNC_CAMPAIGN_DETAILS_URI}
      - PicBaseUrl=${ESHOP_AZURE_STORAGE_MARKETING_URL:-
http://localhost:5110/api/v1/campaigns/[0]/pic/}
      - AzureStorageAccountName=${ESHOP_AZURE_STORAGE_MARKETING_NAME}
      - AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_MARKETING_KEY}
      - AzureServiceBusEnabled=False
      - AzureStorageEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
      - UseLoadTest=${USE_LOADTEST:-False}
    ports:
      - "5110:80"

  webmvc:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
      - BasketUrl=http://basket.api
      - LocationsUrl=http://locations.api
      - IdentityUrl=http://10.0.75.1:5105
      - MarketingUrl=http://marketing.api
      - CatalogUrlHC=http://catalog.api/hc
      - OrderingUrlHC=http://ordering.api/hc
      - IdentityUrlHC=http://identity.api/hc
      - BasketUrlHC=http://basket.api/hc
      - MarketingUrlHC=http://marketing.api/hc
      - PaymentUrlHC=http://payment.api/hc
```

```
      - UseCustomizationData=True
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
      - UseLoadTest=${USE_LOADTEST:-False}
    ports:
      - "5100:80"
  sql.data:
    environment:
      - MSSQL_SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
      - MSSQL_PID=Developer
    ports:
      - "5433:1433"
  nosql.data:
    ports:
      - "27017:27017"
  basket.data:
    ports:
      - "6379:6379"
  rabbitmq:
    ports:
      - "15672:15672"
      - "5672:5672"
```

En este ejemplo, la configuración de reemplazo del desarrollo expone algunos puertos al *host*, define variables de entorno con URLs de redireccionamiento y especifica cadenas de conexión para el entorno de desarrollo. Estas configuraciones son sólo para el entorno de desarrollo.

Cuando ejecuta **`docker-compose up`** (o ejecuta la aplicación desde Visual Studio), el comando lee los reemplazos automáticamente, como si fusionara ambos ficheros.

Supongamos que quiere otro fichero Compose para el entorno de producción, con diferentes valores de configuración, puertos o cadenas de conexión. Puede crear otro fichero de reemplazo, como el fichero **`docker-compose.prod.yml`**, con diferentes configuraciones y variables de entorno. Ese fichero puede almacenarse en un repositorio Git diferente o ser administrado y asegurado por un equipo diferente.

Cómo desplegar con un fichero de reemplazo específico

Para utilizar varios ficheros de reemplazo, o un fichero de reemplazo con otro nombre, puede usar la opción -f con el comando **`docker-compose`** y especificar los ficheros. Compose combina los ficheros en el orden en que se especifican en la línea de comandos. El siguiente ejemplo muestra cómo desplegar con ficheros de reemplazo.

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

Usando variables de entorno en ficheros docker-compose

Es conveniente, especialmente en entornos de producción, poder obtener parámetros de configuración a partir de variables de entorno, como hemos mostrado en ejemplos anteriores. Puede hacer referencia a una variable de entorno en sus ficheros **`docker-compose`** utilizando la sintaxis

${MY_VAR}. La siguiente línea de un fichero **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, dependiendo del entorno de su *host* (Linux, Windows, *cluster* de la nube, etc.). Sin embargo, un enfoque conveniente es usar un fichero ".env". Los ficheros **docker-compose** admiten la declaración de variables de entorno por defecto en el fichero ".env". Esos son los valores por defecto para las variables de entorno, pero pueden ser reemplazados por los valores que haya definido en cada uno de sus entornos (sistema operativo *host* o variables de entorno de su *cluster*). Coloque el fichero ".env" en la carpeta donde se ejecuta el comando **docker-compose**.

El siguiente ejemplo muestra el contenido de un fichero ".env", como el de la aplicación eShopOnContainers.

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

Docker-compose espera que cada línea en un fichero ".env" tenga el formato **<variable>=<valor>**.

Tenga en cuenta que los valores establecidos en el entorno de ejecución siempre reemplazan los valores definidos dentro del fichero ".env". De manera similar, los valores pasados a través de los argumentos de la línea de comando, también reemplazan los valores por defecto establecidos en el fichero ".env".

Recursos adicionales

- **Overview of Docker Compose**
  https://docs.docker.com/compose/overview/
- **Multiple Compose files**
  https://docs.docker.com/compose/extends/#multiple-compose-files

Construyendo imágenes Docker optimizadas para ASP.NET Core

Si está explorando acerca de Docker y .NET Core en la Internet, encontrará ficheros **Dockerfiles** que muestran lo fácil que es construir una imagen Docker al copiar su fuente en un contenedor. Estos ejemplos sugieren que, al usar una configuración simple, puede tener una imagen Docker con el entorno empaquetado con su aplicación. El siguiente ejemplo muestra un fichero **Dockerfile** simple dentro de esta línea de pensamiento.

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

Un **`Dockerfile`** como este funcionará. Sin embargo, puede optimizar sustancialmente sus imágenes, especialmente sus imágenes de producción.

En el modelo de contenedores y microservicios, está constantemente iniciando contenedores. La forma típica de usar contenedores no es reiniciando un contenedor suspendido, porque el contenedor es desechable. Los orquestadores (como Docker Swarm, Kubernetes, DC/OS o Azure Service Fabric) simplemente crean nuevas instancias de imágenes. Lo que esto significa es que tendría que optimizar, precompilando la aplicación durante la construcción, para que el proceso de creación de instancias sea más rápido. Cuando se inicia el contenedor, debería estar listo para funcionar. No debería restaurar y compilar en tiempo de ejecución, utilizando los comandos **`dotnet restore`** y **`dotnet build`** de la CLI de dotnet, como se muestra en muchos artículos de blog sobre .NET Core y Docker.

El equipo de .NET ha estado haciendo un trabajo importante para optimizar .NET Core y ASP.NET Core para correr en contenedores. No sólo .NET Core es un *framework* liviano con poco uso de memoria; el equipo se ha enfocado en el rendimiento de inicio y ha producido algunas imágenes Docker optimizadas, como la imagen microsoft/aspnetcore disponible en Docker Hub, en comparación con las imágenes normales de microsoft/dotnet o microsoft/nanoserver. La imagen de microsoft/aspnetcore proporciona una configuración automática de **`aspnetcore_urls`** para el puerto 80 y el caché de ensamblados **`pre-ngend`**. Ambas configuraciones dan como resultado un inicio más rápido.

Recursos adicionales

- **Building Optimized Docker Images with ASP.NET Core**
  https://blogs.msdn.microsoft.com/stevelasker/2016/09/29/building-optimized-docker-images-with-asp-net-core/

Construyendo aplicaciones desde un contenedor de integración continua (CI)

Otro beneficio de Docker es que puede construir su aplicación desde un contenedor preconfigurado, como se muestra en la Figura 6-13, por lo que no necesita crear una máquina o máquina virtual para construir su aplicación. Puede usar o probar ese contenedor ejecutándolo en su máquina de desarrollo. Pero lo que es aún más interesante es que puede usar el mismo contenedor de compilación desde su proceso de CI.

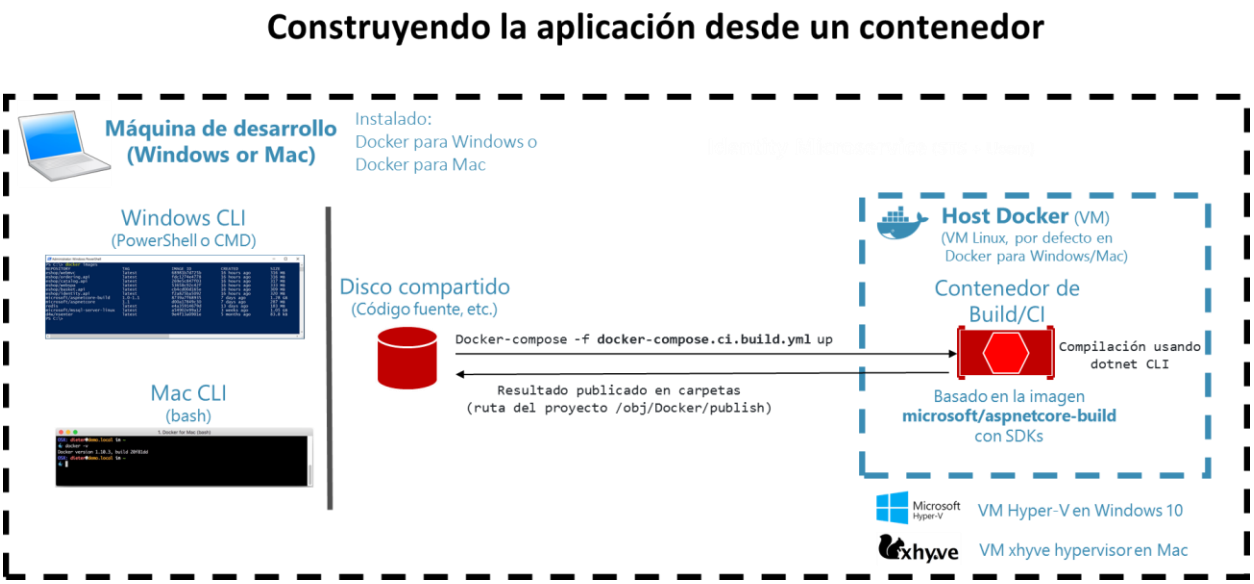


Figura 6-13. *Compilando la aplicación .NET en un contenedor de construcción*

Para este escenario, proporcionamos la imagen [microsoft/aspnetcore-build](), que puede usar para compilar y crear sus aplicaciones ASP.NET Core. El resultado se coloca en una imagen basada en la imagen [microsoft/aspnetcore](), que es una imagen optimizada del tiempo de ejecución, como se indicó anteriormente.

La imagen **aspnetcore-build** contiene todo lo que necesita para compilar una aplicación ASP.NET Core, que incluye .NET Core, ASP.NET SDK, npm, Bower, Gulp, etc.

Necesitamos estas dependencias en tiempo de compilación. Pero no queremos llevar esto en tiempo de ejecución, porque haría innecesariamente grande la imagen. En la aplicación eShopOnContainers, puede construir la aplicación desde un contenedor simplemente ejecutando el siguiente comando **docker-compose**.

```
docker-compose -f docker-compose.ci.build.yml up
```

La Figura 6-14 muestra este comando corriendo en la línea de comando.

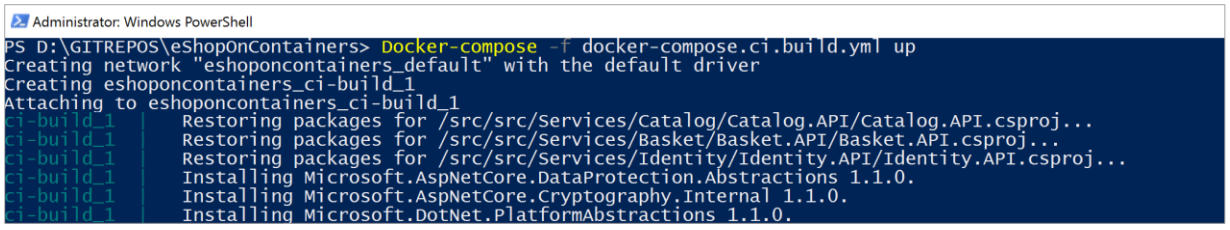


Figura 6-14. *Contruyendo la aplicación .NET desde un contenedor*

Como puede ver, el contenedor que está corriendo es **ci-build_1**. Este se basa en la imagen **aspnetcore-build** para que pueda compilar y construir toda su aplicación desde ese contenedor en lugar de desde su PC. Es por eso que en realidad está compilando y construyendo los proyectos .NET Core en Linux, porque ese contenedor se está ejecutando en el *host* Docker Linux por defecto.

El fichero docker-compose.ci.build.yml para esa imagen (parte de eShopOnContainers) contiene lo siguiente. Puede ver que inicia un contenedor de compilación utilizando la imagen `microsoft/aspnetcore-build`.

```
version: '3'

services:

  ci-build:

    image: microsoft/aspnetcore-build:2.0

    volumes:
      - .:/src

    working_dir: /src

    command: /bin/bash -c "pushd ./src/Web/WebSPA && npm rebuild node-sass && popd
&& dotnet restore ./eShopOnContainers-ServicesAndWebApps.sln && dotnet publish
./eShopOnContainers-ServicesAndWebApps.sln -c Release -o ./obj/Docker/publish"
```

Una vez que el contenedor de compilación está funcionando, ejecuta los comandos del .NET Core SDK **`dotnet restore`** y **`dotnet publish`** para compilar todos los proyectos en la solución y generar los ejecutables de .NET. En este caso, debido a que eShopOnContainers también tiene un SPA basado en TypeScript y Angular para el lado del cliente, también necesita verificar las dependencias de JavaScript con npm, pero esa acción no está relacionada con los ejecutables de .NET.

El comando **`dotnet publish`** construye y publica el resultado dentro de cada proyecto en la carpeta **`../obj/Docker/publish`**, como se muestra en la Figura 6-15.

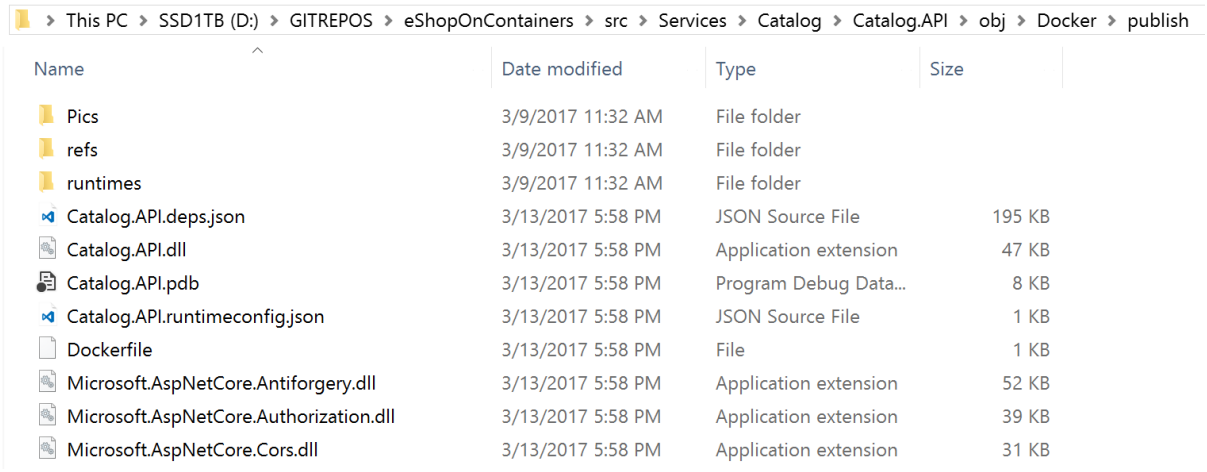


Figura 6-15. *Ficheros binarios generados por el comando* `dotnet publish`

Creando las imágenes Docker desde la CLI

Una vez que el resultado de la compilación se publica en las carpetas relacionadas (dentro de cada proyecto), el siguiente paso es construir realmente las imágenes Docker. Para hacer esto, use los comandos **`docker-compose build`** y **`docker-compose up`**, como se muestra en la Figura 6-16.

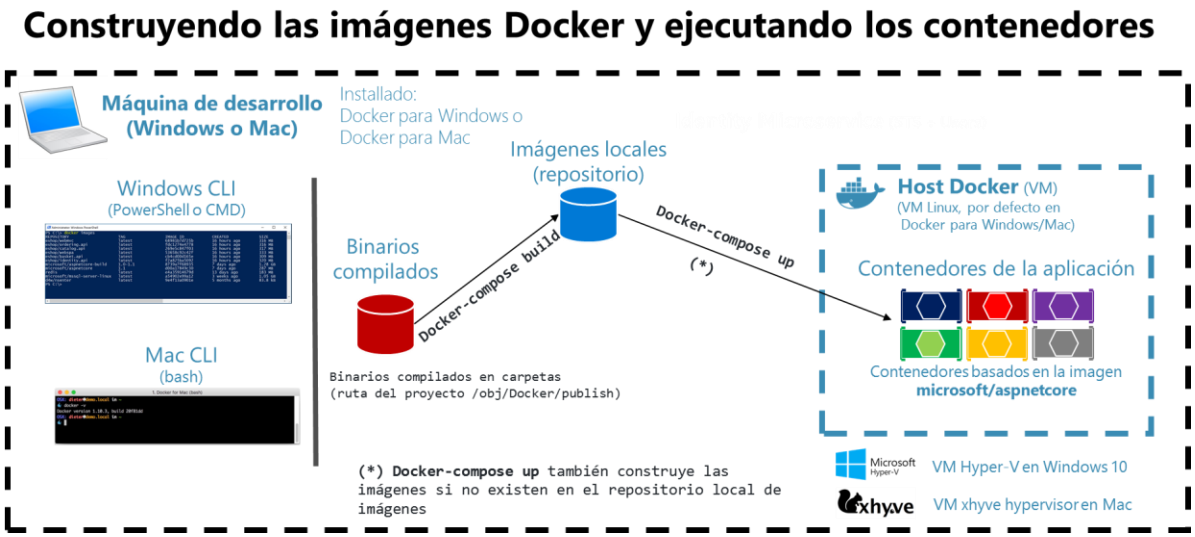


Figura 6-16. *Construyendo las imágenes Docker y ejecutando los contendores*

En la Figura 6-17, puede ver cómo se ejecuta el comando **`docker-compose build`**.

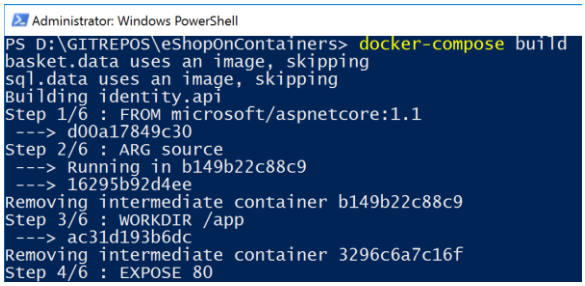


Figura 6-17*. Construyendo las imágenes Docker con el comando* `docker-compose build`

La diferencia entre los comandos **`docker-compose build`** y **`docker-compose up`** es que **`docker-compose up`** construye las imágenes e inicia los contenedores.

Cuando usa Visual Studio, todos estos pasos se realizan por debajo. Visual Studio compila su aplicación .NET, crea las imágenes Docker y despliega los contenedores en el *host* Docker. Visual Studio ofrece funciones adicionales, como la capacidad de depurar los contenedores que se ejecutan en Docker, directamente desde Visual Studio.

El aprendizaje general aquí es que puede construir su aplicación de la misma forma como lo haría su proceso de CI/CD, desde un contenedor en lugar de hacerlo desde una máquina local. Después de crear las imágenes, sólo necesita ejecutar las imágenes Docker con el comando **`docker-compose up`**.

Recursos adicionales

- **Building bits from a container: Setting the eShopOnContainers solution up in a Windows CLI environment (dotnet CLI, Docker CLI and VS Code)**
  https://github.com/dotnet/eShopOnContainers/wiki/03.-Setting-the-eShopOnContainers-solution-up-in-a-Windows-CLI-environment-(dotnet-CLI,-Docker-CLI-and-VS-Code)

Usando una base de datos corriendo en un contenedor

Puede tener sus bases de datos (SQL Server, PostgreSQL, MySQL, etc.) en servidores autónomos regulares, en *clusters* locales o en servicios PaaS en la nube como Azure SQL DB. Sin embargo, para entornos de desarrollo y pruebas, es conveniente que las bases de datos se ejecuten como contenedores, ya que no tiene ninguna dependencia externa y con simplemente ejecutar el comando **`docker-compose`** se inicia toda la aplicación. Tener esas bases de datos como contenedores también es excelente para las pruebas de integración, porque las bases de datos se inician en el contenedor y siempre se inicializan con los mismos datos de ejemplo, por lo que las pruebas pueden ser más predecibles.

SQL Server corriendo como contenedor con una base de datos de los microservicios

En eShopOnContainers, hay un contenedor llamado **`sql.data`** definido en el fichero **`docker-compose.yml`** que ejecuta SQL Server para Linux con las bases de datos necesarias para todos los microservicios. (También podría tener un contenedor SQL Server para cada base de datos, pero eso requeriría más memoria asignada a Docker.) El punto importante en microservicios es que cada microservicio posee sus datos relacionados, por lo tanto, su base de datos SQL relacionada en este caso. Pero las bases de datos pueden estar en cualquier sitio.

El contenedor de SQL Server en la aplicación de referencia se configura con el siguiente código YAML en el fichero **`docker-compose.yml`**, que se ejecuta al correr **`docker-compose up`**. Tenga en cuenta que el código YAML tiene información de configuración consolidada del fichero genérico **`docker-compose.yml`** y el fichero **`docker-compose.override.yml`**. (Aunque, en general, debería separar la configuración del entorno y la configuración de la base o estática, relacionada con la imagen de SQL Server, como mencionamos anteriormente).

```
sql.data:
  image: microsoft/mssql-server-linux
  environment:
    - MSSQL_SA_PASSWORD=Pass@word
    - ACCEPT_EULA=Y
    - MSSQL_PID=Developer
  ports:
    - "5434:1433"
```

De manera similar, en lugar de utilizar **`docker-compose`**, el siguiente comando **`docker run`** puede ejecutar ese contenedor:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD= your@password' -p 1433:1433 -d
microsoft/mssql-server-linux
```

Sin embargo, si está desplegando una aplicación de multi-contenedor como eShopOnContainers, es más conveniente usar el comando **`docker-compose up`**, para que despliegue 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 proporciona. Una vez que SQL Server se ejecuta como un contenedor, puede

actualizar la base de datos conectándose a través de cualquier conexión regular de SQL, como SQL Server Management Studio, Visual Studio o desde el código en C#.

La aplicación eShopOnContainers inicializa cada base de datos de microservicio, cargándola con datos de ejemplo al inicio, como se explica en la siguiente sección.

Tener SQL Server ejecutándose como un contenedor no solo es útil para una demostración donde es posible que no tenga acceso a una instancia de SQL Server. Como se indicó, también es ideal para entornos de desarrollo y prueba, de forma que pueda ejecutar fácilmente pruebas de integración a partir de una imagen limpia de SQL Server con datos iniciales conocidos.

Recursos adicionales

- **Run the SQL Server Docker image on Linux, Mac, or Windows**
  https://docs.microsoft.com/sql/linux/sql-server-linux-setup-docker
- **Connect and query SQL Server on Linux with sqlcmd**
  https://docs.microsoft.com/sql/linux/sql-server-linux-connect-and-query-sqlcmd

Inicializando con datos de prueba al arrancar la aplicación Web

Para cargar datos a la base de datos cuando se inicia la aplicación, puede agregar código como el siguiente al método **`Configure`** en la clase **`Startup`** del proyecto de la API Web:

```
public class Startup
{
    // Other Startup code...

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

        // Seed data through our custom class
        CatalogContextSeed.SeedAsync(app)
                          .Wait();

        // Other Configure code...
    }
}
```

El siguiente código en la clase **CatalogContextSeed** carga los datos iniciales.

```
public class CatalogContextSeed
{
    public static async Task SeedAsync(IApplicationBuilder applicationBuilder)
    {
        var context = (CatalogContext)applicationBuilder
            .ApplicationServices.GetService(typeof(CatalogContext));
        using (context)
        {
            context.Database.Migrate();

            if (!context.CatalogBrands.Any())
            {
                context.CatalogBrands.AddRange(
                    GetPreconfiguredCatalogBrands());

                await context.SaveChangesAsync();
            }
            if (!context.CatalogTypes.Any())
            {
                context.CatalogTypes.AddRange(
                    GetPreconfiguredCatalogTypes());

                await context.SaveChangesAsync();
            }
        }
    }
    static IEnumerable<CatalogBrand> GetPreconfiguredCatalogBrands()
    {
        return new List<CatalogBrand>()
        {
            new CatalogBrand() { Brand = "Azure"},
            new CatalogBrand() { Brand = ".NET" },
            new CatalogBrand() { Brand = "Visual Studio" },
            new CatalogBrand() { Brand = "SQL Server" }
        };
    }

    static IEnumerable<CatalogType> GetPreconfiguredCatalogTypes()
    {
        return new List<CatalogType>()
        {
            new CatalogType() { Type = "Mug"},
            new CatalogType() { Type = "T-Shirt" },
            new CatalogType() { Type = "Backpack" },
            new CatalogType() { Type = "USB Memory Stick" }
        };
    }
}
```

Para realizar pruebas de integración, es necesario tener una manera de generar datos consistentes con las pruebas. Es ideal poder crear todo desde cero, incluida una instancia de SQL Server ejecutándose en un contenedor.

La base de datos EF Core InMemory versus SQL Server en un contenedor

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

```
public class Startup
{
   // Other Startup code ...
   public void ConfigureServices(IServiceCollection services)
   {
       services.AddSingleton<IConfiguration>(Configuration);

       // DbContext using an InMemory database provider
       services.AddDbContext<CatalogContext>(opt => opt.UseInMemoryDatabase());

       //(Alternative: DbContext using a SQL Server provider
       //services.AddDbContext<CatalogContext>(c =>
       //{
       //    c.UseSqlServer(Configuration["ConnectionString"]);
       //
       //});
    }
    // Other Startup code ...
}
```

Sin embargo, hay un detalle importante. La base de datos *in-memory* no soporta muchas restricciones que son específicas de una base de datos en particular. Por ejemplo, puede agregar un índice único en una columna de su modelo EF Core y escribir una prueba en su base de datos *in-memory* para verificar que no le permite agregar un valor duplicado. Pero la base de datos *in-memory* no soporta índices únicos en una columna. Por lo tanto, este tipo de base de datos en memoria no se comporta exactamente igual que una base de datos real de SQL Server.

Aun así, una base de datos *in-memory* todavía es útil para pruebas y creación de prototipos. Pero si desea crear pruebas de integración precisas, que tengan en cuenta el comportamiento de una implementación específica de base de datos, debe usar una base de datos real como SQL Server. Para ello, ejecutar SQL Server en un contenedor es una muy buena opción y más precisa que el proveedor de base de datos EF Core *in-memory*.

Usando un servicio de cache Redis corriendo 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 sólo para sus máquinas de desarrollo local, sino también para sus entornos de prueba en sus procesos 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 una PaaS (Plataforma como servicio). En su código, solo necesita cambiar las cadenas de conexión.

Redis Labs proporciona una imagen Docker con Redis. Esa imagen está disponible desde Docker Hub en esta URL:

https://hub.docker.com/_/redis/

Puede ejecutar directamente un contenedor Docker Redis ejecutando el siguiente comando Docker CLI desde la interfaz de comandos del sistema:

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

La imagen de Redis incluye **expose:6379** (el puerto utilizado por Redis), por lo que la vinculación de contenedores estándar lo hará disponible automáticamente para los contenedores vinculados.

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

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

Este código en **docker-compose.yml** define un contenedor llamado **basket.data** basado en la imagen redis y publica internamente el puerto 6379, lo que significa que sólo será accesible desde otros contenedores que se ejecutan dentro del *host* Docker.

Finalmente, en el fichero **docker-compose.override.yml**, el microservicio **basket.api** para nuestra aplicación eShopOnContainers, define la cadena de conexión que se utilizará para ese contenedor Redis:

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

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

Como se mencionó anteriormente, cuando utiliza la comunicación basada en eventos, un microservicio publica un evento cuando sucede algo notable, como cuando actualiza una entidad del negocio. Otros microservicios se suscriben a esos eventos y cuando reciben, o se enteran de un evento, pueden actualizar sus propias entidades del negocio, lo que puede llevar a la publicación de más eventos. Esta es, precisamente, la base de la consistencia eventual. Este sistema de publicación/suscripción generalmente se realiza mediante el uso de un bus de eventos. El bus de eventos se puede diseñar como una interfaz con las APIs necesarias para publicar eventos y suscribirse a ellos o anular la suscripción. También puede tener una o más implementaciones basadas en cualquier comunicación entre procesos o mensajes, 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 del negocio que abarcan múltiples servicios, lo que le brinda consistencia eventual entre esos servicios. Una transacción eventualmente consistente está formada por una serie de acciones distribuidas. En cada acción, el microservicio actualiza una entidad del negocio y publica otro evento que desencadena la siguiente acción.

***Figura 6-18**. Comunicaciones asíncronas basadas en eventos, a través de un event bus*

Esta sección describe cómo puede implementar este tipo de comunicación con .NET utilizando 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 servicio *open source* de terceros o un *service bus* comercial.

Usando brokers de mensajes y buses de servicios para sistemas de producción

Como se señaló en la sección de arquitectura, puede elegir entre varias tecnologías de mensajería para implementar su bus de eventos abstractos. Pero estas tecnologías están en diferentes niveles de abstracción. Por ejemplo, RabbitMQ, un transporte basado en un *broker* de mensajería, está 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. La elección del producto depende de la cantidad de funciones y de la escalabilidad que ofrecen de entrada y lo que necesita para su aplicación.

Para implementar sólo una prueba de concepto de bus de evento 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 misión crítica y de producción que requieran alta escalabilidad, probablemente deba evaluar y usar Azure Service Bus.

Si necesita abstracciones de alto nivel y funciones más completas como Sagas para procesos de larga duración que faciliten el desarrollo distribuido, vale la pena evaluar otros buses de servicio comerciales y *open source* como NServiceBus, MassTransit y Brighter. En este caso, las abstracciones y las API que se usarían normalmente, serían las proporcionadas directamente por esos buses en lugar de sus propias abstracciones (como las abstracciones simples de bus de eventos provistas en eShopOnContainers). En ese caso, puede investigar el fork de eShopOnContainers usando NServiceBus (ejemplo implementado por *Particular Software*).

Por supuesto, siempre puede construir sus propias características particulares 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 insistir en el punto: las abstracciones e implementación del bus de eventos mostradas como ejemplo en la muestra de eShopOnContainers están destinadas a ser utilizadas sólo como una prueba de concepto. Una vez que haya decidido que desea tener una comunicación asíncrona y controlada por eventos, como se explica en esta sección, debe elegir el producto del 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. Esto se hace mediante la publicación de eventos de integración fuera del microservicio. Cuando se publica un evento para varios microservicios receptores (tantos como estén suscritos al evento de integración), el manejador de eventos apropiado en cada microservicio receptor se encarga de procesar el evento.

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

```
public class ProductPriceChangedIntegrationEvent : IntegrationEvent
{
    public int ProductId { get; private set; }
    public decimal NewPrice { get; private set; }
    public decimal OldPrice { get; private set; }

    public ProductPriceChangedIntegrationEvent(int productId, decimal newPrice,
                                               decimal oldPrice)
    {
        ProductId = productId;
        NewPrice = newPrice;
        OldPrice = oldPrice;
    }
}
```

Los eventos de integración se pueden definir en el nivel de aplicación de cada microservicio, para que estén desacoplados de otros microservicios, de forma similar a como se definen *ViewModels* en el servidor y el cliente. No se recomienda compartir una librería común de eventos de integración entre múltiples microservicios. Hacerlo sería acoplar esos microservicios con una sola librería de definición de eventos. Eso no es recomendable por las mismas razones por las que no desea compartir un modelo de dominio común en varios microservicios: los microservicios deben ser completamente autónomos.

Hay sólo algunos tipos de librerías que se deberían compartir entre de microservicios. Uno son las que conforman los bloques finales de la aplicación, como la API del cliente del Bus de Eventos, como en eShopOnContainers. Otros son las librerías que constituyen herramientas que también podrían compartirse como componentes NuGet, como los serializadores JSON, por ejemplo.

El bus de eventos

Un bus de eventos permite la comunicación de tipo publicación/suscripción entre microservicios sin que los componentes se tengan en cuenta explícitamente entre sí, como se muestra en la Figura 6-19.

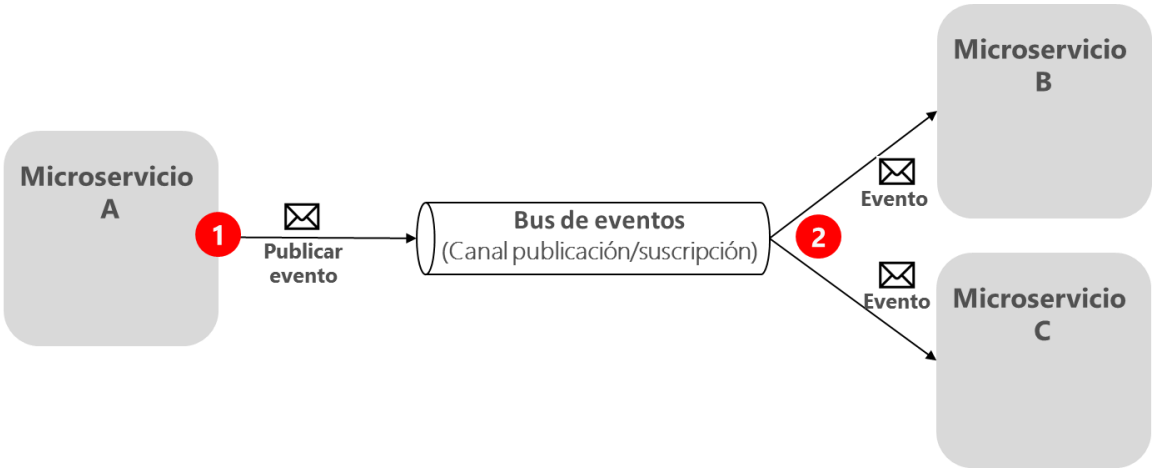


***Figura 6-19**. Aspectos básicos de publicación/suscripción con un bus de eventos*

El bus de eventos está relacionado con el patrón Observador y el patrón Publicación/Suscripción.

Patrón Observador

En el patrón Observador, el objeto primario (conocido como Observable) notifica a otros objetos interesados (conocidos como Observadores) con información relevante (eventos).

Patrón Publicación-Suscripción (Pub/Sub)

El propósito del patrón Publicación-Suscripción es el mismo que el patrón Observer: desea notificar a otros servicios cuando ocurren ciertos eventos. Pero hay una diferencia importante entre los patrones Observer y Pub/Sub. En el patrón del Observer, la transmisión se realiza directamente desde el observable al observador, así que ambos se "conocen". Pero cuando se usa un patrón Pub/Sub, hay un tercer componente, llamado *broker* o *message broker* o *event bus*, que es conocido tanto por el editor como por el suscriptor. Por lo tanto, cuando se utiliza el patrón Pub/Sub, el editor y los suscriptores se desacoplan, precisamente, gracias al mencionado bus de eventos o *message broker*.

El intermediario o bus de eventos

¿Cómo se logra el anonimato entre el que publica y el suscriptor? Una manera fácil es dejar que un intermediario se ocupe de todas las comunicaciones. Un bus de eventos es uno de esos intermediarios.

Un bus de eventos generalmente 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 varias implementaciones para que pueda intercambiarlas, según los requisitos del entorno (por ejemplo, entornos de producción o desarrollo).

En la Figura 6-20 puede ver una abstracción de un bus de eventos con implementaciones múltiples, basadas en distintas tecnologías de infraestructura de mensajería, como RabbitMQ, Azure Service Bus u otro intermediario de eventos/mensajes.

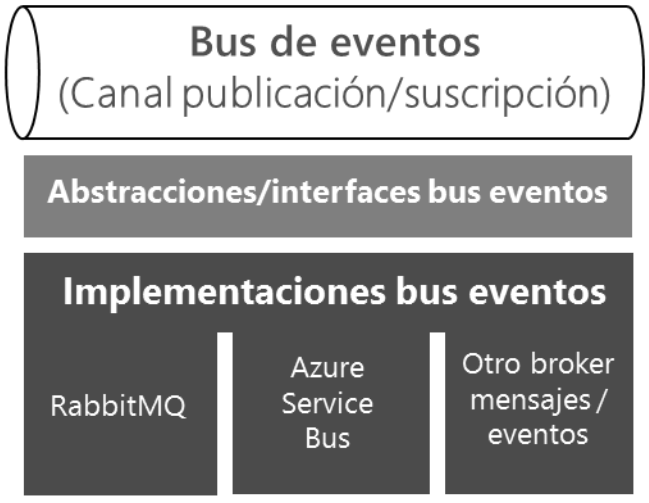


Figura 6- 20. *Múltiples implementaciones de un bus de eventos*

Sin embargo, como se mencionó anteriormente, usar sus propias abstracciones (la interfaz del bus de eventos) es bueno sólo si necesita funciones básicas soportadas por sus abstracciones. Si necesita funciones más complejas, probablemente debería utilizar la API y abstracciones proporcionadas por bus de eventos comercial, en lugar de sus propias abstracciones.

Definiendo la interfaz de un bus de eventos

Comencemos con una implementación de la interfaz del bus de eventos y otras posibles implementaciones, sólo para explorar un poco. La interfaz debe ser genérica y sencilla, como la siguiente.

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

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

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

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

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

El método **`Publish`** es sencillo. El bus de eventos transmitirá el evento de integración que le haya sido transferido a cualquier aplicación de microservicios, o incluso una aplicación externa, suscrita a ese evento. Este método es utilizado por el microservicio que está publicando el evento.

Los métodos **`Subscribe`** (puede tener varias implementaciones según los argumentos) son utilizados por los microservicios que desean recibir eventos. Este método tiene dos argumentos. El primero es el evento de integración al que se quiere suscribir (**`IntegrationEvent`**). El segundo argumento es el manejador de eventos de integración (o método *callback*), de tipo **`IIntegrationEventHandler<T>`**, para ser ejecutado cuando el microservicio receptor reciba ese mensaje de evento de integración.

Implementando un bus de eventos con RabbitMQ para entornos de desarrollo o pruebas

Deberíamos comenzar diciendo que, si crea su bus de eventos personalizado basado en RabbitMQ ejecutándose en un contenedor, como lo hace la aplicación eShopOnContainers, debería usarlo sólo para los entornos de desarrollo y pruebas. No debe usarlo para su entorno de producción, a menos que lo esté construyendo como parte de un bus de servicio orientado a producción. En un bus de eventos personalizado simple le pueden faltar muchas características críticas, que ya están listas en un bus de servicio comercial.

Una de las implementaciones personalizadas del bus de eventos en eShopOnContainers es básicamente una librería que utiliza la API de RabbitMQ (hay otra implementación basada en el bus de servicio de Azure).

La implementación del bus de eventos permite a los microservicios suscribirse a eventos, publicarlos y recibirlos, como se muestra en la Figura 6-21.

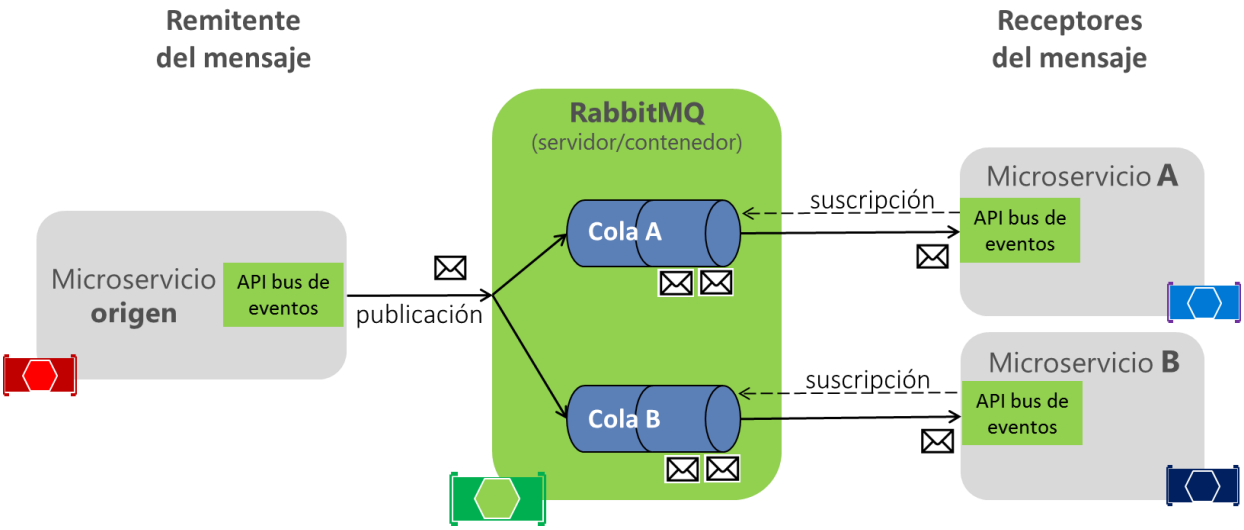


Figura 6-21. *Implementación de un bus de eventos con RabbitMQ*

En el código, la clase **`EventBusRabbitMQ`** implementa la interfaz genérica **`IEventBus`**. Esto se basa en Inyección de Dependencia para que pueda cambiar de esta versión de desarrollo/pruebas a una versión de producción.

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

La implementación con RabbitMQ de un bus de eventos para desarrollo/pruebas tiene un código muy básico. Tiene que gestionar la conexión con el servidor RabbitMQ y proporcionar un código para publicar un evento en las colas de mensajes. También tiene que implementar un diccionario de colecciones de manejadores de eventos de integración para cada tipo de evento; estos tipos de eventos pueden tener constructores y métodos de suscripción diferentes para cada microservicio receptor, como se muestra en la Figura 6-21.

Implementando un método Publish simple con RabbitMQ

El siguiente código es parte de una implementación simplificada de bus de eventos para RabbitMQ, aunque está mejorada en el código real de eShopOnContainers. Por lo general, no necesita cambiarlo a menos que esté realizando mejoras. El código obtiene una conexión y un canal a RabbitMQ, crea un mensaje y luego lo publica en la cola.

```
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods ...
    // ...
    public void Publish(IntegrationEvent @event)
    {
        var eventName = @event.GetType().Name;
        var factory = new ConnectionFactory() { HostName = _connectionString };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: _brokerName,
                                    type: "direct");

            string message = JsonConvert.SerializeObject(@event);
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish(exchange: _brokerName,
                                 routingKey: eventName,
                                 basicProperties: null,
                                 body: body);
        }
    }
}
```

El código real del método **`Publish`** en la aplicación eShopOnContainers se mejora usando una política de reintentos con Polly, que reintenta la tarea una cierta cantidad de veces en caso de que el contenedor RabbitMQ no esté listo. Esto puede ocurrir cuando **`docker-compose`** está iniciando los contenedores y, por ejemplo, el contenedor RabbitMQ arranque más lentamente que los otros contenedores.

Como se mencionó anteriormente, hay muchas configuraciones posibles en RabbitMQ, por lo que este código se debe usar sólo para entornos de desarrollo/pruebas.

Implementando las suscripciones con el API de RabbitMQ

Al igual que con el código de publicación, el siguiente código es una simplificación de una parte de la implementación del bus de eventos para RabbitMQ. De nuevo, generalmente no necesita cambiarlo a menos que lo esté mejorando.

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

    public void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIntegrationEventHandler<T>
    {
        var eventName = _subsManager.GetEventKey<T>();

        var containsKey = _subsManager.HasSubscriptionsForEvent(eventName);
        if (!containsKey)
        {
            if (!_persistentConnection.IsConnected)
            {
                _persistentConnection.TryConnect();
            }

            using (var channel = _persistentConnection.CreateModel())
            {
                channel.QueueBind(queue: _queueName,
                                  exchange: BROKER_NAME,
                                  routingKey: eventName);
            }

        }

        _subsManager.AddSubscription<T, TH>();
    }

}
```

Cada tipo de evento tiene un canal relacionado para obtener eventos de RabbitMQ. Puede tener tantos manejadores de eventos por canal y tipo de evento como sea necesario.

El método **`Subscribe`** acepta un objeto **`IIntegrationEventHandler`**, que es como un método *callback* en el microservicio actual, más su objeto relacionado **`IntegrationEvent`**. En el código se agrega ese manejador de eventos a la lista de manejadores que cada tipo de evento puede tener por cada microservicio cliente. Si el cliente no se ha suscrito al evento, se crea un canal para el tipo de evento para que pueda recibir eventos en un estilo *push* de RabbitMQ cuando ese evento se publica desde cualquier otro servicio.

Suscribiéndose a eventos

El primer paso para usar el bus de eventos es suscribir los microservicios a los eventos que desean recibir. Eso debería hacerse en los microservicios receptores.

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

Por ejemplo, al suscribirse al evento **ProductPriceChangedIntegrationEvent**, esto hace que el microservicio del carrito de compras tenga conocimiento de cualquier cambio en el precio del producto y le permite advertir al usuario sobre el cambio, si ese producto está en su carrito de compras.

```
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();

eventBus.Subscribe<ProductPriceChangedIntegrationEvent,
                   ProductPriceChangedIntegrationEventHandler>();
eventBus.Subscribe<OrderStartedIntegrationEvent,
                   OrderStartedIntegrationEventHandler>();
```

Después de ejecutar este código, el microservicio suscriptor estará escuchando a través de los canales de RabbitMQ. Cuando llega un mensaje de tipo **ProductPriceChangedIntegrationEvent**, el código invoca al manejador de eventos registrado, para que lo procese.

Publicando eventos a través del bus de eventos

Finalmente, el remitente del mensaje (el microservicio de origen) publica los eventos de integración con un código similar al del siguiente ejemplo. (Este es un ejemplo simplificado que no tiene en cuenta la atomicidad de la transacción). Implementaría un código similar cada vez que un evento se deba propagar a través de múltiples microservicios, generalmente justo después de confirmar datos o transacciones desde el microservicio de origen.

Primero, el objeto que implementa el bus de eventos (basado en RabbitMQ o en un bus de servicio) se inyectaría en el constructor del controlador, como en el siguiente código:

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _context;
    private readonly IOptionsSnapshot<Settings> _settings;
    private readonly IEventBus _eventBus;

    public CatalogController(CatalogContext context,
                             IOptionsSnapshot<Settings> settings,
                             IEventBus eventBus)
    {
        _context = context;
        _settings = settings;
        _eventBus = eventBus;
        // ...
    }
```

Luego lo usaría en el método del controlador, como en el método **UpdateProduct**:

```
[Route("update")]
[HttpPost]
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem product)
{
    var item = await _context.CatalogItems.SingleOrDefaultAsync(
                                                        i => i.Id == product.Id);
    // ...
    if (item.Price != product.Price)
    {
        var oldPrice = item.Price;
        item.Price = product.Price;
        _context.CatalogItems.Update(item);

        var @event = new ProductPriceChangedIntegrationEvent(item.Id,
                                                             item.Price,
                                                             oldPrice);
        // Commit changes in original transaction
        await _context.SaveChangesAsync();

       // Publish integration event to the event bus
       // (RabbitMQ or a service bus underneath)
       _eventBus.Publish(@event);
       // ...
```

En este caso, dado que el microservicio de origen es un CRUD simple, ese código se coloca directamente en un controlador de API Web.

En microservicios más avanzados, como cuando se utilizan enfoques CQRS, se puede implementar en la clase **CommandHandler**, en el método **Handle()**.

Diseñando la atomicidad y resiliencia al publicar en el bus de eventos

Cuando publica eventos de integración a través de un sistema de mensajería distribuida como un bus de eventos, existe el problema de actualizar de forma atómica la base de datos original y publicar un evento (es decir, que ocurran ambas operaciones o no ocurra ninguna). Asi, en el ejemplo simplificado

que mostramos 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 realizaran de forma atómica. Sin embargo, si está utilizando 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), esto no se recomienda por las razones descritas por el teorema CAP.

Básicamente, usamos microservicios para construir sistemas escalables y altamente disponibles. Simplificando algo el teorema de CAP, dice que no se puede construir una base de datos distribuida (o un microservicio con su modelo) que esté continuamente disponible, sea consistente y tolerante con cualquier partición. Debe elegir dos de estas tres propiedades.

En las arquitecturas basadas en microservicios, debe elegir la disponibilidad y la tolerancia y debe restarle importancia a la consistencia. Por lo tanto, en la mayoría de las aplicaciones modernas basadas en microservicios, generalmente se utilizan transacciones distribuidas en mensajes, como se hace al implementar transacciones distribuidas basadas en el Windows Distributed Transaction Coordinator (DTC) con MSMQ.

Volvamos al punto inicial y su ejemplo. Si el servicio falla después de que se actualiza 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. Esto podría ser crítico para el negocio, dependiendo de la operación específica con la que se encuentre.

Como se mencionó anteriormente en la sección de arquitectura, puede tener varios enfoques para manejar este problema:

- Usar el patrón Event Sourcing.
- Usar minería del registro de transacciones (transaction log mining).
- Usar el patrón Bandeja de Salida. Esta es una tabla transaccional que guarda los eventos de integración (extendiendo la transacción local).

Para este escenario, usar el patrón completo de *Event Sourcing* (ES) es uno de los mejores enfoques, sino el mejor. Sin embargo, en muchos escenarios, es posible que no pueda implementar un sistema ES completo. ES significa almacenar sólo eventos de dominio en su base de datos transaccional, en lugar de almacenar datos de estado actuales. Almacenar sólo 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 del pasado. Sin embargo, la implementación de un sistema completo de ES requiere que rediseñar la arquitectura del sistema y presenta muchas otras complejidades y requisitos. Por ejemplo, desearía utilizar una base de datos creada específicamente para el ES, como Event Store o una base de datos orientada a documentos como Azure Cosmos DB, MongoDB, Cassandra, CouchDB o RavenDB. *Event Sourcing* es un gran enfoque para este problema, pero no es la solución más fácil, a menos que ya esté familiarizado con el patrón.

La opción de usar *transaction log mining* inicialmente parece muy transparente. Sin embargo, para usar este enfoque, el microservicio debe estar acoplado al registro de transacciones de la RDBMS, como el registro de transacciones de SQL Server. Esto probablemente no es algo que quiera hacer. Otro inconveniente es que las actualizaciones de bajo nivel registradas en el registro de transacciones pueden no estar en el mismo nivel que sus 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 balanceado es la 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 se establece en el evento original cuando lo guarda en la tabla de eventos de integración. A continuación, intenta publicar el evento en el bus de eventos. Si la acción de publicar evento tiene éxito, inicie otra transacción en el servicio de origen y mueva el estado de "listo para publicar el evento" a "evento ya publicado".

Si falla la acción publicar-evento en el bus de eventos, los datos aún serán consistentes dentro del microservicio de origen, porque los eventos siguen marcados como "listo para publicar el evento" y, con respecto al resto de los servicios, eventualmente serán consistentes, cuando se reintente la publicación del evento. Para esto puede tener trabajos en segundo plano para verificar el estado de las transacciones o 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.

Observe que, con este enfoque, sólo persiste los eventos de integración para cada microservicio de origen y sólo los eventos que desea comunicar a otros microservicios o sistemas externos. Por el contrario, en un sistema ES completo, también se almacenarían todos los eventos de dominio.

Por lo tanto, este enfoque balanceado es un sistema ES simplificado. Necesita una lista de eventos de integración con su estado actual ("listo para publicar" frente a "publicado"). Pero sólo necesita implementar estos estados para los eventos de integración. Y en este enfoque, no necesita almacenar todos sus datos de dominio como eventos en la base de datos transaccional, como lo haría en un sistema ES completo.

Si ya está utilizando una base de datos relacional, puede usar una tabla transaccional para almacenar eventos de integración. Para lograr la atomicidad en su aplicación, utilice un proceso de dos pasos basado en transacciones locales. Básicamente, usar una tabla **`IntegrationEvent`** en la misma base de datos donde tiene las entidades del dominio. Esa tabla funciona como un seguro para lograr la atomicidad, para que incluya eventos de integración persistentes en las mismas transacciones que están guardando sus datos de dominio.

Paso a paso, el proceso es el siguiente: la aplicación comienza una transacción de base de datos local. Luego actualiza el estado de las entidades del dominio e inserta un evento en la tabla de eventos de integración. Finalmente, guarda la transacción. Así obtiene la atomicidad deseada.

Al implementar los pasos de publicación de eventos, tiene estas opciones:

- Publicar el evento de integración justo después de guardar la transacción y usar otra transacción local para marcar los eventos en la tabla como publicados. Luego, use la tabla sólo como un artefacto para rastrear los eventos de integración en caso de problemas en los otros microservicios y realice acciones compensatorias basadas en los eventos de integración almacenados.
- Usar la tabla como una especia de cola. Luego, un hilo o proceso diferente consulta la tabla de eventos de integración, publica los eventos en el bus de eventos y utiliza una transacción local para marcar los eventos como publicados.

La figura 6-22 muestra el primero de estos dos enfoques.

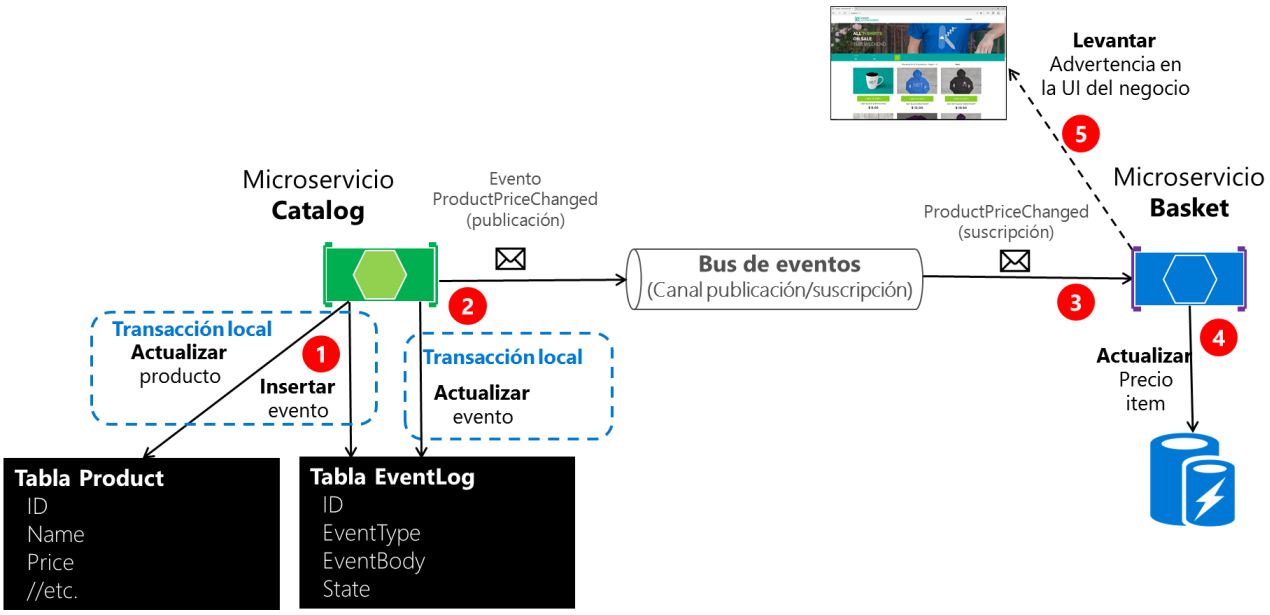


***Figura 6-22**. Atomicidad al puclicar eventos en el bus de datos*

Al enfoque mostrado en la Figura 6-22 le falta un microservicio *worker* (como tarea de fondo independiente) adicional, que se encarga de verificar y confirmar el éxito de los eventos de integración publicados. En caso de falla, ese microservicio de verificación adicional puede leer eventos de la tabla y volver a publicarlos, es decir, que repita el paso identificado con el número 2.

Acerca del segundo enfoque: utilice la tabla **`EventLog`** como una cola y use siempre un microservicio *worker* para publicar los mensajes. En ese caso, el proceso es como el que se muestra en la figura 6-23. Esta muestra el microservicio adicional y la tabla es la única fuente para publicación de eventos.

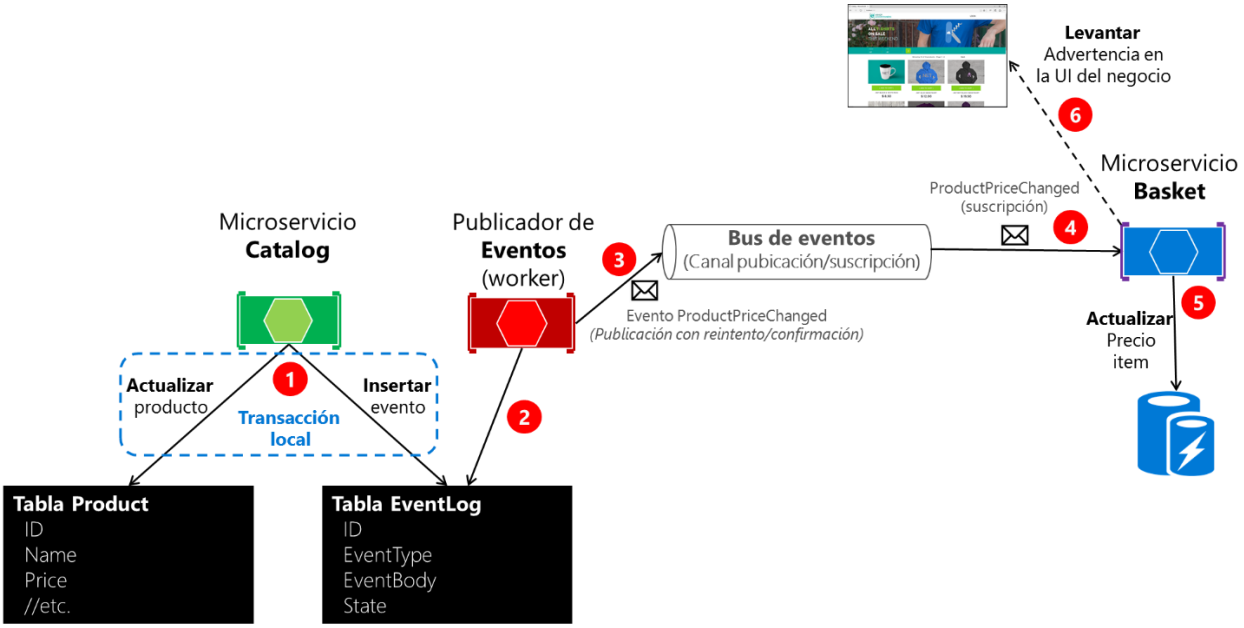


***Figura 6-23**. Atomicidad cuando se publican eventos en el bus de eventos con un microservicio worker*

Para simplificar, el ejemplo de eShopOnContainers utiliza el primer enfoque (sin procesos adicionales o microservicios de verificación) más el bus de eventos. Sin embargo, eShopOnContainers no está manejando todos los posibles casos de falla. En una aplicación real implementada en la nube, debe aceptar el hecho de que eventualmente surgirán problemas y debe implementar esa verificación y la lógica para volver a enviar. Usar la tabla como cola puede ser más efectivo que el primer enfoque, si tiene esa tabla como fuente única de eventos cuando los publica con el *worker* a través del bus de eventos.

Implementando la atomicidad al publicar eventos a través del bus de eventos

El siguiente código muestra cómo puede crear una transacción única que involucre varios **`DbContext`**: un contexto relacionado con los datos originales que se actualizan y el segundo contexto relacionado con la tabla **`IntegrationEventLog`**.

Tenga en cuenta que la transacción en el siguiente ejemplo no será resiliente 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 podría mover las bases de datos entre servidores. Para implementar transacciones resilientes en contextos múltiples, consulte la sección Implementando conexiones SQL resilientes con Entity Framework Core más adelante.

Para mayor claridad, el siguiente ejemplo muestra todo el proceso en un solo segmento de código. Sin embargo, en la implementación real de eShopOnContainers, se refactoriza y divide esta lógica en varias clases para que sea más fácil de mantener.

```
// Update Product from the Catalog microservice
//
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem
                                                                productToUpdate)
{
  var catalogItem =
       await _catalogContext.CatalogItems.SingleOrDefaultAsync(i => i.Id ==
                                                              productToUpdate.Id);
  if (catalogItem == null) return NotFound();

  bool raiseProductPriceChangedEvent = false;
  IntegrationEvent priceChangedEvent = null;
  if (catalogItem.Price != productToUpdate.Price)
          raiseProductPriceChangedEvent = true;

  if (raiseProductPriceChangedEvent) // Create event if price has changed
  {
      var oldPrice = catalogItem.Price;
      priceChangedEvent = new ProductPriceChangedIntegrationEvent(catalogItem.Id,
                                                          productToUpdate.Price,
                                                          oldPrice);
  }
  // Update current product
  catalogItem = productToUpdate;

  // Just save the updated product if the Product's Price hasn't changed.
  if !(raiseProductPriceChangedEvent)
  {
      await _catalogContext.SaveChangesAsync();
  }
  else  // Publish to event bus only if product price changed
  {
        // Achieving atomicity between original DB and the IntegrationEventLog
        // with a local transaction
        using (var transaction = _catalogContext.Database.BeginTransaction())
        {
                _catalogContext.CatalogItems.Update(catalogItem);
                await _catalogContext.SaveChangesAsync();

                // Save to EventLog only if product price changed
                if(raiseProductPriceChangedEvent)
                    await
       _integrationEventLogService.SaveEventAsync(priceChangedEvent);

                transaction.Commit();
        }

      // Publish the intergation event through the event bus
      _eventBus.Publish(priceChangedEvent);

      integrationEventLogService.MarkEventAsPublishedAsync(
                                                priceChangedEvent);
  }

  return Ok();
}
```

Después de crear el evento de integración **`ProductPriceChangedIntegrationEvent`**, la transacción que contiene la operación de dominio original (actualizar el elemento de catálogo) también incluye la persistencia del evento en la tabla **`EventLog`**. Esto lo convierte en una transacción única y siempre podrá verificar si se enviaron mensajes de eventos.

La tabla de registro de eventos se actualiza atómicamente con la operación de 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 consistencia entre las operaciones del dominio y los mensajes de eventos registrados en la tabla.

Recibiendo mensajes de las suscripciones: manejadores de eventos en los microservicios receptores

Además de la lógica de suscripción de eventos, también debe implementar el código para los manejadores de eventos de integración (similares a un método *callback*). En el manejador de eventos se especifica dónde se recibirán y procesarán los mensajes de eventos de un cierto tipo.

Un manejador de eventos primero recibe una instancia del **evento** desde el bus de eventos. Luego ubica el componente a procesar relacionado con ese evento de integración, propagando y persistiendo el evento como un cambio en el estado en el microservicio receptor. Por ejemplo, si un evento **ProductPriceChanged** se origina en el microservicio de catálogo, se maneja en el microservicio de carrito de compras y también cambia el estado en este microservicio receptor, como se muestra en el siguiente código.

```
//Integration-Event handler

Namespace Microsoft.eShopOnContainers.Services.Basket.
                                                    API.IntegrationEvents.EventHandling
{
    public class ProductPriceChangedIntegrationEventHandler :
                    IIntegrationEventHandler<ProductPriceChangedIntegrationEvent>
    {
        private readonly IBasketRepository _repository;

        public ProductPriceChangedIntegrationEventHandler(
                                                    IBasketRepository repository)
        {
            _repository = repository ??
                                throw new ArgumentNullException(nameof(repository));
        }

        public async Task Handle(ProductPriceChangedIntegrationEvent @event)
        {
            var userIds = await _repository.GetUsers();
            foreach (var id in userIds)
            {
                var basket = await _repository.GetBasket(id);
                await UpdatePriceInBasketItems(@event.ProductId, @event.NewPrice,
                                                basket);
            }
        }

        private async Task UpdatePriceInBasketItems(int productId, decimal newPrice,
                                                    CustomerBasket basket)
        {
            var itemsToUpdate = basket?.Items?.
                        Where(x => int.Parse(x.ProductId) == productId).ToList();

            if (itemsToUpdate != null)
            {
                foreach (var item in itemsToUpdate)
                {
                    if(item.UnitPrice != newPrice)
                    {
                        var originalPrice = item.UnitPrice;
                        item.UnitPrice = newPrice;
                        item.OldUnitPrice = originalPrice;
                    }
                }

                await _repository.UpdateBasket(basket);
            }
        }
```

```
        }
}
```

El manejador de eventos necesita verificar si el producto existe en alguna de las instancias del carrito de compras. También actualiza el precio del artículo para cada artículo relacionado en los carritos de compras. Finalmente, crea una alerta que se mostrará al usuario sobre el cambio de precio, como se muestra en la Figura 6-24.

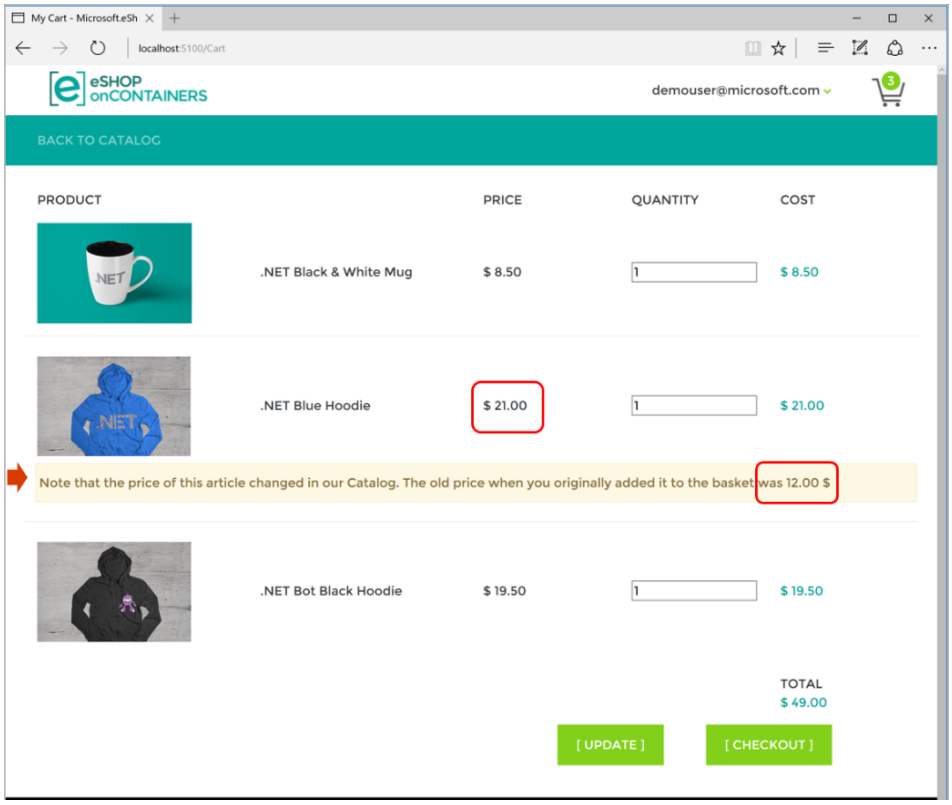


***Figura 6-24**. Mostrando el cambio de precio de un ítem, según lo indicado por eventos de integración*

Idempotencia en los eventos de actualización

Un aspecto importante de los eventos de mensajes de actualización es que una falla en cualquier punto de la comunicación debe provocar que se vuelva a intentar el mensaje. De lo contrario, una tarea en segundo plano podría intentar publicar un evento que ya ha sido publicado, creando una condición de carrera. Debe asegurarse de que las actualizaciones sean idempotentes o de que proporcionen información suficiente para garantizar que pueda detectar un duplicado, descartarlo y enviar sólo una respuesta.

Como se señaló anteriormente, idempotencia significa que una operación se pueda 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 receptor. Esto puede ser necesario debido a la naturaleza del evento en sí mismo o debido a la forma en que el sistema maneja el evento. La idempotencia del mensaje es importante en cualquier aplicación que utilice mensajes, no sólo en aplicaciones que implementan el patrón de bus de eventos.

Un ejemplo de operación idempotente es una declaración SQL que inserta datos en una tabla, sólo si esos datos no están ya en la tabla. No importa cuántas veces ejecute esa instrucción SQL de inserción, el resultado será el mismo: la tabla contendrá esa información. Idempotencia como esta también podría ser necesaria cuando se trata de mensajes si, por cualquier razón, los mensajes se pueden enviar y, por lo tanto, procesar más de una vez. Por ejemplo, si la lógica de reintento hace que un remitente envíe exactamente el mismo mensaje más de una vez, se debe asegurar de que sea idempotente.

Es posible diseñar mensajes idempotentes. Por ejemplo, puede crear un evento que diga "establecer el precio del producto a $25" en lugar de "agregar $5 al precio del producto". Puede procesar el primer mensaje varias 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 orden-completada que se propaga a múltiples suscriptores. Es importante actualizar la información de pedido en otros sistemas sólo una vez, incluso si hay eventos de mensaje duplicados para el mismo evento de orden-completada.

Es conveniente tener algún tipo de identidad de evento, para que pueda crear una lógica que asegure que cada evento se procese sólo una vez por receptor.

Algunos procesos de mensajes son intrínsecamente idempotentes. Por ejemplo, si un sistema genera imágenes miniaturas (*thumbnails*), puede no importar cuántas veces se procesa el mensaje sobre la *thumbnail* generada, el resultado es que las miniaturas se generan y son las mismas siempre. Por otro lado, operaciones como llamar a una pasarela de pago para cargar una tarjeta de crédito pueden no ser idempotentes en absoluto. En estos casos, se debe asegurar de que el procesamiento de un mensaje varias veces tenga el efecto que espera.

Recursos adicionales

- **Honoring message idempotency**
  https://msdn.microsoft.com/library/jj591565.aspx#honoring_message_idempotency

Desduplicando mensajes de eventos de integración

Puede asegurarse de que los eventos de mensajes se envíen y procesen una sola vez por suscriptor en diferentes niveles. Una forma es usar una característica de deduplicación ofrecida por la infraestructura de mensajería que esté utilizando. Otra es implementar lógica personalizada en su microservicio de destino. Tener validaciones tanto a nivel de transporte como a nivel de aplicación es su mejor apuesta.

Desduplicando eventos de mensaje a nivel del manejador de eventos

Una forma de asegurarse de que un evento sea procesado una sola vez por cualquier receptor, es identificando el evento para que el manejador pueda verificar si ya se procesó anteriormente. Por ejemplo, en la aplicación eShopOnContainers se usa ese enfoque, como puede ver en el **`código fuente de la clase UserCheckoutAcceptedIntegrationEventHandler`**. cuando recibe un comando **`CreateOrderCommand`**. (En este caso, "envolvemos" el comando **`CreateOrderCommand`** en un **`IdentifiedCommand`**, usando el **`eventMsg.RequestId`** como identificador, antes de enviarlo al manejador de comandos).

Desduplicando mensajes al usar RabbitMQ

Cuando se producen fallas intermitentes de la red, los mensajes pueden duplicarse y el receptor del mensaje debe estar listo para manejar esas duplicaciones. Si es posible, los receptores deben manejar los mensajes de una manera idempotente, lo cual es mejor que manejarlos explícitamente con desduplicación.

De acuerdo con la documentación de RabbitMQ, "si un mensaje se entrega a un consumidor (receptor) y luego se vuelve a poner en la cola (porque no fue confirmado antes de que fallara la conexión, por ejemplo), RabbitMQ marcará el mensaje como "entregado nuevamente" cuando se vuelva a entregar (ya sea el mismo consumidor o uno diferente).

Si el mensaje está marcado como "reentregado", el receptor debe tenerlo en cuenta, ya que el mensaje pudo haber sido procesado anteriormente. Pero eso no está garantizado, es posible que el mensaje nunca haya llegado al receptor después de que salió del intermediario de mensajes, tal vez debido a problemas de red. Por otro lado, si el mensaje no está marcado como "reentregado", se garantiza que el mensaje no se ha enviado más de una vez. Por lo tanto, el receptor necesita desduplicar mensajes o procesarlos de manera idempotente sólo si el mensaje está marcado como "reentregado".

Recursos adicionales

- **Forked eShopOnContainers using NServiceBus (Particular Software)**
  http://go.particular.net/eShopOnContainers
- **Event Driven Messaging**
  http://soapatterns.org/design_patterns/event_driven_messaging
- **Jimmy Bogard. Refactoring Towards Resilience: Evaluating Coupling**
  https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/
- **Publish-Subscribe channel**
  http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
- **Communicating Between Bounded Contexts**
  https://msdn.microsoft.com/library/jj591572.aspx
- **Eventual Consistency**
  https://en.wikipedia.org/wiki/Eventual_consistency

- **Philip Brown. Strategies for Integrating Bounded Contexts**
  http://culttt.com/2014/11/26/strategies-integrating-bounded-contexts/
- **Chris Richardson. Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 2**
  https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson
- **Chris Richardson. Event Sourcing pattern**
  http://microservices.io/patterns/data/event-sourcing.html
- **Introducing Event Sourcing**
  https://msdn.microsoft.com/library/jj591559.aspx
- **Event Store database**. Official site.
  https://eventstore.org/
- **Patrick Nommensen. Event-Driven Data Management for Microservices**
  https://dzone.com/articles/event-driven-data-management-for-microservices-1
- **The CAP Theorem**
  https://en.wikipedia.org/wiki/CAP_theorem
- **What is CAP Theorem?**
  https://www.quora.com/What-Is-CAP-Theorem-1
- **Data Consistency Primer**
  https://msdn.microsoft.com/library/dn589800.aspx
- **Rick Saling. The CAP Theorem: Why "Everything is Different" with the Cloud and Internet**
  https://blogs.msdn.microsoft.com/rickatmicrosoft/2013/01/03/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/
- **Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed**
  https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
- **Azure Service Bus. Brokered Messaging: Duplicate Detection**
  https://code.msdn.microsoft.com/Brokered-Messaging-c0acea25
- **Reliability Guide** (RabbitMQ documentation)
  https://www.rabbitmq.com/reliability.html#consumer

Probando servicios y aplicaciones web ASP.NET Core

Los controladores son una parte central de cualquier servicio API Web o aplicación Web MVC de ASP.NET Core. Por lo tanto, debe tener la seguridad de que se comportan según lo previsto. Las pruebas automatizadas le pueden proporcionar esta confianza y detectar errores antes de que lleguen a producción.

Debe probar cómo se comporta el controlador en función de entradas válidas o no válidas y probar las respuestas del controlador en función del resultado de la operación del negocio que realiza. Por lo tanto, debe tener este tipo de pruebas en sus microservicios:

- Pruebas unitarias. Esto garantiza que los componentes individuales de la aplicación funcionen como se esperaba. Las aserciones prueban la API de los componentes.
- Pruebas de integración. Estas aseguran que las interacciones de componentes funcionen como se espera contra artefactos externos como bases de datos. Las aserciones pueden probar la API de componentes, la interfaz de usuario o los efectos secundarios de acciones como la Entrada/Salida de base de datos, los registros, etc.
- Pruebas funcionales para cada microservicio. Estas aseguran que la aplicación funciona como se espera desde la perspectiva del usuario.
- Pruebas de servicio. Estas garantizan que se prueben los casos de uso del servicio, de punta a punta, incluyendo al mismo tiempo, las pruebas de los múltiples servicios necesarios. Para este tipo de prueba, primero debe preparar el entorno. En este caso, significa iniciar los servicios (por ejemplo, al usar **`docker-compose up`**).

Implementando pruebas unitarias para ASP.NET Core Web API

Las pruebas unitarias implican probar una parte de una aplicación, aislada de su infraestructura y dependencias. Cuando se prueba la lógica del controlador, sólo se prueba el contenido de una acción o método, no el comportamiento de sus dependencias o del propio *framework*. Las pruebas unitarias no detectan problemas en la interacción entre componentes, ese es el propósito de las pruebas de integración.

A medida que prueba las acciones de su controlador, asegúrese de enfocarse sólo en su comportamiento. Una prueba unitaria de controlador evita cosas como filtros, enrutamiento o *model binding* (el mapeo de los datos de la petición a un *ViewModel* o DTO). Debido a que se enfocan en probar sólo 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 generar demasiada carga.

Las pruebas unitarias se implementan en *frameworks* de pruebas como xUnit.net, MSTest, NUnit. Para la aplicación de ejemplo eShopOnContainers, estamos usando xUnit.

Cuando escriba una prueba unitaria para un controlador API Web, debe crear una instancia de la clase de controlador directamente usando la palabra **new** clave en C#, de modo que la prueba se ejecute lo más rápido posible. El siguiente ejemplo muestra cómo hacerlo al usar xUnit como el *framework* de pruebas.

```
[Fact]
public async Task Get_order_detail_success()
{
    //Arrange
    var fakeOrderId = "12";
    var fakeOrder = GetFakeOrder();

    //...

    //Act
    var orderController = new OrderController(
        _orderServiceMock.Object,
        _basketServiceMock.Object,
        _identityParserMock.Object);

    orderController.ControllerContext.HttpContext = _contextMock.Object;
    var actionResult = await orderController.Detail(fakeOrderId);

    //Assert
    var viewResult = Assert.IsType<ViewResult>(actionResult);
    Assert.IsAssignableFrom<Order>(viewResult.ViewData.Model);
}
```

Implementando pruebas de integración y funcionales para cada microservicio

Como se señaló, las pruebas de integración y funcionales tienen diferentes propósitos y objetivos. Sin embargo, la forma en que implementan ambas cuando prueba los controladores ASP.NET Core es similar, así que en esta sección nos concentraremos en las pruebas de integración.

Las pruebas de integración garantizan que los componentes de una aplicación funcionen correctamente cuando se ensamblan. ASP.NET Core soporta pruebas de integración usando *frameworks* de pruebas unitarias y un *host* web de pruebas (incluido en ASP.NET Core) que se puede usar para manejar peticiones sin tener que trabajar sobre la red.

A diferencia de las pruebas unitarias, las pruebas de integración con frecuencia involucran asuntos de infraestructura de las aplicaciones, como una base de datos, un sistema de ficheros, recursos de red o peticiones y respuestas web. En las pruebas unitarias se usan maquetas o simulaciones de objetos, para no tener que manejar esos asuntos de infraestructura. Pero el propósito de las pruebas de integración es confirmar que el sistema funciona como se espera con estos sistemas, así que para las pruebas de integración no se usan maquetas ni simulaciones. En cambio, se incluye infraestructura real, como el acceso a bases de datos o la invocación de servicios desde otros servicios.

Como las pruebas de integración abarcan mayores segmentos de código que las pruebas unitarias y debido a que las pruebas de integración dependen de elementos de infraestructura, tienden a ser órdenes de magnitud más lentas que las pruebas unitarias. Por lo tanto, es una buena idea limitar la cantidad de pruebas de integración se escriben y se ejecutan.

ASP.NET Core incluye un *host* web de pruebas, que se puede usar para manejar peticiones HTTP sin necesidad de pasar por la red, lo que significa que puede ejecutar esas pruebas más rápido que cuando se usa un servidor web real. El *host* web de prueba está disponible en un paquete NuGet como **`Microsoft.AspNetCore.TestHost`**. Puede agregarse a proyectos de prueba de integración y utilizarse para alojar aplicaciones ASP.NET Core.

Como puede ver en el siguiente código, cuando crea pruebas de integración para controladores ASP.NET Core, se crean instancias de los controladores a través del *host* de prueba. Esto es comparable a una solicitud HTTP, pero se ejecuta más rápido.

```
public class PrimeWebDefaultRequestShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;
    public PrimeWebDefaultRequestShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
                                            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }
    [Fact]
    public async Task ReturnHelloWorld()
    {
        // Act
        var response = await _client.GetAsync("/");
        response.EnsureSuccessStatusCode();

        var responseString = await response.Content.ReadAsStringAsync();

        // Assert
        Assert.Equal("Hello World!",
            responseString);
    }
}
```

Recursos adicionales

- **Steve Smith. Probar la lógica del controlador en ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/mvc/controllers/testing
- **Steve Smith. Pruebas de integración en ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/testing/integration-testing
- **Prueba unitaria de C# en .NET Core mediante pruebas de dotnet y xUnit**
  https://docs.microsoft.com/dotnet/articles/core/testing/unit-testing-with-dotnet-test

Implementando pruebas de servicios para aplicaciones multi-contenedor

Como se señaló anteriormente, cuando se prueban aplicaciones multi-contenedor, todos los microservicios se deben ejecutar dentro del *host* de Docker o del *cluster* de contenedores. Las pruebas de servicio de principio a fin, que incluyen múltiples operaciones que involucran varios microservicios, requieren que se despliegue e inicie toda la aplicación en el *host* Docker ejecutando **`docker-compose up`** (o un mecanismo similar si está utilizando un orquestador). Una vez que se ejecuta la aplicación completa y todos sus servicios, puede ejecutar pruebas de integración y funcionales de principio a fin.

Hay algunos enfoques que puede usar para esto. En el fichero **`docker-compose.yml`** que utiliza para desplegar la aplicación (o similares, como **`docker-compose.ci.build.yml`**), en el nivel de solución puede expandir el punto de entrada para usar `dotnet test`. También puede usar otro fichero de composición que ejecute sus pruebas en la imagen a la que apunta. Al usar otro fichero de composición para pruebas de integración, que incluya sus microservicios y bases de datos en contenedores, debe asegurarse de que los datos relacionados siempre se restablezcan a su estado original antes de ejecutar las pruebas.

Una vez que la aplicación compuesta esté en funcionamiento, puede aprovechar los *breakpoints* y las excepciones si está ejecutando Visual Studio. O puede ejecutar las pruebas de integración automáticamente en su proceso de CI en Visual Studio Team Services o cualquier otro sistema de CI/CD que soporte contenedores Docker.

Implementando tareas en segundo plano en microservicios con IHostedService y la clase BackgroundService

Las tareas en segundo plano y las tareas programadas son algo que quizás necesite implementar eventualmente, tanto en una aplicación basada en microservicios como en cualquier otro tipo de aplicación. La diferencia al usar una arquitectura de microservicios es que puede implementar un proceso/contenedor único para alojar estas tareas en segundo plano, de modo que pueda escalar hacia arriba/abajo según lo necesite o incluso puede asegurarse de ejecutar una sola instancia de ese proceso/contenedor de microservicios.

Desde un punto de vista genérico, en .NET Core llamamos a este tipo de tareas *Hosted Services*, porque son servicios/lógica que usted aloja en su *host*/aplicación/microservicio. Tenga en cuenta que, en este caso, el servicio alojado simplemente significa una clase con la lógica de tareas en segundo plano.

Desde .NET Core 2.0, el *framework* proporciona una nueva interfaz llamada **`IHostedService`** que le ayuda a implementar fácilmente los *hosted services*. La idea básica es que puede registrar varias tareas (*hosted services*), que se ejecutan en segundo plano, mientras está corriendo su *host* o *host web*, como se muestra en la imagen 8-25.

Figura 6-25. *Using IHostedService in a WebHost vs. a Host*

Note la diferencia que se hace entre **`WebHost`** y **`Host`**.

Un **`WebHost`** (clase base que implementa **`IWebHost`**) en ASP.NET Core 2.0 es el artefacto de infraestructura que utiliza para proporcionar funciones de servidor Http a su proceso, por ejemplo, si está implementando una aplicación web MVC o un servicio de API Web. Proporciona todas las nuevas bondades de la infraestructura en ASP.NET Core, lo que le permite utilizar la inyección de dependencias, insertar *middlewares* en el *request pipeline*, etc. y utilizar estos **`IHostedServices`** precisamente para las tareas en segundo plano.

Sin embargo, un **`Host`** (clase base que implementa **`IHost`**) es algo nuevo en .NET Core 2.1. Básicamente, un **`Host`** le permite tener una infraestructura similar a la que tiene con **`WebHost`** (inyección de dependencias, *hosted services*, etc.), pero en este caso, sólo necesita un proceso simple y más liviano como *host*, sin nada relacionado con las características de un servidor MVC, API Web o Http.

Por lo tanto, puede elegir y crear un proceso *host* especializado con **`IHost`** para manejar los servicios alojados y nada más, como un microservicio hecho sólo para alojar los **`IHostedServices`** o, alternativamente, puede extender un WebHost ASP.NET Core, tal como una aplicación ASP.NET Core Web API o MVC existente.

Cada enfoque tiene ventajas e inconvenientes, dependiendo de las necesidades del negocio y la escalabilidad. La conclusión básica es que si sus tareas de fondo no tienen nada que ver con HTTP (IWebHost) debería usar **`IHost`**, cuando esté disponible en. NET Core 2.1.

Registrando *hosted services* en un Host o WebHost

Vamos a profundizar 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 utiliza *hosted services*, pero también puede usarlo para cosas mucho más simples como:

- Una tarea en segundo plano para revisar una base de datos en busca de cambios
- Una tarea programada que actualiza algún caché periódicamente.
- Una implementación de **`QueueBackgroundWorkItem`** que permite que una tarea se ejecute en una cadena de fondo.
- Procesar mensajes de una cola de mensajes en segundo plano de una aplicación web, mientras se comparten servicios comunes como **`ILogger`**.

Básicamente puede descargar cualquiera de esas acciones a una tarea en segundo plano basada en **`IHostedService`**.

La forma de agregar uno o varios **IHostedServices** a su **WebHost** o **Host** es registrándolos mediante inyección de dependencias en un **WebHost** de ASP.NET Core (o en un **Host** de .NET Core 2.1). Básicamente, debe registrar los *hosted services* dentro del conocido método **ConfigureServices()** de la clase **Startup**, como en el siguiente código de un **WebHost** de ASP.NET Core típico.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    //Other DI registrations;

    // Register Hosted Services
    services.AddSingleton<IHostedService, GracePeriodManagerService>();
    services.AddSingleton<IHostedService, MyHostedServiceB>();
    services.AddSingleton<IHostedService, MyHostedServiceC>();
    //...
}
```

En ese código, el *hosted service* **GracePeriodManagerService** es código real del microservicio de pedidos de eShopOnContainers, mientras que los otros dos son sólo dos ejemplos adicionales.

La ejecución de la tarea en segundo plano **IHostedService** se coordina con la vida de la aplicación (*host* o microservicio, para el caso). Se registran las tareas cuando al iniciar la aplicación y tiene la oportunidad de realizar algún proceso de cierre ordenado o de limpieza cuando se termina la aplicación.

También podría iniciar un hilo en segundo plano para ejecutar cualquier tarea, sin utilizar **IHostedService**. La diferencia es precisamente en el momento de cierre de la aplicación cuando ese hilo simplemente se terminaría sin tener la oportunidad de ejecutar acciones de cierre ordenadas.

La interfaz IHostedService

Cuando registra un **IHostedService**, .NET Core llamará a los métodos **StartAsync()** y **StopAsync()** de su tipo **IHostedService** durante el inicio y la terminación de la aplicación, respectivamente. Específicamente, se invoca **StartAsync** después de que el servidor ha comenzado y antes de que se dispare **IApplicationLifetime.ApplicationStarted**.

IHostedService, tal como se define en .NET Core, tiene el siguiente aspecto.

```
namespace Microsoft.Extensions.Hosting
{
    //
    // Summary:
    //     Defines methods for objects that are managed by the host.
    public interface IHostedService
    {
        //
        // Summary:
        // Triggered when the application host is ready to start the service.
        Task StartAsync(CancellationToken cancellationToken);
        //
        // Summary:
        // Triggered when the application host is performing a graceful shutdown.
        Task StopAsync(CancellationToken cancellationToken);
    }
}
```

Como se puede imaginar, puede crear múltiples implementaciones de **`IHostedService`** y registrarlas en el contenedor de dependencias, en el método **`ConfigureServices()`**, como se ha mostrado anteriormente. Todos esos *hosted services* se iniciarán y se detendrán junto con la aplicación/microservicio.

Como desarrollador, usted es responsable de manejar la acción de terminación de sus servicios cuando el *host* dispara el método **`StopAsync()`**.

Implementando IHostedService con un *hosted service* derivado de la clase BackgroundService

También podría crear una clase como *hosted service* personalizado desde cero e implementar **`IHostedService`**, como se debe hacer al usar .NET Core 2.0.

Sin embargo, dado que la mayoría de las tareas en segundo plano tienen necesidades bastante similares con respecto a la gestión de tokens de cancelación y otras operaciones típicas, .NET Core 2.1 proporcionará una clase base abstracta muy conveniente de la que puede derivarse, denominada **`BackgroundService`**.

Esa clase proporciona el trabajo principal necesario para configurar la tarea de segundo plano. Tenga en cuenta que esta clase vendrá en la librería .NET Core 2.1, así que no tendrá que escribirla.

Sin embargo, al momento de escribir este documento, .NET Core 2.1 no ha sido liberado. Por lo tanto, en eShopOnContainers que actualmente está usando. NET Core 2.0, estamos incorporando temporalmente esa clase del repositorio *open-source* de .NET Core 2.1 (sin necesidad de ninguna licencia propietaria aparte de la licencia *open-source*) porque es compatible con la interfaz actual de **`IHostedService`** en .NET Core 2.0. Cuando se libere .NET Core 2.1, sólo tendrá que apuntar al paquete NuGet correcto.

El siguiente código es la clase base abstracta **BackgroundService** implementada en .NET Core 2.1.

```
// Copyright (c) .NET Foundation. Licensed under the Apache License, Version 2.0.
/// <summary>
/// Base class for implementing a long running <see cref="IHostedService"/>.
/// </summary>
public abstract class BackgroundService : IHostedService, IDisposable
{
    private Task _executingTask;
    private readonly CancellationTokenSource _stoppingCts =
                                                    new CancellationTokenSource();

    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);

    public virtual Task StartAsync(CancellationToken cancellationToken)
    {
        // Store the task we're executing
        _executingTask = ExecuteAsync(_stoppingCts.Token);

        // If the task is completed then return it,
        // this will bubble cancellation and failure to the caller
        if (_executingTask.IsCompleted)
        {
            return _executingTask;
        }

        // Otherwise it's running
        return Task.CompletedTask;
    }

    public virtual async Task StopAsync(CancellationToken cancellationToken)
    {
        // Stop called without start
        if (_executingTask == null)
        {
            return;
        }

        try
        {
            // Signal cancellation to the executing method
            _stoppingCts.Cancel();
        }
        finally
        {
            // Wait until the task completes or the stop token triggers
            await Task.WhenAny(_executingTask, Task.Delay(Timeout.Infinite,
                                                          cancellationToken));
        }

    }

    public virtual void Dispose()
    {
        _stoppingCts.Cancel();
    }
}
```

Cuando se deriva de la clase base abstracta anterior, gracias a esa implementación heredada, sólo necesita implementar el método **ExecuteAsync()** en su propia clase del *hosted service*, como en el siguiente código simplificado de eShopOnContainers, que supervisa una base de datos y publica eventos de integración en el bus de eventos cuando es necesario.

```
public class GracePeriodManagerService
       : BackgroundService
{
    private readonly ILogger<GracePeriodManagerService> _logger;
    private readonly OrderingBackgroundSettings _settings;

    private readonly IEventBus _eventBus;

    public GracePeriodManagerService(IOptions<OrderingBackgroundSettings> settings,
                                     IEventBus eventBus,
                                     ILogger<GracePeriodManagerService> logger)
        {
            //Constructor's parameters validations...
        }

        protected override async Task ExecuteAsync(CancellationToken stoppingToken)
        {
            _logger.LogDebug($"GracePeriodManagerService is starting.");

            stoppingToken.Register(() =>
                    _logger.LogDebug($" GracePeriod background task is stopping."));

            while (!stoppingToken.IsCancellationRequested)
            {
                _logger.LogDebug($"GracePeriod task doing background work.");

                // This eShopOnContainers method is quering a database table
                // and publishing events into the Event Bus (RabbitMS / ServiceBus)
                CheckConfirmedGracePeriodOrders();

                await Task.Delay(_settings.CheckUpdateTime, stoppingToken);
            }

            _logger.LogDebug($"GracePeriod background task is stopping.");

        }

        protected override async Task StopAsync (CancellationToken stoppingToken)
        {
              // Run your graceful clean-up actions
        }
}
```

En este caso específico para eShopOnContainers, se está ejecutando un método de la aplicación que consulta una tabla de base de datos buscando pedidos con un estado específico y al aplicar cambios, publica eventos de integración a través del bus de eventos (debajo se puede usar RabbitMQ o Azure Service Bus).

Por supuesto, puede ejecutar cualquier otra tarea del negocio en segundo plano, en su lugar.

Por defecto, el token de cancelación se establece con un tiempo de espera de 5 segundos, aunque puede cambiar ese valor al construir su **`WebHost`**, usando la extensión **`UseShutdownTimeout`** del **`IWebHostBuilder`**. Esto significa que se espera que nuestro servicio se cancele en 5 segundos, de lo contrario será terminado de forma más abrupta.

El siguiente código estaría cambiando ese tiempo a 10 segundos.

```
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...
```

Diagrama de clases resumen

La siguiente figura representa un resumen de las clases e interfaces involucradas al implementar **`IHostedServices`**.

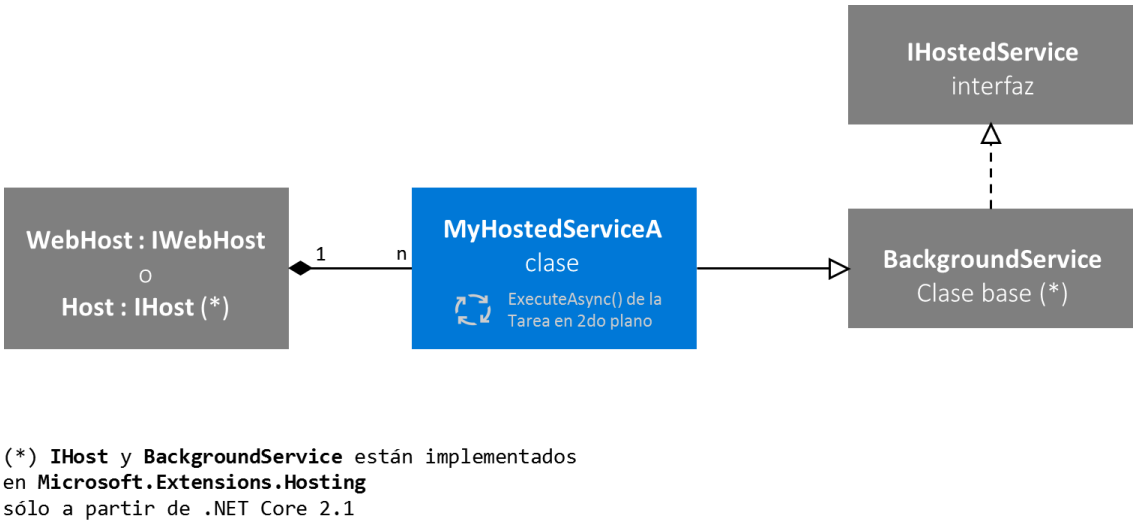


***Figura 6-26**. Diagrama de clases mostrando las clases e interfaces relacionadas con IHostedService*

Consideraciones de despliegue y puntos clave

Es importante tener en cuenta que la forma en que despliegue ASP.NET Core WebHost o .NET Core *Host* podría afectar la solución final. Por ejemplo, si despliega su WebHost en IIS o en un servicio de aplicaciones de Azure normal, su *host* puede terminar debido a los reinicios del *app pool*. Pero si está desplegando su *host* como un contenedor en un orquestador como Kubernetes o Service Fabric, puede controlar la cantidad garantizada de instancias activas de su *host*. Además, podría considerar otros enfoques en la nube especialmente diseñados para estos escenarios, como Azure Functions.

Pero incluso para un WebHost desplegado en un *app pool* (IIS), existen escenarios como el recargado (*repopulating*) o el vaciado (*flushing*) de la caché en memoria de la aplicación, que seguirían siendo aplicables.

La interfaz **IHostedService** proporciona una forma conveniente de iniciar tareas en segundo plano en una aplicación web ASP.NET Core (en .NET Core 2.0) o en cualquier proceso/*host* (comenzando en .NET Core 2.1 con **IHost**). Su principal beneficio es la oportunidad de tener un proceso de cierre ordenado para las tareas en segundo plano cuando el *host* se está apagando.

Recursos adicionales

- **Building a scheduled task in ASP.NET Core/Standard 2.0**
  https://blog.maartenballiauw.be/post/2017/08/01/building-a-scheduled-cache-updater-in-aspnet-core-2.html
- **Implementing IHostedService in ASP.NET Core 2.0**
  https://www.stevejgordon.co.uk/asp-net-core-2-ihostedservice
- **ASP.NET Core 2.1 Hosting samples**
  https://github.com/aspnet/Hosting/tree/dev/samples/GenericHostSample

Manejando la Complejidad del Negocio con los patrones DDD y CQRS

Visión

Diseñar un modelo de dominio para cada microservicio o Bounded Context que refleje la comprensión del dominio del negocio.

Esta sección se enfoca en los microservicios más avanzados que usted implementa cuando necesita manejar subsistemas complejos, o microservicios derivados del conocimiento de expertos de un dominio, con reglas del negocio en cambio constante. Los patrones de arquitectura utilizados en esta sección se basan en el diseño orientado por el dominio (DDD) y los enfoques de Segregación de las Responsabilidades de Comando y Consulta (CQRS), como se ilustra en la Figura 7-1.

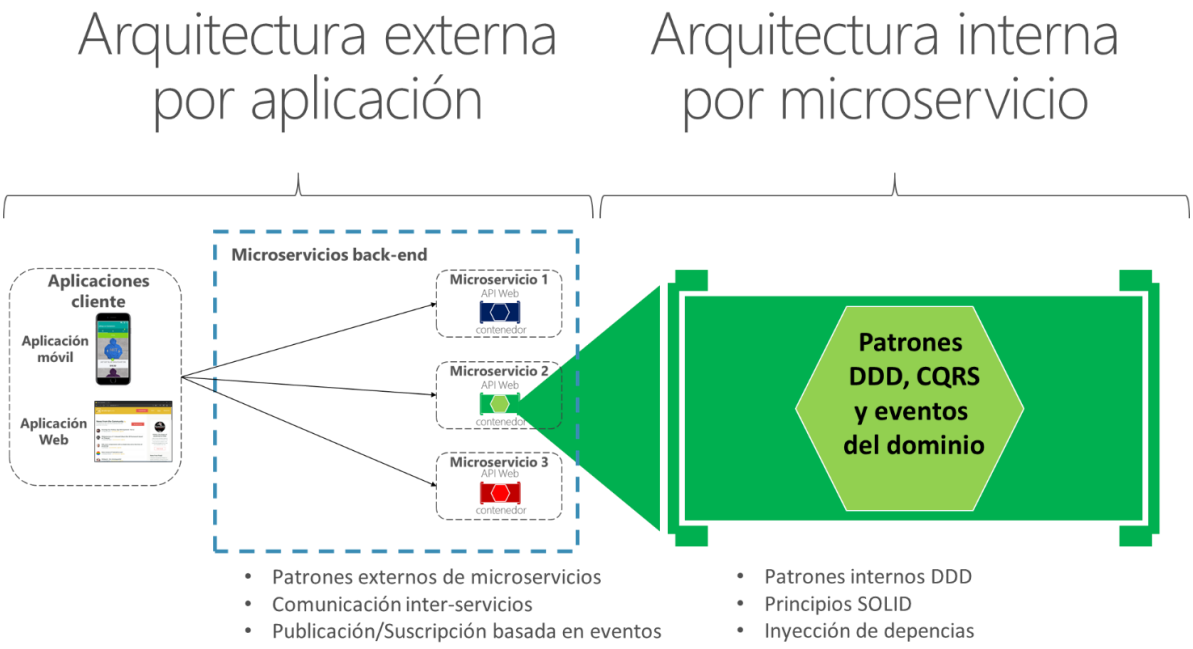


Figura 7-1. *Arquitectura externa de microservicios versus patrones arquitectónicos para cada microservicio*

Sin embargo, la mayoría de las técnicas usadas en microservicios de datos, como la implementación de un servicio ASP.NET Core Web API, o exponer los metadatos Swagger con Swashbuckle, también son aplicables a los microservicios más avanzados implementados con 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.

Esta sección primero proporciona detalles sobre los patrones simplificados CQRS usados en la aplicación de referencia eShopOnContainers. Más adelante, veremos una descripción general de las técnicas de DDD que le permiten encontrar patrones comunes que puede reutilizar en sus aplicaciones.

DDD es un tema amplio, con un gran 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 de DDD/CQRS. Pero, sobre todo, debe intentar aprender a aplicar las técnicas de DDD a partir de las conversaciones, los dibujos en las pizarras y las sesiones de modelado con los expertos en el dominio concreto de su negocio.

Recursos adicionales

DDD (Domain-Driven Design)

- **Eric Evans. Domain Language**
  http://domainlanguage.com/
- **Martin Fowler. Domain-Driven Design**
  http://martinfowler.com/tags/domain%20driven%20design.html
- **Jimmy Bogard. Strengthening your domain: a primer**
  https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/

Libros sobre DDD

- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software**
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/
- **Eric Evans. Domain-Driven Design Reference: Definitions and Pattern Summaries**
  https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-2014-09-22/dp/B01N8YB4ZO/
- **Vaughn Vernon. Implementing Domain-Driven Design**
  https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/
- **Vaughn Vernon. Domain-Driven Design Distilled**
  https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420/
- **Jimmy Nilsson. Applying Domain-Driven Design and Patterns**
  https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202/
- **Cesar de la Torre. N-Layered Domain-Oriented Architecture Guide with .NET**
  https://www.amazon.com/N-Layered-Domain-Oriented-Architecture-Guide-NET/dp/8493903612/
- **Abel Avram and Floyd Marinescu. Domain-Driven Design Quickly**
  https://www.amazon.com/Domain-Driven-Design-Quickly-Abel-Avram/dp/1411609255/
- **Scott Millett, Nick Tune - Patterns, Principles, and Practices of Domain-Driven Design**
  http://www.wrox.com/WileyCDA/WroxTitle/Patterns-Principles-and-Practices-of-Domain-Driven-Design.productCd-1118714709.html

Formación sobre DDD

- **Julie Lerman and Steve Smith. Domain-Driven Design Fundamentals**
  http://bit.ly/PS-DDD

Aplicando patrones simplificados de CQRS y DDD 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 originalmente definido por Bertrand Meyer en su libro *Object Oriented Software Construction*. La idea básica es que se pueden dividir las operaciones de un sistema en dos categorías muy diferentes:

- Consultas. devuelven un resultado, no cambian el estado del sistema y están libres de efectos secundarios.
- Comandos. Estos 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 cambia el estado, pero no ambos. Incluso un objeto del patrón de repositorio puede cumplir con CQS. CQS se puede considerar un principio fundamental para CQRS.

Command and Query Responsibility Segregation (CQRS) fue presentada por Greg Young y fuertemente promovida por Udi Dahan y otros. Se basa en el principio CQS, aunque es más detallado. Se puede considerar un patrón basado en comandos y eventos, además de, opcionalmente, mensajes asíncronos. En muchos casos, CQRS se relaciona 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 sólo almacenaría eventos en el modelo de dominio en lugar de almacenar

los datos del estado actual. Sin embargo, este no es el enfoque utilizado en esta guía. Estamos utilizando el enfoque más simple de CQRS, que consiste 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 (nótese que decimos modelo, no necesariamente una base de datos diferente) y está construido usando su propia combinación de patrones y tecnologías. Lo que es más importante, las dos capas pueden estar dentro del mismo nivel o microservicio, como en el ejemplo (microservicio de pedidos) utilizado para esta guía. O podrían implementarse en diferentes microservicios o procesos para que puedan optimizarse y escalarse por separado sin afectarse mutuamente.

CQRS significa tener dos objetos para una operación de lectura/escritura donde en otros contextos hay uno. Existen razones para tener una base de datos de consulta desnormalizada, sobre los que puede aprender en la literatura de CQRS más avanzada. Pero no estamos usando ese enfoque aquí, donde el objetivo es tener más flexibilidad en las consultas en vez de limitar las consultas a las restricciones de los patrones DDD como los agregados.

Un ejemplo de este tipo de servicio es el microservicio de pedidos de la aplicación eShopOnContainers. Este servicio implementa un microservicio basado en un enfoque simplificado de CQRS. Utiliza una fuente única de datos o base de datos, pero dos modelos lógicos y patrones DDD para el dominio transaccional, como se muestra en la Figura 7-2.

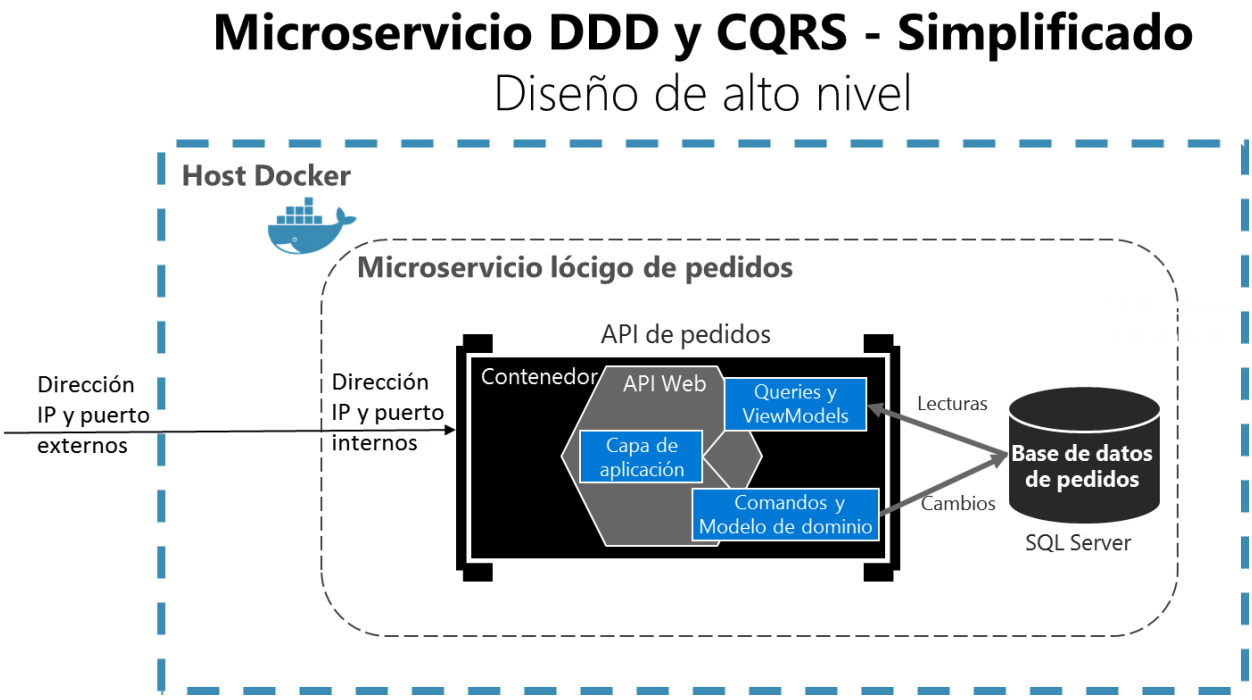


***Figura 7-2**. Microservicio basado en DDD y CQRS - Simplificado*

La capa de aplicación puede ser la API Web en sí misma. El aspecto importante del diseño aquí es que el microservicio ha dividido las consultas y *ViewModels* (modelos de datos especialmente creados para las aplicaciones cliente) de los comandos, modelo de dominio y transacciones que siguen el patrón CQRS. Este enfoque mantiene las consultas independientes de las limitaciones y restricciones

provenientes de los patrones DDD que sólo tienen sentido para las transacciones y actualizaciones, como se explica en secciones posteriores.

Aplicando los patrones CQRS y CQS en un microservicio DDD en eShopOnContainers

El diseño del microservicio de pedidos en la aplicación de referencia eShopOnContainers, se basa en los principios de CQRS. Sin embargo, utiliza el enfoque más simple, que es simplemente separar las consultas de los comandos y usar la misma base de datos para ambas acciones.

La esencia de esos patrones y el punto importante aquí, es que las consultas son idempotentes: no importa cuántas veces consulte un sistema, el estado de ese sistema no cambiará. Incluso podría usar un modelo de datos de "consulta" diferente al modelo que es "actualizado" por la lógica transaccional, aunque el microservicio de pedidos esté usando la misma base de datos. Por lo tanto, este es un enfoque simplificado de CQRS.

Por otro lado, los comandos, que activan transacciones y actualizaciones de datos, cambian de estado en el sistema. Hay que ser cuidadoso con los comandos, al manejar la complejidad y las reglas cambiantes del negocio. Aquí es donde se deben aplicar las técnicas de DDD para tener un sistema mejor modelado.

Los patrones DDD presentados en esta guía no deben aplicarse universalmente. Introducen restricciones en su diseño. Esas restricciones proporcionan beneficios tales como una mayor calidad en el tiempo, especialmente en los comandos y otras operaciones que modifican el estado del sistema. Sin embargo, esas restricciones agregan complejidad y limitaciones a la hora de leer y consultar los datos.

Uno de estos patrones es el patrón Agregado, que examinaremos con más detalle en secciones posteriores. En resumen, en el patrón Agregado, trata muchos objetos del dominio como una sola unidad, producto de sus relaciones en el dominio. Es posible que no siempre obtenga ventajas de este patrón en las consultas, de hecho, puede aumentar la complejidad de las consultas. Para las consultas no aporta ninguna ventaja el tratar varios objetos como un agregado único. Sólo agrega complejidad.

Como se muestra en la Figura 7-2, esta guía sugiere el uso de patrones DDD sólo en el área transaccional/actualizaciones de su microservicio (es decir, las que se producen a consecuencia de los comandos). Las consultas pueden seguir un enfoque más simple y deben separarse de los comandos, siguiendo un enfoque CQRS.

Para implementar el "lado de consulta", puede elegir entre muchos enfoques, desde un ORM completo como EF Core, proyecciones de AutoMapper, procedimientos almacenados, vistas, vistas materializadas o un micro ORM.

En esta guía y en eShopOnContainers (específicamente en el microservicio de pedidos) elegimos implementar consultas directas utilizando un micro ORM como Dapper. Esto le permite implementar cualquier consulta basada en sentencias de SQL para obtener el mejor rendimiento, gracias a un *framework* ligero y muy eficiente.

Tenga en cuenta que cuando utiliza este enfoque, cualquier actualización de su modelo que afecte a la persistencia de las entidades en una base de datos SQL también necesita actualizaciones separadas para las consultas SQL utilizadas por Dapper o cualquier otro enfoque separado (donde no use EF) para realizar consultas.

CQRS y los patrones DDD no son arquitecturas de alto nivel

Es importante entender que CQRS y la mayoría de los patrones DDD (como las capas DDD o un modelo de dominio con agregados) no son estilos arquitectónicos, sino sólo patrones de arquitectura. Microservicios, SOA y arquitectura basada en eventos (EDA) son ejemplos de estilos arquitectónicos. Describen un sistema de muchos componentes, como muchos microservicios. CQRS y los patrones DDD describen algo dentro de un solo sistema o componente, en este caso, algo dentro de un microservicio.

Bounded Contexts (BC) diferentes emplearán patrones diferentes. Tienen diferentes responsabilidades, y eso lleva a soluciones diferentes. Vale la pena enfatizar que forzar el mismo patrón en todas partes conduce al fracaso. No use CQRS y patrones DDD en todas partes. Muchos subsistemas, BC o microservicios son más simples y se pueden implementar más fácilmente usando servicios CRUD simples u otro enfoque.

Sólo hay una arquitectura de aplicación: la arquitectura del sistema o la aplicación completa, de punta a punta, que está diseñando (por ejemplo, la arquitectura de microservicios). Sin embargo, el diseño de cada *Bounded Context* o microservicio, dentro de esa aplicación, refleja sus propios compromisos y decisiones de diseño interno, en un nivel de **patrones** de arquitectura. No intente aplicar los mismos patrones arquitectónicos como CQRS o DDD en todas partes.

Recursos adicionales

- **Martin Fowler. CQRS**
  https://martinfowler.com/bliki/CQRS.html
- **Greg Young. CQS vs. CQRS**
  http://codebetter.com/gregyoung/2009/08/13/command-query-separation/
- **Greg Young. CQRS Documents**
  https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
- **Greg Young. CQRS, Task Based UIs and Event Sourcing**
  http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/
- **Udi Dahan. Clarified CQRS**
  http://udidahan.com/2009/12/09/clarified-cqrs/
- **Event-Sourcing (ES)**
  http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/

Implementando consultas en un microservicio CQRS

Para las lecturas/consultas, el microservicio de pedidos de la aplicación de referencia eShopOnContainers, implementa las consultas independientemente del modelo DDD y del área transaccional. Esto se hizo principalmente porque las exigencias de consultas y transacciones son drásticamente diferentes. Las actualizaciones implican transacciones que deben cumplir con la lógica del dominio. Las consultas, por otro lado, son idempotentes y se pueden distanciar de las reglas de dominio.

El enfoque es simple, como se muestra en la Figura 7-3. La API es implementada por los controladores de API Web, usando cualquier infraestructura (con un micro ORM como Dapper) y devolviendo *ViewModels* dinámicos, dependiendo de las necesidades de la interfaz de usuario.

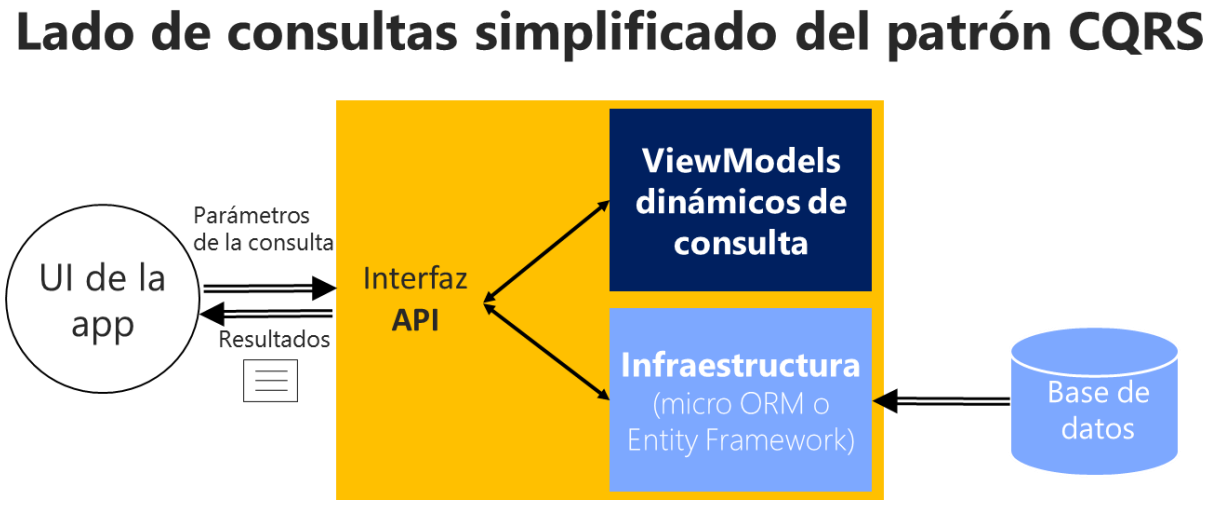


***Figura 7-3**. El enfoque más sencillo posible para consultas en un microservicio CQRS*

Este es el enfoque más simple posible para las consultas. Las definiciones de consulta acceden a la base de datos y devuelven un *ViewModel* dinámico, creado sobre la marcha, para cada consulta. Dado que las consultas son idempotentes, los resultados no cambiarán, sin importar cuántas veces ejecute una consulta. Por lo tanto, no necesita estar restringido por ningún patrón DDD utilizado en el lado de las transacciones, como agregados y otros patrones y es por eso que las consultas están separadas del área transaccional. Simplemente consulte la base de datos para obtener la información que necesita la interfaz de usuario y devuelva un *ViewModel* dinámico que no necesita estar declarado estáticamente en ningún lugar (no hay clases para los *ViewModels*) excepto en las propias sentencias SQL.

Dado que este es un enfoque simple, el código requerido en el lado de las consultas (como el código que usa un micro ORM como Dapper) se puede implementar dentro del mismo proyecto de API Web. La figura 7-4 muestra esto. Las consultas se definen en el proyecto de microservicio **Ordering.API** dentro de la solución eShopOnContainers.

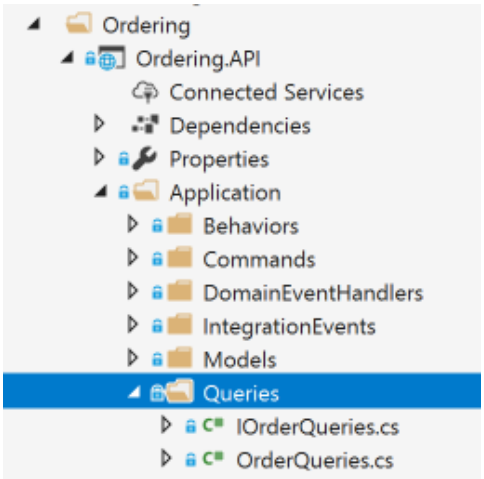


***Figura 7-4**. Las consultas en el microservicio de pedidos en eShopOnContainers*

Usando *ViewModels* hechos específicamente para las aplicaciones cliente, independientes de las restricciones del modelo de dominio

Dado que las consultas se realizan para obtener los datos que necesitan las aplicaciones cliente, el tipo devuelto se puede hacer específicamente para éstas, en función de los datos devueltos por las consultas. Estos modelos, o *Data Transfer Objects* (DTO), se llaman *ViewModels*.

Los datos devueltos (*ViewModel*) pueden ser el resultado de unir datos de múltiples entidades o tablas en la base de datos o, incluso, a partir de múltiples agregados definidos en el modelo de dominio para el área transaccional. En este caso, debido a que está creando consultas independientes del modelo de dominio, los límites y restricciones de los agregados se ignoran por completo y puede consultar cualquier tabla y columna que pueda necesitar. Este enfoque proporciona una gran flexibilidad y productividad para los desarrolladores que crean o actualizan las consultas.

Los *ViewModels* pueden ser tipos estáticos definidos en clases, o se pueden crear dinámicamente en función de las consultas realizadas (como se implementa en el microservicio de pedidos), lo que resulta muy ágil para los desarrolladores.

Usando Dapper como micro ORM para realizar las consultas

Puede usar cualquier micro ORM, Entity Framework Core o, incluso, simplemente ADO.NET para realizar consultas. En la aplicación de referencia, seleccionamos Dapper para el microservicio de pedidos en eShopOnContainers, como un buen ejemplo de un micro ORM popular. Puede ejecutar consultas SQL simples con gran rendimiento, porque es un *framework* muy ligero. Con Dapper, puede escribir una consulta SQL que pueda acceder y hacer *joins* entre varias tablas.

Dapper es un proyecto *open source* (original creado por Sam Saffron) y es parte de los componentes utilizados en Stack Overflow. Para usar Dapper, sólo necesita instalarlo a través del paquete Dapper NuGet, como se muestra en la siguiente figura.

También necesitará agregar una declaración *using* para que su código tenga acceso a los *extension methods* de Dapper.

Cuando use Dapper en su código, se usa directamente la clase **`SqlClient`** disponible en el *namespace* **`System.Data.SqlClient`**. A través del método **`QueryAsync`** y otros *extension methods* que extienden la clase **`SqlClient`**, puede simplemente ejecutar consultas de una manera directa y rápida.

ViewModels dinámicos versus estáticos

Al devolver *ViewModels* desde el lado del servidor a las aplicaciones cliente, puede pensar en esos *ViewModels* como DTO (Data Transfer Objects) que pueden ser diferentes a las entidades del dominio, porque los *ViewModels* manejan los datos como los necesita la aplicación cliente. Por lo tanto, en muchos casos, puede agregar datos provenientes de múltiples entidades de dominio y componer los *ViewModels* precisamente de la forma en que la aplicación cliente los necesita.

Esos *ViewModels* o DTO se pueden definir explícitamente (como clases portadoras de datos), como la clase **`OrderSummary`** mostrada en un fragmento de código más adelante, o simplemente se pueden devolver *ViewModles* dinámicos, en función de los atributos obtenidos con las consultas.

El *ViewModel* como un tipo dinámico

Como se muestra en el siguiente código, las consultas pueden devolver directamente un *ViewModel* basado en los atributos devueltos por el *query*, simplemente usando el tipo **`dynamic`**, como parámetro genérico de **`QueryAsync<>()`**. Eso significa que la lista de atributos devueltos se basa en la consulta misma. Si agrega una nueva columna a la consulta o *join*, esa información se agrega dinámicamente al *ViewModel* resultante.

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;
public class OrderQueries : IOrderQueries
{
  public async Task<IEnumerable<dynamic>> GetOrdersAsync()
  {
      using (var connection = new SqlConnection(_connectionString))
      {
        connection.Open();
        return await connection.QueryAsync<dynamic>(@"SELECT o.[Id] as ordernumber,
                  o.[OrderDate] as [date],os.[Name] as [status],
                  SUM(oi.units*oi.unitprice) as total
                  FROM [ordering].[Orders] o
                  LEFT JOIN[ordering].[orderitems] oi ON  o.Id = oi.orderid
                  LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId = os.Id
                  GROUP BY o.[Id], o.[OrderDate], os.[Name]");
      }
  }
}
```

El punto importante es que, al usar un tipo dinámico, la colección de datos devuelta se ensamblará dinámicamente como el *ViewModel*.

Ventajas: Este enfoque reduce la necesidad de modificar *ViewModels* estáticos cada vez que actualiza la sentencia de SQL de una consulta, lo que hace que este enfoque de diseño sea bastante ágil al codificar, directo y rápido para evolucionar con respecto a cambios futuros.

Inconvenientes: A largo plazo, los tipos dinámicos pueden afectar negativamente la claridad e, incluso, afectar la compatibilidad de un servicio con las aplicaciones cliente. Además, el software de middleware como Swagger, no puede ofrecer el mismo nivel de documentación en los tipos devueltos si se usan tipos dinámicos.

Los *ViewModels* como clases DTO predefinidas

Ventajas: Tener clases estáticas predefinidas como *ViewModel*, como los "contratos" basados en clases explícitas de DTO, es definitivamente mejor para las API públicas, pero también para los microservicios a largo plazo, incluso si sólo son utilizados por la misma aplicación.

Si desea especificar tipos de respuesta para Swagger, debe usar clases DTO explícitas como tipo devuelto. Por lo tanto, las clases DTO predefinidas le permiten ofrecer una información más detallada con Swagger. Eso beneficiará la documentación de la API y la compatibilidad cuando la consuma.

Inconvenientes: Como se mencionó, cuando se actualiza el código, se requieren algunos pasos más para actualizar las clases de DTO.

Sugerencia basada en nuestra experiencia: en las consultas implementadas en el microservicio de pedido en eShopOnContainers, comenzamos a desarrollar mediante el uso de *ViewModels* dinámicos, ya que era muy sencillo y ágil cuando se desarrollaba. Pero, una vez que el desarrollo se estabilizó, optamos por refactorizar esto y usar DTO estáticos para los *ViewModels*, debido a las ventajas mencionadas.

En el siguiente código, puede ver cómo, en este caso, la consulta está devolviendo datos mediante el uso de una clase DTO de *ViewModel* explícita: la clase **`OrderSummary`**.

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;

public class OrderQueries : IOrderQueries
{
  public async Task<IEnumerable<OrderSummary>> GetOrdersAsync()
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            var result = await connection.QueryAsync< OrderSummary>(
                  @"SELECT o.[Id] as ordernumber,
                  o.[OrderDate] as [date],os.[Name] as [status],
                  SUM(oi.units*oi.unitprice) as total
                  FROM [ordering].[Orders] o
                  LEFT JOIN[ordering].[orderitems] oi ON  o.Id = oi.orderid
                  LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId = os.Id
                  GROUP BY o.[Id], o.[OrderDate], os.[Name]
                  ORDER BY o.[Id]");
        }
```

```
    }
}
```

Describiendo los tipos devueltos por las APIs Web

Los desarrolladores que consumen APIs Web y microservicios, están más preocupados por lo que se devuelve, específicamente los tipos de respuesta y los códigos de error (si no son estándar). Estos se manejan en los comentarios de XML y en las anotaciones de datos.

Sin la documentación adecuada en la interfaz de usuario de Swagger, el consumidor no tiene conocimiento de qué tipo se devuelve o cuáles pueden ser los códigos Http resultantes. Ese problema se soluciona agregando el atributo **ProducesResponseType** (definido en **Microsoft.AspNetCore.Mvc**), para que Swagger genere una información más completa sobre el modelo y los valores devueltos por la API, como se muestra en el siguiente código.

```
namespace Microsoft.eShopOnContainers.Services.Ordering.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class OrdersController : Controller
    {
       //Additional code...
       [Route("")]
       [HttpGet]
       [ProducesResponseType(typeof(IEnumerable<OrderSummary>),
                             (int)HttpStatusCode.OK)]
       public async Task<IActionResult> GetOrders()
       {
           var orderTask = _orderQueries.GetOrdersAsync();
           var orders = await orderTask;
           return Ok(orders);
        }
    }
}
```

Sin embargo, el atributo **ProducesResponseType** no puede usar *dynamic* como tipo, sino que requiere el uso de tipos explícitos, como **OrderSummary**, el *ViewModel* DTO que se muestra en el siguiente fragmento de código.

```
public class OrderSummary
{
    public int ordernumber { get; set; }
    public DateTime date { get; set; }
    public string status { get; set; }
    public double total { get; set; }
}
```

Esta es otra razón por la cual, a largo plazo, es mejor devolver tipos explícitos que dinámicos.

Al utilizar el atributo **ProducesResponseType**, también puede especificar cuál es el resultado esperado con respecto a posibles errores/códigos HTTP, como 200,400, etc.

En la siguiente imagen, puede ver cómo la interfaz de usuario de Swagger muestra la información del ResponseType.

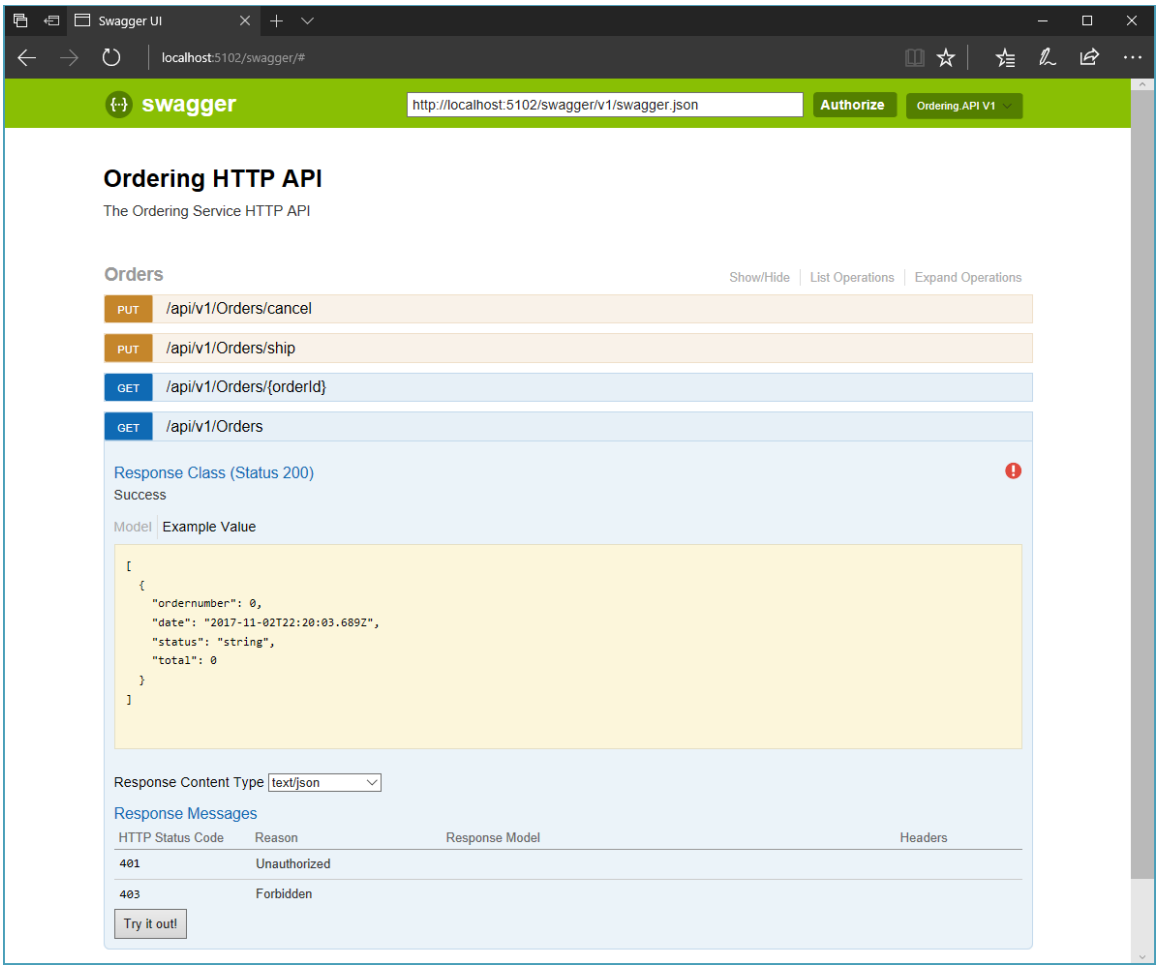


***Figura 7-5**. Swagger UI mostrando los tipos de respuesta y los posibles estatus Http de una API Web*

Puede ver en la imagen de arriba algunos valores de ejemplo basados en los tipos de *ViewModel*, más los posibles códigos Http que se pueden devolver.

Recursos adicionales

- **Dapper**
  https://github.com/StackExchange/dapper-dot-net
- **Julie Lerman. Data Points - Dapper, Entity Framework y aplicaciones híbridas (MSDN Mag. article)**
  https://msdn.microsoft.com/magazine/mt703432.aspx
- **Páginas de ayuda de ASP.NET Core Web API mediante Swagger**
  https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio

Diseñando un microservicio orientado a DDD

El diseño impulsado por dominio (DDD) aboga por el modelado basado en la realidad del negocio, como lo relevante para sus casos de uso. En el contexto de la creación de aplicaciones, DDD se refiere a los problemas como "dominios". Describe las áreas problemáticas independientes como *Bounded Contexts* (cada *Bounded Context* se correlaciona con un microservicio) y enfatiza el uso de un lenguaje común (lenguaje ubicuo) para hablar de estos problemas o dominios. También sugiere muchos conceptos y patrones técnicos, como entidades de dominio con modelos enriquecidos (sin usar modelos de dominio anémicos), objetos de valor (*value objects*), agregados y raíz de agregación (o entidad raíz) para soportar la implementación interna. Esta sección presenta el diseño y la implementación de esos patrones internos.

Algunas veces, estas reglas y patrones técnicos de DDD se perciben como obstáculos, que tienen una curva de aprendizaje pronunciada para implementar enfoques de DDD. Pero la parte importante no son los patrones en sí mismos, sino la organización del código para que esté alineado con los problemas del negocio y usando los mismos términos del negocio (lenguaje ubicuo). Además, los enfoques DDD deberían aplicarse sólo si está implementando microservicios complejos con reglas del negocio significativas. Las responsabilidades más simples, como un servicio CRUD, se pueden gestionar con enfoques más sencillos.

La tarea clave para diseñar y definir un microservicio, es dónde dibujar los límites. Los patrones DDD le ayudan a comprender la complejidad del dominio. Para el modelo de dominio de cada *Bounded Context* (BC), debe identificar y definir las entidades, los *value objects* y los agregados que modelan su dominio. Usted debe crear y refinar un modelo de dominio que esté dentro del límite que define su contexto. Y eso es muy explícito en la forma de un microservicio. Los componentes dentro de esos límites terminan siendo sus microservicios, aunque en algunos casos un BC o microservicio del negocio puede estar compuesto por varios servicios físicos. DDD es sobre límites, así como también lo son los microservicios.

Mantener los límites del contexto del microservicio relativamente pequeños

Determinar dónde colocar los límites entre contextos delimitados, es lograr un balance entre dos objetivos contrapuestos. En primer lugar, se desea crear los microservicios más pequeños posibles, aunque ese no debería ser el principal motivo, sino crear un límite alrededor de las cosas que necesitan cohesión. En segundo lugar, se desea evitar las comunicaciones excesivas (*chatty*, "chismorreras") entre microservicios. Estos objetivos se pueden contradecir entre sí. Debe equilibrarlos descomponiendo el sistema en tantos microservicios pequeños como sea posible, hasta que vea que los límites de comunicación crecen rápidamente con cada intento adicional de separar un nuevo *Bounded Context*. La cohesión es clave dentro de cada *Bounded Context*.

Es similar al Inappropriate Intimacy code smell (problema por intimidad inapropiada) al implementar clases. Si dos microservicios necesitan colaborar mucho entre ellos, probablemente deberían ser el mismo microservicio.

Otra forma de ver esto es la autonomía. Si un microservicio debe confiar en otro servicio para atender directamente una solicitud, no es verdaderamente autónomo.

Las capas en los microservicios DDD

La mayoría de las aplicaciones empresariales, con una complejidad técnica y del negocio significativa, están definidas por varias capas. Las capas son un artefacto lógico y no están relacionadas con el despliegue del servicio. Son un recurso para ayudar a los desarrolladores a administrar la complejidad del código. Las diferentes capas (como la capa de modelo de dominio frente a la capa de presentación, etc.) pueden manejar diferentes tipos (clases), lo que obliga a realizar traducciones entre esos tipos.

Por ejemplo, una entidad se podría cargar desde la base de datos. Entonces parte de esa información, o una agregación de información que incluya datos adicionales de otras entidades, se puede enviar a la interfaz de usuario del cliente a través de una API REST Web. El punto aquí es que la entidad de dominio está contenida dentro de la capa de modelo de dominio y no se debe propagar a otras áreas a las que no pertenece, como la capa de presentación.

Además, debe tener entidades siempre válidas (consulte la sección Diseñar validaciones en la capa de modelo de dominio, más adelante) controladas por raíces de agregación (entidades raíz). Por lo tanto, las entidades no se deben usar directamente en las vistas del cliente, porque en el nivel de interfaz de usuario hay datos que no se pueden validar. Para esto es el *ViewModel*. El *ViewModel* es un modelo de datos exclusivo para las necesidades de la capa de presentación. Las entidades de dominio no pertenecen directamente a un *ViewModel*. En su lugar, debe traducir entre *ViewModels* y entidades de dominio y viceversa.

Al abordar la complejidad, es importante tener un modelo de dominio controlado por raíces de agregación (profundizaremos en esto más adelante) que asegure que todas las invariantes y reglas relacionadas con ese grupo de entidades (agregado) se realicen a través de un solo punto de entrada, la raíz (o entidad) de agregación.

La Figura 7-5 muestra cómo se implementa un diseño en capas en la aplicación eShopOnContainers.

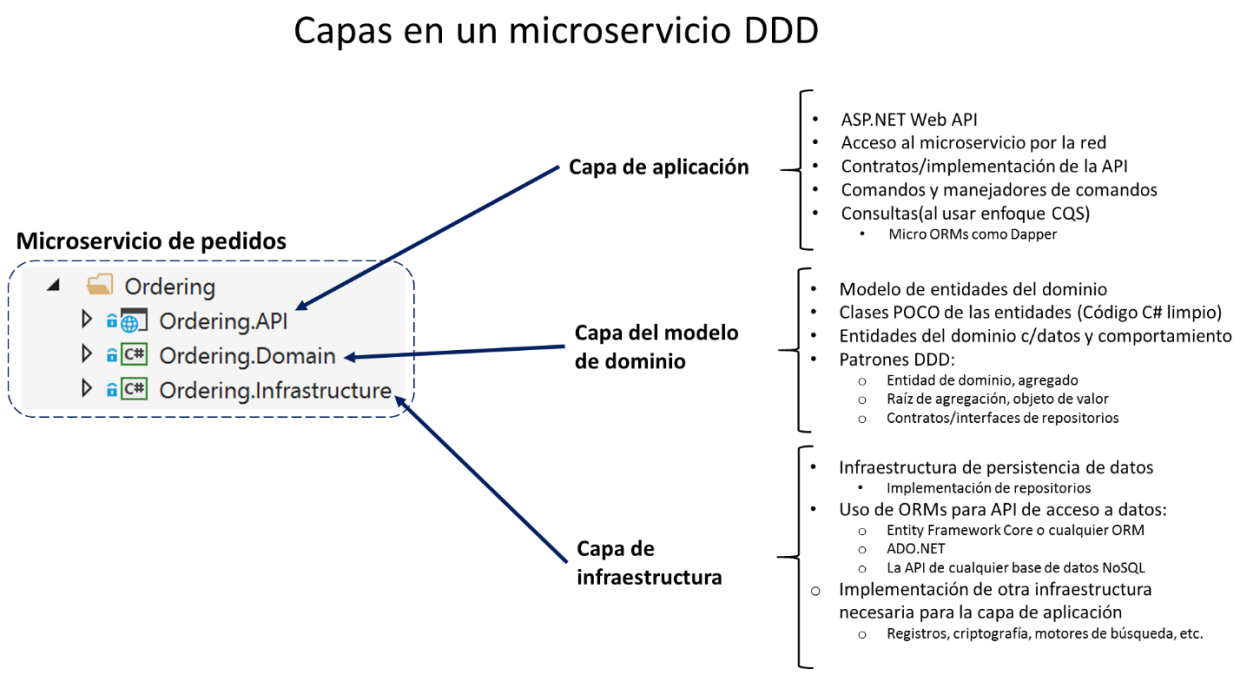


***Figura 7-5**. Capas DDD del microservicio de pedidos en eShopOnContainers*

Debe diseñar el sistema para que cada capa se comunique sólo con ciertas otras capas. Eso puede ser más fácil de aplicar si las capas se implementan como librerías diferentes, porque puede identificar claramente qué dependencias se establecen entre ellas. Por ejemplo, la capa del modelo de dominio no debe tomar una dependencia de ninguna otra capa (las clases de modelo de dominio deben ser Plain Old CLR Objects o POCO). Como se muestra en la Figura 7-6, la librería de la capa **Ordering.Domain** tiene dependencias solo de las librerías .NET Core o paquetes NuGet, pero no en ninguna otra librería personalizada (como datos, persistencia, etc.).

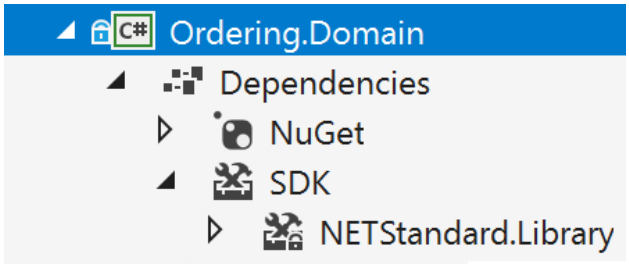


***Figura 7-6**. Implementar las capas como librerías, simplifica el control de las dependencias entre ellas*

La capa del modelo de dominio

El excelente libro de Eric Evans, Domain-Driven Design, dice lo siguiente sobre la capa del modelo de dominio y la capa de aplicación.

> ***Capa del modelo de dominio**: Es la responsable de representar conceptos del negocio, información sobre la situación y las reglas del negocio. El estado que refleja la situación del negocio se controla y*

se usa aquí, aunque los detalles técnicos de su almacenamiento se delegan en la infraestructura. Esta capa es el corazón del software del negocio.

La capa del modelo de dominio es donde se expresa el negocio. Cuando implementa una capa del modelo de dominio como un microservicio en .NET, esa capa se prepara como una librería de clases, con las entidades de dominio que capturan los datos más el comportamiento (métodos con lógica).

Siguiendo los principios de Ignorancia de la Persistencia e Ignorancia de la Infraestructura, esta capa debe ignorar por completo los detalles de persistencia de datos. Estas tareas de persistencia deben ser realizadas por la capa de infraestructura. Por lo tanto, esta capa no debe tomar dependencias directas de la infraestructura, lo que significa que una regla importante es que las clases de entidad del modelo de dominio deben ser POCOs.

Las entidades de dominio no deben tener ninguna dependencia directa (como derivar de una clase base) de ningún *framework* de infraestructura de acceso a datos como Entity Framework o NHibernate. Idealmente, las entidades de su dominio no deberían derivar o implementar ningún tipo definido en ningún *framework* de infraestructura.

La mayoría de los *frameworks* ORM modernos como Entity Framework Core permiten este enfoque, de modo que las clases de su modelo de dominio no estén acopladas a la infraestructura. Sin embargo, tener entidades POCO no siempre es posible cuando se utilizan ciertas bases de datos y *frameworks* NoSQL, como Actores y Reliable Collections en Azure Service Fabric.

Incluso cuando es importante seguir el principio de Ignorancia de la Persistencia para su modelo de Dominio, no debe ignorar los problemas de persistencia. Es muy importante comprender el modelo de datos físicos y cómo se correlaciona con el modelo de objetos de su entidad. De lo contrario, puede crear diseños imposibles de implementar.

Además, esto no significa que pueda tomar un modelo diseñado para una base de datos relacional y moverlo directamente a una base de datos NoSQL u orientada a documentos. En algunos modelos de entidades, el modelo podría ajustarse, pero por lo general no es así. Todavía existen restricciones que su modelo de entidades debe cumplir, basadas tanto en la tecnología de almacenamiento como en la tecnología ORM.

La capa de la aplicación

Pasando a la capa de aplicación, podemos citar nuevamente el libro Domain-Driven Design, de Eric Evans:

> **Capa de la aplicación**: Define los trabajos que el software debe hacer y dirige los objetos del dominio para resolver problemas. Las tareas de las que esta capa es responsable son significativas para la empresa o necesarias para la interacción con las capas de aplicación de otros sistemas. Esta capa se mantiene delgada. No contiene reglas ni conocimiento del negocio, sino que sólo coordina tareas y delega el trabajo a colaboraciones de objetos de dominio en la siguiente capa, hacia abajo. No tiene un estado que refleje la situación del negocio, pero puede tener un estado que refleje el progreso de una tarea para el usuario o el programa.

La capa de aplicación de un microservicio en .NET suele codificarse como un proyecto ASP.NET Core Web API. El proyecto implementa la interacción del microservicio, el acceso remoto a la red y las API Web externas utilizadas desde la interfaz de usuario o las aplicaciones cliente. Incluye consultas, comandos aceptados por el microservicio, si se usa un enfoque CQRS e, incluso, la comunicación

basada en eventos entre microservicios (eventos de integración). La API Web ASP.NET Core que representa la capa de aplicación no debe contener reglas del negocio o conocimiento de dominio (especialmente reglas de dominio para transacciones o actualizaciones), estos deben ser propiedad de la librería del modelo de dominio. La capa de aplicación sólo debe coordinar tareas y no debe contener ni definir ningún estado del dominio (modelo de dominio). Delega la ejecución de las reglas de negocio en las propias clases del modelo de dominio (agregados raíz y entidades del dominio), lo que finalmente actualizará los datos dentro de esas entidades.

Básicamente, la lógica de la aplicación es donde se implementa todos los casos de uso que dependen de un *front-end* determinado. Por ejemplo, la implementación relacionada con un servicio de API Web.

El objetivo es que la lógica de dominio en la capa del modelo de dominio, sus invariantes, el modelo de datos y las reglas de negocio relacionadas sean completamente independientes de las capas de presentación y aplicación. Más que nada, la capa de modelo de dominio no debe depender directamente de ningún *framework* de infraestructura.

La capa de infraestructura

La capa de infraestructura es la forma en que los datos que se mantienen inicialmente en las entidades de dominio (en la memoria) se conservan en bases de datos u otro almacén persistente. Un ejemplo es el usar Entity Framework Core para implementar las clases del patrón *Repository* que usan un **`DBContext`** para persistir datos en una base de datos relacional.

De acuerdo con los principios de Ignorancia de la Persistencia e Ignorancia de la Infraestructura mencionados anteriormente, la capa de infraestructura no debe "contaminar" la capa de modelo de dominio. Debe mantener las entidades de modelo de dominio independientes de la infraestructura que utiliza para conservar los datos (EF o cualquier otro *framework*) al no tener dependencias fuertes sobre los *frameworks*. La librería de la capa del modelo de dominio sólo debe tener código del dominio, sólo las entidades POCO que implementan el núcleo de su software y están completamente desacopladas de las tecnologías de infraestructura.

Por lo tanto, sus capas o librería y proyectos deberían depender, en última instancia, de la capa (librería) de su modelo de dominio y no viceversa, como se muestra en la Figura 7-7.

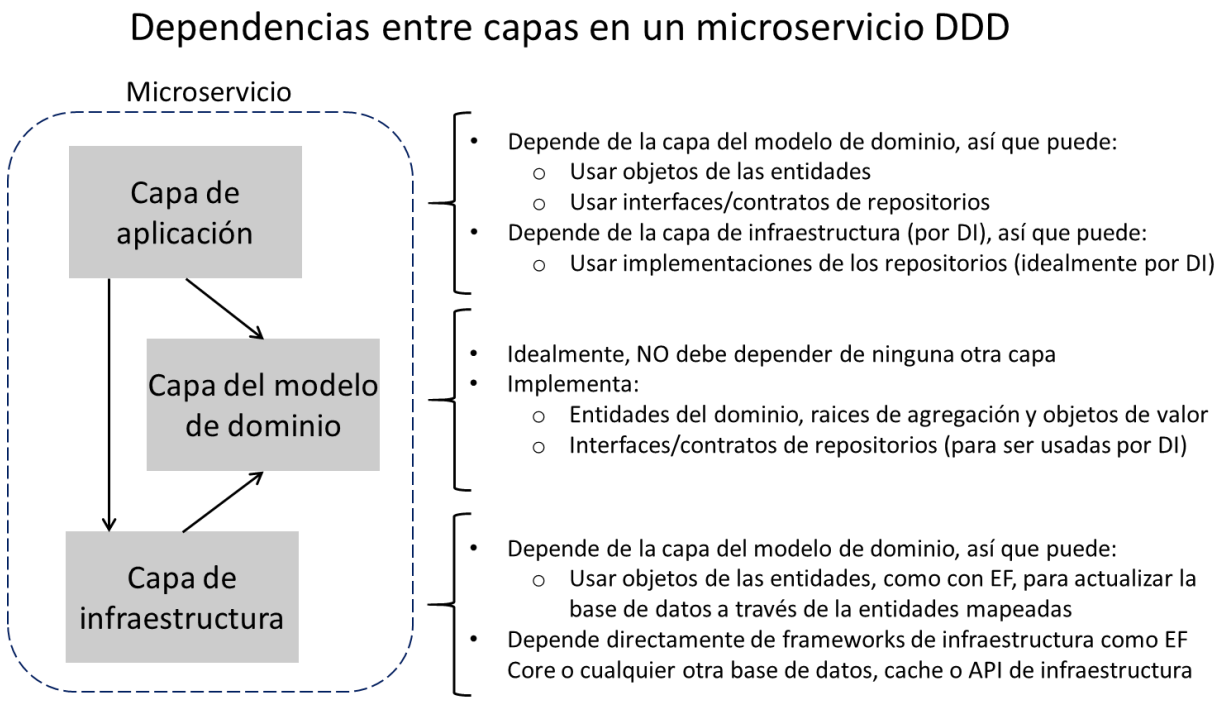


***Figura 7-7**. Dependencies between layers in DDD*

Este diseño de capa debe ser independiente para cada microservicio. Como se señaló anteriormente, puede implementar los microservicios más complejos siguiendo patrones DDD, mientras que puede implementas microservicios más simples basados en datos (microservicios CRUD simples en una sola capa) de una manera más sencilla.

Recursos adicionales

- **DevIQ. Persistence Ignorance principle**
  http://deviq.com/persistence-ignorance/
- **Oren Eini. Infrastructure Ignorance**
  https://ayende.com/blog/3137/infrastructure-ignorance
- **Angel Lopez. Layered Architecture In Domain-Driven Design**
  https://ajlopez.wordpress.com/2008/09/12/layered-architecture-in-domain-driven-design/

Diseñando un modelo de dominio como un microservicio

Definir un modelo de dominio expresivo para cada microservicio o Bounded Context

Su objetivo es crear un modelo de dominio único y cohesivo para cada microservicio del negocio o *Bounded Context* (BC). Tenga en cuenta, sin embargo, que un BC o microservicio del negocio a veces podría estar compuesto por varios servicios físicos que comparten un modelo de dominio único. El modelo de dominio debe capturar las reglas, el comportamiento, el lenguaje del negocio y las restricciones del *Bounded Context* único o microservicio del negocio que representa.

El patrón Entidad del Dominio

Las entidades representan objetos del dominio y se definen principalmente por su identidad, continuidad y persistencia en el tiempo y nó solo por los atributos que las componen. Como dice Eric Evans, "un objeto definido principalmente por su identidad se llama Entidad". Las entidades son muy importantes en el modelo de dominio, ya que son la base del modelo. Por lo tanto, debe identificarlas y diseñarlas cuidadosamente.

La identidad de una entidad puede pasar por múltiples microservicios o Bounded Contexts.

La misma identidad (es decir, el valor del Id que identifica la entidad, aunque no la misma entidad) se puede modelar a través de múltiples *Bounded Contexts* o microservicios. Sin embargo, eso no implica que la misma entidad, con los mismos atributos y lógica, se implemente en múltiples *Bounded Contexts*. En cambio, las entidades en cada *Bounded Context* limitan sus atributos y comportamientos a los requeridos en el dominio del *Bounded Context*.

Por ejemplo, la entidad “Comprador” puede tener la mayoría de los atributos de una persona que se definen como entidad “Usuario” en el microservicio de identidad o perfil, incluida la identidad. Pero la entidad “Comprador” en el microservicio de pedidos puede tener menos atributos, porque sólo ciertos datos del comprador están relacionados con el proceso de pedido. El contexto de cada microservicio o *Bounded Context* afecta su modelo de dominio.

Las entidades de dominio deben implementar comportamiento además de implementar atributos de datos

Una entidad de dominio en DDD debe implementar la lógica de dominio o el comportamiento relacionado con los datos de la entidad (el objeto al que se accede en la memoria). Por ejemplo, como parte de una clase de la entidad “Pedido”, debe tener la lógica de negocios y las operaciones implementadas como métodos para tareas tales como agregar un artículo de pedido, validar los datos y calcular el total. Los métodos de la entidad se ocupan de las invariantes y las reglas de la entidad, en lugar de tener esas reglas repartidas en la capa de la aplicación.

La figura 7-8 muestra una entidad del dominio que implementa no sólo atributos de datos, sino también operaciones o métodos con lógica relacionada del dominio.

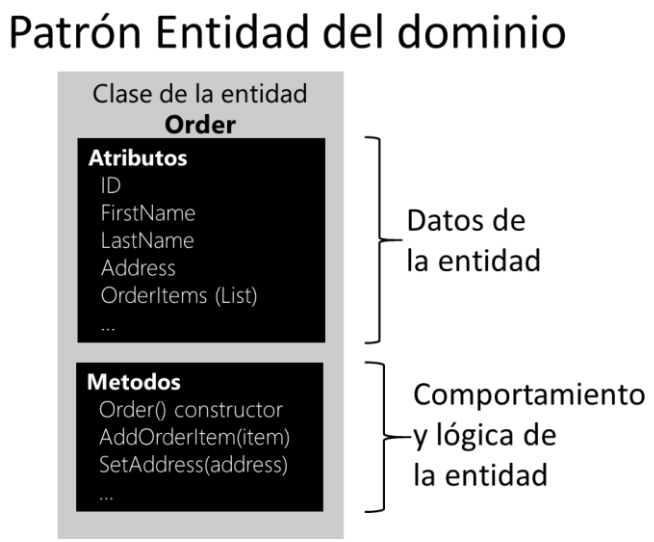


***Figura 7-8**. Ejemplo del diseño de una entidad del dominio, implementando datos y comportamiento*

Por supuesto, a veces puede tener entidades que no implementan ninguna lógica como parte de la clase de entidad. Esto puede ocurrir con entidades secundarias dentro de un agregado, si la entidad hija no tiene ninguna lógica especial porque la mayoría de la lógica se define en la raíz del agregado. Si tiene un microservicio complejo que tiene mucha lógica implementada en las clases de servicio en lugar de en las entidades de dominio, podría caer en el modelo de dominio anémico, que se explica en la siguiente sección.

Modelo de dominio expresivo (rico) versus modelo de dominio anémico

En su publicación AnemicDomainModel, Martin Fowler describe un modelo de dominio anémico de esta forma:

> *El síntoma básico de un Modelo de Dominio Anémico es que a primera vista parece real. Hay objetos, muchos de ellos nombrados usando los nombres en el espacio del dominio y estos objetos están conectados con las relaciones expresivas y la estructura que tienen los modelos del dominio verdaderos. El problema viene cuando se observa el comportamiento y es evidente que casi no hay ningún comportamiento en estos objetos, convirtiéndolos en poco más que bolsas de getters y setters.*

Por supuesto, cuando utiliza un modelo de dominio anémico, esos modelos de datos se usarán a partir de un conjunto de objetos de servicio (tradicionalmente denominado la capa del negocio) que captura todo el dominio o lógica del negocio. La capa del negocio se ubica por encima del modelo de dominio y lo usa sólo como datos.

El modelo de dominio anémico es sólo un diseño de estilo procedimental. Los objetos de entidades anémicas no son objetos reales, porque carecen de comportamiento (métodos). Sólo tienen propiedades de datos y, por lo tanto, no es un diseño orientado a objetos. Al poner todo el comportamiento en objetos de servicio (la capa de negocios) esencialmente se termina con código espagueti o scripts de transacción y, por lo tanto, se pierden las ventajas que ofrece un modelo de dominio.

De todos modos, si su microservicio o *Bounded Context* es muy simple (un servicio CRUD), el modelo de dominio anémico, en forma de objetos que sólo tienen propiedades de datos, podría ser lo suficientemente bueno y podría no ser útil implementar patrones DDD más complejos. En ese caso, será simplemente un modelo de persistencia, porque usted ha creado intencionalmente una entidad con sólo datos, para los propósitos de un CRUD.

Es por eso que las arquitecturas de microservicios son perfectas para un enfoque multi-arquitectónico, que depende de cada *Bounded Context*. Por ejemplo, en eShopOnContainers, el microservicio de pedidos implementa patrones DDD, pero el microservicio de catálogo, que es un servicio CRUD simple, no lo hace.

Algunas personas dicen que el modelo de dominio anémico es un anti patrón. Realmente depende de lo que se esté implementando. Si el microservicio que está creando es lo suficientemente simple (por ejemplo, un servicio CRUD), seguir el modelo de dominio anémico no es un anti patrón. Sin embargo, si necesita abordar la complejidad del dominio de un microservicio que tiene muchas reglas del negocio en cambio constante, el modelo de dominio anémico sí podría ser un anti patrón para ese microservicio. En ese caso, diseñarlo como un modelo rico y expresivo, con entidades que contengan datos y comportamientos, así como implementar patrones DDD adicionales (agregados, *value objects*, etc.) podría tener enormes beneficios para el éxito a largo plazo de ese microservicio.

Recursos adicionales

- **DevIQ. Domain Entity**
  http://deviq.com/entity/
- **Martin Fowler. The Domain Model**
  https://martinfowler.com/eaaCatalog/domainModel.html
- **Martin Fowler. The Anemic Domain Model**
  https://martinfowler.com/bliki/AnemicDomainModel.html

El patrón Objeto de Valor (Value Object)

Como ha señalado Eric Evans, "muchos objetos no tienen identidad conceptual. Estos objetos describen ciertas características de una cosa".

Una entidad requiere una identidad, pero hay muchos objetos en un sistema que no la tienen, como el patrón *Value Object*. Un *Value Object* es un objeto sin identidad conceptual, que describe un aspecto de dominio. Estos son objetos que crea una instancia para representar elementos de diseño que sólo le interesan temporalmente. En ese caso, importa *lo que son*, no *quiénes son*. Los ejemplos incluyen números y *strings*, pero también pueden ser conceptos de nivel superior como grupos de atributos.

Algo que es una entidad en un microservicio podría no serlo en otro microservicio, porque en el segundo caso, el *Bounded Context* podría tener un significado diferente. Por ejemplo, una dirección en una aplicación de comercio electrónico podría no tener una identidad, ya que podría sólo representar un grupo de atributos del perfil del cliente para una persona o empresa. En este caso, la dirección debe clasificarse como un objeto de valor. Sin embargo, en una solicitud para una empresa de servicios de energía eléctrica, la dirección del cliente podría ser importante para el dominio del negocio. Por lo tanto, la dirección debe tener una identidad para que el sistema de facturación pueda vincularse directamente con la dirección. En ese caso, una dirección debe clasificarse como una entidad de dominio.

Una persona con un nombre y un apellido suele ser una entidad porque una persona tiene identidad, incluso si el nombre y el apellido coinciden con otro conjunto de valores, porque esos nombres se refieren a una persona diferente.

Los *value objects* son difíciles de administrar en bases de datos relacionales y ORM como EF, mientras que en las bases de datos orientadas a documentos son más fáciles de implementar y usar.

EF Core 2.0, incluye la característica Owned Entities, que facilita el manejo de *Value Objects*, como veremos en detalle más adelante.

Recursos adicionales

- **Martin Fowler. Value Object pattern**
  https://martinfowler.com/bliki/ValueObject.html
- **Value Object**
  http://deviq.com/value-object/
- **Value Objects in Test-Driven Development**
  https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects
- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software.**
  (Book; includes a discussion of value objects)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/

El patrón Agregado

Un modelo de dominio contiene grupos de diferentes entidades y procesos que pueden controlar un área importante de funcionalidad, como la entrega de pedidos o el inventario. Una unidad DDD de grano más fino es el agregado, que describe un grupo o grupo de entidades y comportamientos que pueden tratarse como una unidad cohesiva.

Por lo general, se define un agregado según las transacciones que necesita. Un ejemplo clásico es un pedido que también contiene una lista de artículos del pedido. Un artículo de pedido generalmente será una entidad. Pero será una entidad hija dentro del agregado de pedido, que también contendrá la entidad de pedido como su entidad raíz, típicamente denominada raíz de agregación (aggregate root).

La identificación de agregados puede ser difícil. Un agregado es un grupo de objetos que deben ser consistentes entre sí, pero no se puede simplemente seleccionar un grupo de objetos y etiquetarlos como un agregado. Debe comenzar con un concepto de dominio y pensar en las entidades que se utilizan en las transacciones más comunes relacionadas con ese concepto. Aquellas entidades que necesitan ser consistentes transaccionalmente son lo que forma un agregado. Pensar en las transacciones es probablemente la mejor manera de identificar agregados.

El patrón Raíz de Agregación o Entidad Raíz

Un agregado se compone de al menos una entidad: la raíz del agregado, también llamada entidad raíz o entidad primaria. Además, puede tener múltiples entidades secundarias y *value objects*, con todas las entidades y objetos trabajando juntos para implementar el comportamiento y las transacciones requeridas.

El propósito de una entidad raíz es garantizar la consistencia del agregado; debe ser el único punto de entrada para las actualizaciones del agregado, a través de métodos u operaciones en la clase de entidad raíz. Sólo se deben realizar cambios de las entidades dentro del agregado, a través de la

entidad raíz. Es el guardián de la consistencia del agregado, teniendo en cuenta todas las invariantes y las reglas de consistencia que necesitan cumplir en su conjunto. Si cambia una entidad secundaria u objeto de valor de forma independiente, la entidad raíz no puede garantizar que el agregado se encuentre en un estado válido. Sería como una mesa con una pierna suelta. Mantener la consistencia es el objetivo principal de la entidad raíz.

En la figura 7-9, puede ver agregados de ejemplo como el agregado del comprador, que contiene una sola entidad (la entidad raíz **Buyer**). El agregado de pedido contiene múltiples entidades y un objeto de valor.

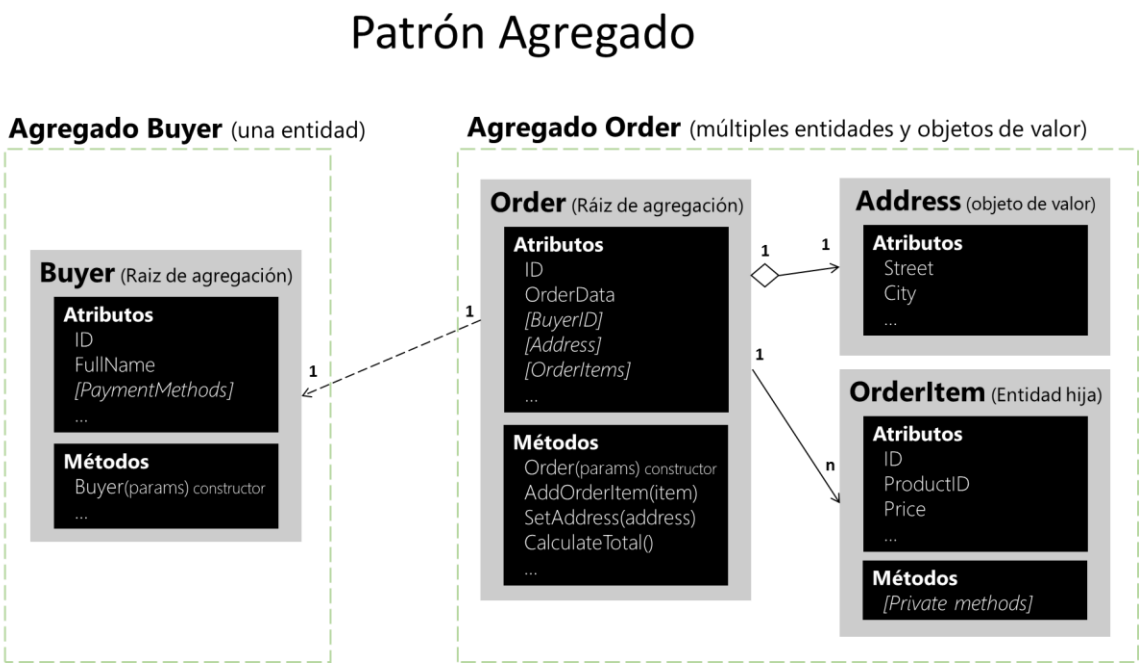


***Figura 7-9**. Ejemplo de agregados con una sola entidad y múltiples entidades*

Tenga en cuenta que el agregado **Buyer** podría tener entidades secundarias adicionales, dependiendo de su dominio, como lo hace en el microservicio de pedidos en la aplicación eShopOnContainers. La Figura 7-9 simplemente ilustra un caso en el que el comprador tiene una sola entidad, como un ejemplo de un agregado que contiene sólo la entidad raíz.

Para mantener la separación de agregados y mantener los límites claros entre ellos, es una buena práctica en un modelo de dominio DDD, rechazar la navegación directa entre agregados y sólo tener el campo de clave foránea (FK), como se implementó en el modelo de dominio del microservicio de pedidos en eShopOnContainers. La entidad **Order** sólo tiene un campo FK para el comprador, pero no una propiedad de navegación EF Core, como se muestra en el siguiente código:

```
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId;  //FK pointing to a different aggregate root
    public OrderStatus OrderStatus { get; private set; }
    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    //… Additional code
}
```

Identificar y trabajar con agregados requiere investigación y experiencia. Para obtener más información, consulte la siguiente lista de recursos adicionales.

Recursos adicionales

- **Vaughn Vernon. Effective Aggregate Design - Part I: Modeling a Single Aggregate**
  https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_1.pdf
- **Vaughn Vernon. Effective Aggregate Design - Part II: Making Aggregates Work Together**
  https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_2.pdf
- **Vaughn Vernon. Effective Aggregate Design - Part III: Gaining Insight Through Discovery**
  https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_3.pdf
- **Sergey Grybniak. DDD Tactical Design Patterns**
  https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part
- **Chris Richardson. Developing Transactional Microservices Using Aggregates**
  https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson
- **DevIQ. The Aggregate pattern**
  http://deviq.com/aggregate-pattern/

Implementando un microservicio del modelo de dominio con .NET Core

En la sección anterior, se explicaron los principios y patrones de diseño fundamentales para desarrollar un modelo de dominio. Ahora es el momento de explorar posibles formas de implementar el modelo de dominio utilizando .NET Core (código simple en C#) y EF Core. Tenga en cuenta que su modelo de dominio estará compuesto simplemente por su código. Sólo tendrá los requisitos del modelo EF Core, pero no dependencias reales de EF. No debe tener dependencias ni referencias fuertes a EF Core ni a ningún otro ORM en su modelo de dominio.

Estructura del modelo de dominio en una librería particular de .NET Standard Library

La organización de carpetas utilizada para la aplicación de referencia eShopOnContainers, demuestra el modelo DDD para la aplicación. Es posible que descubra que una organización de carpetas diferente comunica más claramente las elecciones de diseño realizadas para su aplicación. Como puede ver en la Figura 7-10, en el modelo de dominio de pedidos hay dos agregados, el agregado del pedido y el agregado de comprador. Cada agregado es un grupo de entidades de dominio y *value objects*, aunque también podría tener un agregado compuesto por una entidad de dominio única (la raíz de agregación o la entidad raíz).

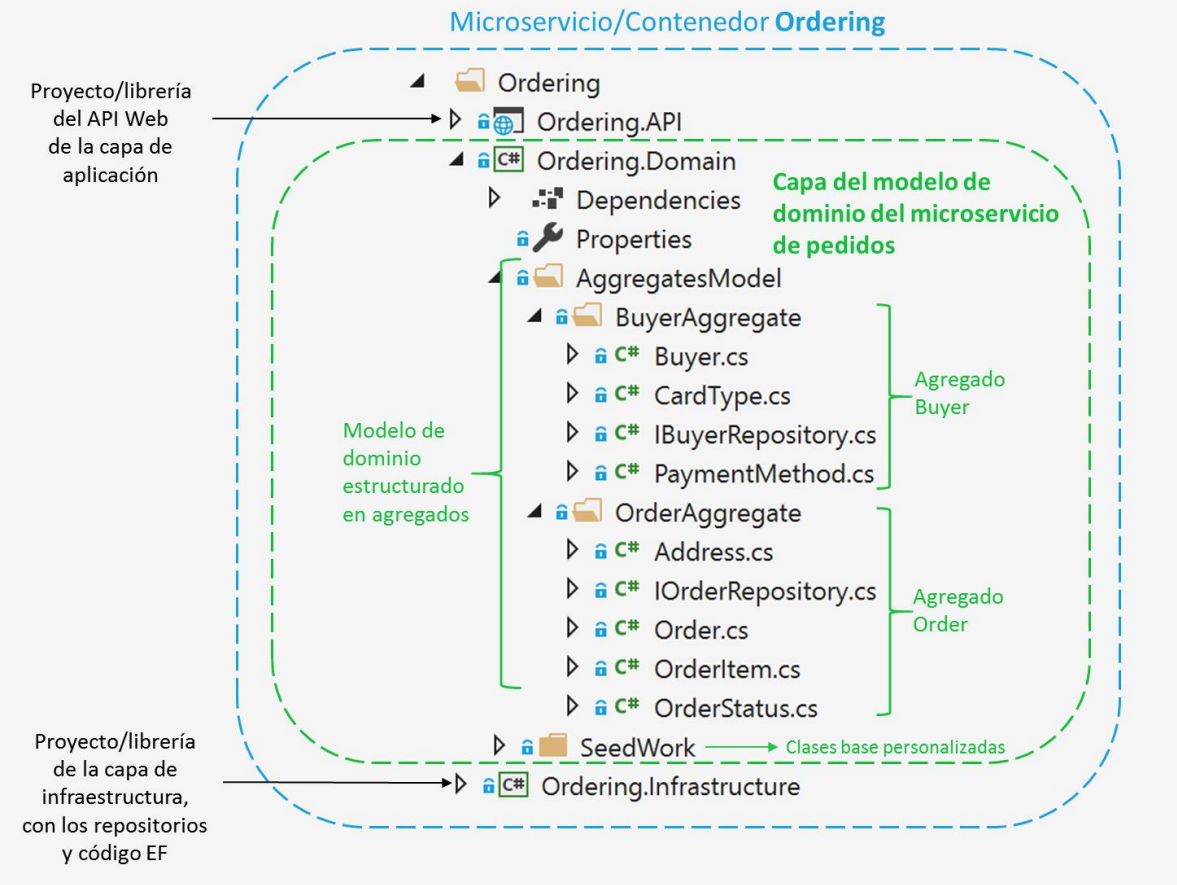


***Figura 7-10**. Estructura del modelo de dominio en el microservicio de pedidos de eShopOnContainers*

Además, la capa del modelo de dominio incluye los contratos de los repositorios (interfaces) que son los requisitos de infraestructura de su modelo de dominio. En otras palabras, estas interfaces expresan qué repositorios y qué métodos debe implementar la capa de infraestructura Es fundamental que la implementación de los repositorios se coloque fuera de la capa de modelo de dominio, en la librería de la capa de infraestructura, para que la capa de modelo de dominio no esté "contaminada" por APIs o clases de infraestructura, como Entity Framework.

También puede ver una carpeta SeedWork que contiene clases base personalizadas que puede usar como base para las entidades de su dominio y *value objects*, para no tener código redundante en cada clase del dominio.

Estructurando los agregados en una librería personalizada de .NET Standard

Un agregado hace referencia a varios objetos del dominio agrupados para que sean coherentes la consistencia transaccional. Esos objetos podrían ser instancias de entidades (una de las cuales es la entidad raíz o raíz de agregación) más cualquier objeto de valor adicional.

La consistencia transaccional significa que se garantiza que un agregado sea consistente y esté actualizado al final de una acción del negocio. Por ejemplo, el agregado de pedido del modelo de dominio está compuesto como se muestra en la Figura 7-11.

Figura 7-11. *El agragado* ***Order*** *en el proyecto de Visual Studio*

Si abre alguno de los ficheros en una carpeta del agregado, puede ver cómo está marcada como una clase base o interfaz personalizada, como entidad u objeto de valor, tal como se implementó en la carpeta Seedwork.

Implementando entidades del dominio como clases POCO

Para implementar un modelo de dominio en .NET, se crean clases POCO que corresponden a las entidades de su dominio. En el siguiente ejemplo, la clase **Order** se define como una entidad y también como una raíz de agregación. Como la clase **Order** se deriva de la clase base **Entity**, puede reutilizar el código común relacionado con las entidades. Tenga en cuenta que estas clases base e interfaces las define usted en el proyecto de modelo de dominio, por lo que es su código, no el código de infraestructura de un ORM como EF.

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE 2.0
// Entity is a custom base class with the ID
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId;

    public OrderStatus OrderStatus { get; private set; }
    private int _orderStatusId;

    private string _description;
    private int? _paymentMethodId;

    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    public Order(string userId, Address address, int cardTypeId,
                 string cardNumber, string cardSecurityNumber,
                 string cardHolderName, DateTime cardExpiration,
                 int? buyerId = null, int? paymentMethodId = null)
    {
        _orderItems = new List<OrderItem>();
        _buyerId = buyerId;
        _paymentMethodId = paymentMethodId;
        _orderStatusId = OrderStatus.Submitted.Id;
        _orderDate = DateTime.UtcNow;
        Address = address;

        // ...Additional code ...
    }

    public void AddOrderItem(int productId, string productName,
                             decimal unitPrice, decimal discount,
                             string pictureUrl, int units = 1)
    {
        //...
```

```
        // Domain rules/logic for adding the OrderItem to the order
        // ...

        var orderItem = new OrderItem(productId, productName, unitPrice,
discount, pictureUrl, units);

        _orderItems.Add(orderItem);

    }
    // ...
    // Additional methods with domain rules/logic related to the Order
aggregate
    // ...
}
```

Es importante tener en cuenta que esta es una entidad de dominio implementada como una clase POCO. No tiene ninguna dependencia directa a Entity Framework Core ni en ningún otro *framework* de infraestructura. Esta es una implementación adecuada para DDD, solo código C# implementando un modelo de dominio.

Además, la clase está decorada con una interfaz llamada **`IAggregateRoot`**. Esa es una interfaz vacía, a veces llamada interfaz de marcador (*marker interface*), que se usa solo para indicar que esta clase también es una raíz de agregación.

Una interfaz de marcador a veces se considera como un anti patrón, sin embargo, también es una manera limpia de marcar una clase, especialmente cuando esa interfaz podría estar evolucionando. Un atributo podría ser la alternativa para el marcador, pero es más rápido ver la clase base (**`Entity`**) junto a la interfaz **`IAggregate`** en lugar de poner un atributo **`Aggregate`** encima de la clase. En todo caso, es más un asunto de preferencias.

Tener una raíz de agregación significa, que la mayoría del código relacionado con la consistencia y las reglas de negocio de las entidades del agregado, deben implementarse como métodos en la clase raíz del agregado **`Order`**, es decir, la clase **`Order`** (por ejemplo, **`AddOrderItem()`** al añadir un **`OrderItem`** al agregado). No debe crear ni actualizar objetos **`OrderItem`** de forma directa; la clase **`AggregateRoot`** debe mantener el control y la coherencia de cualquier operación de actualización contra sus entidades "hijas".

Encapsulando data en las Entidades del Dominio

Un problema común en los modelos de entidades, es que exponen las propiedades de navegación de sus colecciones como listas con acceso público. Esto permite que cualquier desarrollador manipule directamente los contenidos de esas colecciones, lo que puede obviar reglas del negocio importantes relacionadas con la colección, posiblemente dejando al objeto en un estado no válido. La solución a esto es exponer el acceso a las colecciones relacionadas como sólo lectura y proporcionar métodos explícitos que definan cómo se pueden manipular.

En el código anterior, observe que muchos atributos son de sólo lectura o privados y sólo actualizables por los métodos de clase, por lo que cualquier actualización tiene en cuenta invariantes y la lógica del dominio de negocio, especificadas dentro de los métodos de clase.

Por ejemplo, siguiendo DDD, no debe hacer lo siguiente desde ningún método manejador de comandos o clase de capa de aplicación:

```
// WRONG ACCORDING TO DDD PATTERNS – CODE AT THE APPLICATION LAYER OR
// COMMAND HANDLERS

// Code in command handler methods or Web API controllers

//... (WRONG) Some code with business logic out of the domain classes ...
OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName,
                                         pictureUrl, unitPrice, discount, units);

//... (WRONG) Accessing the OrderItems colletion directly from the application layer
// or command handlers

myOrder.OrderItems.Add(myNewOrderItem);

//...
```

En este caso, el método **Add** es únicamente una operación para agregar datos, con acceso directo a la colección **OrderItems**. Por lo tanto, la mayor parte de la lógica de dominio, las reglas o las validaciones relacionadas con esa operación con las entidades secundarias, estarán dispersas en la capa de aplicación (manejadores de comandos y controladores de API Web).

Si obviamos la raíz de agregación, esta no puede garantizar sus invariantes, su validez o su consistencia. Eventualmente tendremos código spaghetti o código de script transaccional.

Para seguir patrones DDD, las entidades no deben aceptar asignaciones públicas en ninguna propiedad de la entidad. Los cambios en una entidad deben ser realizados por métodos explícitos, con lenguaje ubicuo explícito, sobre el cambio que están realizando en la entidad.

Además, las colecciones dentro de la entidad (como los ítems del pedido) deben ser propiedades de sólo lectura (el método **AsReadOnly** se explica más adelante). Debería poder actualizarlo sólo desde los métodos de la raíz de agregación o los métodos de la entidad hija.

Como puede ver en el código para la raíz de agregación **Order**, todos los *setters* (lo que permite actualizar las propiedades de la clase) deben ser privados o de sólo lectura externamente, de modo que cualquier operación contra los datos de la entidad o sus entidades secundarias se deba realizar a través de métodos en la clase de entidad. Esto mantiene la coherencia de forma controlada y orientada a objetos en lugar de implementar código de script transaccional.

El siguiente fragmento de código muestra la forma correcta de codificar la tarea de agregar un objeto **OrderItem** al agregado de **Order**.

```
// RIGHT ACCORDING TO DDD--CODE AT THE APPLICATION LAYER OR COMMAND HANDLERS

// The code in command handlers or WebAPI controllers, related only to application stuff
// There is NO code here related to OrderItem object's business logic

myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units);

// The code related to OrderItem params validations or domain rules should
// be WITHIN the AddOrderItem method.
```

En este fragmento, la mayoría de las validaciones o lógica relacionadas con la creación de un objeto **`OrderItem`** estarán bajo el control de la raíz de agregación **`Order`** en el método **`AddOrderItem`**, especialmente las validaciones y la lógica relacionadas con otros elementos en el agregado. Por ejemplo, puede obtener el mismo artículo de producto como resultado de múltiples llamadas a **`AddOrderItem`**. En ese método, puede examinar los artículos del producto y consolidar los mismos artículos del producto en un único objeto **`OrderItem`** con varias unidades. Además, si hay diferentes montos de descuento, pero la identificación del producto es la misma, es probable que aplique el mayor descuento. Este principio se aplica a cualquier otra lógica de dominio para el objeto **`OrderItem`**.

Además, la operación **`new OrderItem(params)`** también será controlada y ejecutada por el método **`AddOrderItem`** desde la raíz del agregado **`Order`**. Por lo tanto, la mayor parte de la lógica o validaciones relacionadas con esa operación (especialmente cualquier cosa que afecte la consistencia entre otras entidades secundarias) estará en un solo lugar, dentro de la raíz de agregación. Ese es el objetivo final del patrón de Agregado.

Cuando utiliza Entity Framework Core 1.1, 2.0 o posterior, se puede expresar mejor una entidad DDD, porque una de las características de EF Core 1.1 o 2.0 es que permite mapear campos además de las propiedades. Esto es útil cuando se protegen colecciones de entidades secundarias o *value objects*. Con esta mejora, puede usar campos privados simples en lugar de propiedades y puede implementar cualquier actualización de la colección en métodos públicos y proporcionar acceso de solo lectura a través del método **`AsReadOnly`**.

En DDD, sólo queremos actualizar la entidad a través de métodos en la entidad (o el constructor) para controlar cualquier invariante y la consistencia de los datos, por lo que las propiedades se definen sólo con un acceso de lectura. Las propiedades están respaldadas por campos privados. Sólo se puede acceder a los miembros privados desde dentro de la clase. Sin embargo, hay una excepción: EF Core sí puede asignar estos campos (para poder devolvern un objeto con los valores adecuados).

Mapeando propiedades de sólo lectura a campos en la tabla

La asignación de propiedades a las columnas de la tabla de la base de datos no es responsabilidad del dominio, sino parte de la capa de infraestructura y persistencia. Mencionamos esto aquí para que esté al tanto de las nuevas capacidades en EF Core 1.1, 2.0 o posterior, relacionadas con la forma en que puede modelar entidades. Detalles adicionales sobre este tema se explican en la sección de infraestructura y persistencia.

Cuando utiliza EF Core 1.0, dentro del **`DbContext`** necesita mapear las propiedades que están definidas sólo lectura a campos reales en la tabla. Esto se hace con el método **`HasField`** de la clase **`PropertyBuilder`**.

Mapeando campos sin propiedades

Con EF Core 1.1 y 2.0 puede mapear columnas a campos de la clase (privados), así, también es posible no usar propiedades para esto. Un caso de uso común para esto es, usar campos privados para un estado interno, al que no es necesario acceder desde fuera de la entidad.

Por ejemplo, en el ejemplo del código anterior de **`OrderAggregate`**, hay varios campos privados, como el campo **`_paymentMethodId`**, que no tienen ninguna propiedad relacionada para un *setter* o *getter*. Ese campo también se puede calcular dentro de la lógica del negocio de la orden y se puede usar a partir de los métodos de la orden, pero también se debe persistir en la base de datos. Para

esto, en EF Core (desde v1.1) hay una forma de mapear un campo sin una propiedad relacionada a una columna en la base de datos. Esto también se explica en la sección Capa de infraestructura de esta guía.

Recursos adicionales

- **Vaughn Vernon. Modeling Aggregates with DDD and Entity Framework.** Note that this is *not* Entity Framework Core.
  https://vaughnvernon.co/?p=879
- **Julie Lerman. Coding for Domain-Driven Design: Tips for Data-Focused Devs**
  https://msdn.microsoft.com/magazine/dn342868.aspx
- **Udi Dahan. How to create fully encapsulated Domain Models**
  http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/

SeedWork (clases base e interfaces reutilizables para el modelo de dominio)

Como se mencionó, en la carpeta de soluciones también puede ver una carpeta **SeedWork**. Esta carpeta contiene clases base personalizadas que puede usar como base para sus entidades de dominio y *value objects*, para evitar código redundante en las clases de objetos del dominio. La carpeta para este tipo de clases se llama SeedWork y no es algo así como *framework*, porque la carpeta contiene sólo un pequeño grupo de clases reutilizables, que realmente no pueden considerarse un *framework*. Seedwork es un término introducido por Michael Feathers y popularizado por Martin Fowler, pero también podría nombrar esa carpeta *Common*, *SharedKernel* o similar.

La Figura 7-12 muestra las clases que forman la carpeta SeedWork del modelo de dominio en el microservicio de pedidos. Tiene algunas clases básicas personalizadas como **`Entity`**, **`ValueObject`** y **`Enumeration`**, además de algunas interfaces. Estas interfaces (**`IRepository`** e **`IUnitOfWork`**) informan a la capa de infraestructura sobre lo que debe implementarse. Esas interfaces también se utilizan a través de inyección de dependencias (DI) desde la capa de la aplicación.

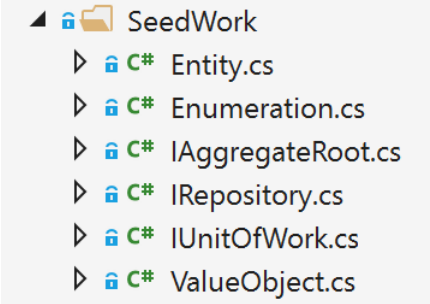


Figura 7-12. *Un ejemplo de de las clases base e interfaces "SeedWork" del modelo de dominio*

Este es el tipo de reutilización de "copiar y pegar" que muchos desarrolladores comparten entre proyectos, no un *framework* formal. Puede tener *SeedWork* en cualquier capa o librería. Sin embargo, si el conjunto de clases e interfaces es lo suficientemente grande, es posible que desee crear una sola librería de clases.

La clase base personalizada Entity

El siguiente código es un ejemplo de una clase **Entidad** base, donde puede colocar código que sea usado de la misma manera por cualquier entidad de dominio, como la ID de entidad, operadores de igualdad, una lista de eventos de dominio por entidad, etc.

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE (1.1, 2.0 or later)
public abstract class Entity
{
    int? _requestedHashCode;
    int _Id;
    private List<INotification> _domainEvents;
    public virtual int Id
    {
        get
        {
            return _Id;
        }
        protected set
        {
            _Id = value;
        }
    }

    public List<INotification> DomainEvents => _domainEvents;
    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }
    public void RemoveDomainEvent(INotification eventItem)
    {
        if (_domainEvents is null) return;
        _domainEvents.Remove(eventItem);
    }

    public bool IsTransient()
    {
        return this.Id == default(Int32);
    }

    public override bool Equals(object obj)
    {
        if (obj == null || !(obj is Entity))
            return false;
        if (Object.ReferenceEquals(this, obj))
```

```
            return true;
        if (this.GetType() != obj.GetType())
            return false;
        Entity item = (Entity)obj;
        if (item.IsTransient() || this.IsTransient())
            return false;
        else
            return item.Id == this.Id;
    }

    public override int GetHashCode()
    {
        if (!IsTransient())
        {
            if (!_requestedHashCode.HasValue)
                _requestedHashCode = this.Id.GetHashCode() ^ 31;
             // XOR for random distribution. See:
              //
// http://blogs.msdn.com/b/ericlippert/archive/2011/02/28/guidelines-and-rules-for-
gethashcode.aspx
            return _requestedHashCode.Value;
        }
        else
            return base.GetHashCode();
    }
    public static bool operator ==(Entity left, Entity right)
    {
        if (Object.Equals(left, null))
            return (Object.Equals(right, null)) ? true : false;
        else
            return left.Equals(right);
    }
    public static bool operator !=(Entity left, Entity right)
    {
        return !(left == right);
    }
}
```

El código anterior, que utiliza una lista de eventos por entidad, se explicará en las próximas secciones, cuando nos enfoquemos en los eventos del dominio.

Contratos de repositorio (interfaces) en la capa del modelo de dominio

Los contratos de repositorio son simplemente interfaces .NET que expresan los requisitos del contrato de los repositorios que se utilizarán para cada agregado.

Los repositorios en sí, con código EF Core o cualquier otra dependencia de infraestructura o librerías (Linq, SQL, etc.), no se deben implementar dentro del modelo de dominio, los repositorios sólo deberían implementar las interfaces que defina en el modelo de dominio.

Un patrón relacionado con esta práctica (colocar las interfaces de repositorios en la capa del modelo de dominio) es el patrón de Interfaz Separada. Como explica Martin Fowler, "Use Interfaz Separada para definir una interfaz en un paquete pero implementarla en otro. De esta forma, un cliente que necesita la dependencia de la interfaz puede desconocer por completo la implementación ".

Seguir el patrón de interfaz separada permite que la capa de aplicación (en este caso, el proyecto de API web para el microservicio) dependa de los requisitos definidos en el modelo de dominio, pero no de una dependencia directa a la capa de infraestructura/persistencia. Además, puede usar Inyección de Dependencias (DI) para aislar la implementación, que está incluida en la capa de infraestructura/persistencia mediante repositorios.

Por ejemplo, en el siguiente ejemplo con la interfaz **`IOrderRepository`**, se define qué operaciones deberá implementar la clase **`OrderRepository`** en la capa de infraestructura. En la implementación actual de la aplicación, el código sólo necesita agregar o actualizar pedidos a la base de datos, ya que las consultas se dividen siguiendo el enfoque simplificado de CQRS.

```
// Defined at IOrderRepository.cs
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);

    void Update(Order order);

    Task<Order> GetAsync(int orderId);
}

// Defined at IRepository.cs (Part of the Domain Seedwork)
public interface IRepository<T> where T : IAggregateRoot
{
    IUnitOfWork UnitOfWork { get; }
}
```

Recursos adicionales

- **Martin Fowler. Separated Interface.**
  http://www.martinfowler.com/eaaCatalog/separatedInterface.html

Implementando objetos de valor (*value objects*)

Como se discutió en secciones anteriores sobre entidades y agregados, la identidad es fundamental para las entidades. Sin embargo, hay muchos objetos y elementos de datos en un sistema que no requieren identidad ni hacer un seguimiento de ésta, como los objetos de valor (*value objects*).

Un value object puede hacer referencia a otras entidades. Por ejemplo, en una aplicación que genera una ruta que describe cómo llegar de un punto a otro, esa ruta sería un objeto de valor. Sería una instantánea de puntos en una ruta específica, pero esta ruta sugerida no tendría una identidad, aunque internamente podría referirse a otras entidades como **`City`**, **`Road`**, etc.

La Figura 7-13 muestra el objeto valor **`Address`** dentro del agregado **`Order`**.

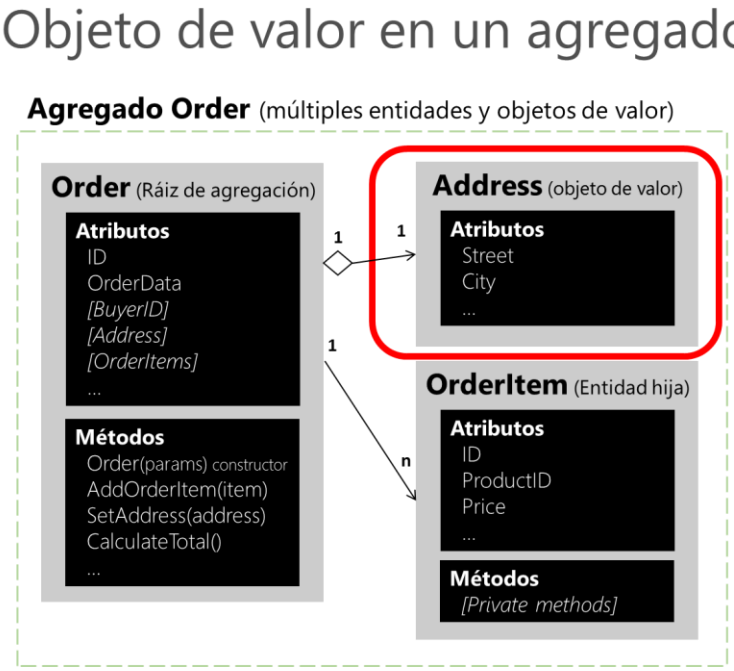


Figura 7-13. *Objeto de valor* `Address` *dentro del agregado* `Order`

Como se muestra en la figura 7-13, una entidad generalmente se compone de múltiples atributos. Por ejemplo, la entidad **`Order`** puede modelarse como una entidad con una identidad, compuesta internamente de un conjunto de atributos como **`OrderId`**, **`OrderDate`**, **`OrderItems`**, etc. Pero la dirección, que es simplemente un valor complejo compuesto de **`Country`**, **`Street`**, **`City`**, etc. y no tiene identidad en este dominio, por lo tanto, debe modelarse y tratarse como un objeto de valor.

Características importantes de los *value objects*

Hay dos características principales para los *value objects*:

- No tienen identidad.
- Son inmutables.

La primera característica ya fue discutida. La inmutabilidad es un requisito importante. Los valores de un objeto de valor deben ser inmutables una vez que se crea el objeto. Por lo tanto, cuando se

construye el objeto, debe proporcionar los valores requeridos, pero no debe permitir que cambien durante la vida del objeto.

Los *value objects* le permiten realizar ciertos trucos para mejorar el rendimiento, gracias a su naturaleza inmutable. Esto es especialmente cierto en sistemas donde puede haber miles de instancias de *value objects*, muchas de las cuales tienen los mismos valores. Su naturaleza inmutable les permite ser reutilizados, pueden ser objetos intercambiables, ya que sus valores son los mismos y no tienen identidad. Este tipo de optimización a veces puede hacer una diferencia entre el software que se ejecuta lentamente y el software con buen rendimiento. Por supuesto, todos estos casos dependen del entorno de la aplicación y del contexto de despliegue.

Implementación de *value objects* en C#

En términos de implementación, puede tener una clase base de *value objects* que tenga métodos utilitarios básicos como igualdad basada en la comparación entre todos los atributos (ya que un objeto de valor no debe basarse en la identidad) y otras características fundamentales. El siguiente ejemplo muestra una clase base de objeto de valor utilizada en el microservicio de pedidos de eShopOnContainers.

```
public abstract class ValueObject
{
    protected static bool EqualOperator(ValueObject left, ValueObject right)
    {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
        {
            return false;
        }
        return ReferenceEquals(left, null) || left.Equals(right);
    }

    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }

    protected abstract IEnumerable<object> GetAtomicValues();

    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
        }
        ValueObject other = (ValueObject)obj;
        IEnumerator<object> thisValues = GetAtomicValues().GetEnumerator();
        IEnumerator<object> otherValues = other.GetAtomicValues().GetEnumerator();
        while (thisValues.MoveNext() && otherValues.MoveNext())
        {
            if (ReferenceEquals(thisValues.Current, null) ^
                                    ReferenceEquals(otherValues.Current, null))
            {
                return false;
            }
            if (thisValues.Current != null &&
                                !thisValues.Current.Equals(otherValues.Current))
            {
                return false;
            }
        }
        return !thisValues.MoveNext() && !otherValues.MoveNext();
    }
    // Other utilility methods
}
```

Puede utilizar esta clase al implementar su objeto de valor real, como el **Address** que se muestra en el siguiente ejemplo:

```
public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }
    public Address(string street, string city, string state,
                   string country, string zipcode)
    {
        Street = street;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;
    }

    protected override IEnumerable<object> GetAtomicValues()
    {
        // Using a yield return statement to return each element one at a time
        yield return Street;
        yield return City;
        yield return State;
        yield return Country;
        yield return ZipCode;
    }
}
```

Puede ver cómo esta implementación del objeto de valor **Address** no tiene identidad y, por lo tanto, no se ha registrado ID, ni en la clase **Address** ni en la clase **ValueObject**.

Antes de EF Core 2.0 no era posible manejar una clase (en EF) sin campo de identidad (ID), pero esta nueva facilidad ayuda en gran medida para implementar *value objects*, como veremos en la sección siguiente.

Cómo persistir *value objects* en la base de datos con EF Core 2.0

Acabamos de ver cómo definir un objeto de valor en un modelo de dominio. Pero, ¿cómo se puede persistir en la base de datos a través de Entity Framework Core, que generalmente apunta a entidades con identidad?

Antecedentes y enfoques anteriores, usando EF Core 1.1

A modo de historia, una limitación al usar EF Core 1.0 y 1.1 era que no se podían usar tipos complejos como lo permite EF 6.x en el .NET Framework tradicional. Por lo tanto, si usa EF Core 1.0 o 1.1, necesita almacenar un objeto de valor como una entidad EF con un campo ID. Entonces, para que pareciera más un objeto de valor sin identidad, podía ocultar su ID para dejar claro que la identidad de un objeto de valor no es importante en el modelo de dominio. Puede ocultar esa ID usándola como una shadow property. Dado que la configuración para ocultar el ID en el modelo se hace en el nivel de infraestructura de EF, sería algo transparente para su modelo de dominio.

En la versión inicial de eShopOnContainers (.NET Core 1.1), la identificación oculta que necesita la infraestructura de EF Core se implementó de la siguiente manera en el nivel de **DbContext**, utilizando la *Fluent API* en el proyecto de infraestructura. Por lo tanto, el ID estaba oculto desde el punto de vista del modelo de dominio, pero aún estaba presente en la infraestructura.

```
// Old approach with EF Core 1.1
// Fluent API within the OrderingContext:DbContext in the Infrastructure project
void ConfigureAddress(EntityTypeBuilder<Address> addressConfiguration)
{
    addressConfiguration.ToTable("address", DEFAULT_SCHEMA);

    addressConfiguration.Property<int>("Id")  // Id is a shadow property
        .IsRequired();
    addressConfiguration.HasKey("Id");   // Id is a shadow property
}
```

Sin embargo, la persistencia de ese objeto de valor en la base de datos se realizaba como una entidad regular en una tabla diferente.

Con EF Core 2.0 hay nuevas y mejores formas de persistir los *value objects*.

Persistiendo los value objects como "owned entities" en EF Core 2.0

Aun cuando todavía hay algunas brechas entre el patrón canónico de *value objects* en DDD y las *owned entities* (la traducción de "owned entity" sería como una entidad sin identidad, que le pertenece o está supeditada a otra) en EF Core, actualmente es la mejor forma de persistir *value objects* con EF Core 2.0. Puede ver las limitaciones al final de esta sección.

La característica de *owned entity* se agregó a EF Core en la versión 2.0.

El tipo *owned entity* le permite mapear tipos que no tienen su propia identidad definida explícitamente en el modelo del dominio y se usan como propiedades (como un objeto de valor) dentro de cualquiera de sus entidades. Un tipo *owned entity* comparte el mismo tipo del CLR que cualquier otro tipo de entidad (es decir, es una clase como cualquier otra). La entidad propietaria contiene la propiedad de navegación que define una *owned entity*. Cuando se hace una consulta a la entidad propietaria, se incluyen, por defecto, las *owned entities* como cualquier otra propiedad.

Con sólo mirar el modelo de dominio, parece que una *owned entity* no tiene ninguna identidad, pero, por debajo, si la tienen, lo que pasa es que la propiedad de navegación del propietario es parte de la identidad.

La identidad de instancias de tipos *owned entities* no es completamente suya, precisamente porque están supeditadas a la entidad propietaria. Consta de tres componentes:

- La identidad del propietario
- La propiedad de navegación correspondiente y
- En el caso de colecciones de *owned entities*, un componente independiente (Todavía no soportados en EF Core 2.0).

Por ejemplo, en el modelo de dominio de pedidos en eShopOnContainers, como parte de la entidad **Order**, el objeto de valor **Address** se implementa como una propiedad *owned entity* dentro de la entidad propietaria, que es **Order**. **Address** es un tipo sin identidad definida en el modelo de

dominio. Se utiliza como una propiedad del tipo de orden para especificar la dirección de envío para un pedido en particular.

Por convención, se creará una clave principal paralela para la *owned entity* y se asignará a la misma tabla que el propietario, usando *table splitting*. Esto permite usar *owned entities* de forma similar a como se usan los tipos complejos en EF6 en el .NET Framework tradicional.

Es importante tener en cuenta que los *owned entities* nunca se descubren por convención en EF Core, por lo que debe declararlos explícitamente.

En eShopOnContainers, en **`OrderingContext.cs`**, dentro del método **`OnModelCreating()`**, se están aplicando varias configuraciones de la infraestructura. Una de ellas está relacionada con la entidad **`Order`**.

```
// Part of the OrderingContext.cs class at the Ordering.Infrastructure project
//
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new ClientRequestEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new PaymentMethodEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderItemEntityTypeConfiguration());
    //...Additional type configurations
}
```

En el siguiente código es donde se define la infraestructura de persistencia para la entidad **`Order`**.

```
// Part of the OrderEntityTypeConfiguration.cs class
//
public void Configure(EntityTypeBuilder<Order> orderConfiguration)
{
    orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
    orderConfiguration.HasKey(o => o.Id);
    orderConfiguration.Ignore(b => b.DomainEvents);
    orderConfiguration.Property(o => o.Id)
        .ForSqlServerUseSequenceHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

    //Address value object persisted as owned entity in EF Core 2.0
    orderConfiguration.OwnsOne(o => o.Address);

    orderConfiguration.Property<DateTime>("OrderDate").IsRequired();

    //...Additional validations, constraints and code...
    //...
}
```

En el código anterior, la línea **`orderConfiguration.OwnsOne (o => o.Address)`** especifica que la propiedad **`Address`** es una propiedad *owned entity* del tipo **`Order`**.

Según la convención de EF Core, las columnas de la base de datos para las propiedades de la *owned entity*, se llamarán como **`EntityProperty_OwnedEntityProperty`**. Por lo tanto, las propiedades internas de Dirección aparecerán en la tabla **`Orders`** con los nombres **`Address_Street`**, **`Address_City`** (y así sucesivamente para **`State`**, **`Country`** y **`ZipCode`**).

Puede agregar el método **Property().HasColumnName()** a la configuración, para cambiar el nombre de esas columnas. En este caso, como **`Address`** es una propiedad pública, las asignaciones serían como las siguientes.

```
orderConfiguration.OwnsOne(p => p.Address)
                            .Property(p=>p.Street).HasColumnName("ShippingStreet");

orderConfiguration.OwnsOne(p => p.Address)
                            .Property(p=>p.City).HasColumnName("ShippingCity");
```

Es posible encadenar el método **`OwnsOne`** en con la *Fuent API*. En el siguiente ejemplo hipotético, **`OrderDetails`** posee **`BillingAddress`** y **`ShippingAddress`**, que son ambas del tipo **`Address`**. Entonces **`OrderDetails`** es una *owned entity* de **`Order`**.

```
orderConfiguration.OwnsOne(p => p.OrderDetails, cb =>
                                {
                                    cb.OwnsOne(c => c.BillingAddress);
                                    cb.OwnsOne(c => c.ShippingAddress);
                                });
//...
//...
public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}

public class OrderDetails
{
    public StreetAddress BillingAddress { get; set; }
    public StreetAddress ShippingAddress { get; set; }
}

public class StreetAddress
{
    public string Street { get; set; }
    public string City { get; set; }
}
```

Detalles adicionales de los tipos owned entities

- Las *owned entities* se definen cuando configura una propiedad de navegación, de un tipo particular, usando la *Fluent API* **`OwnsOne`**
- La definición de una *owned entity* en la metadata de nuestro modelo es una combinación de: el tipo de propietario, la propiedad de navegación y el tipo de la *owned entity*
- La identidad (clave) de una instancia de una *owned entity* es una combinación de la identidad del propietario y la definición de la *owned entity* (el punto anterior)

Posibilidades de las *Owned Entities*:

- Una *owned entity* puede hacer referencia a otras entidades, ya sean *owned entities* (*owned entities* anidadas) o propiedades de navegación regulares hacia otras entidades)
- Puede mapear propiedades diferentes del mismo tipo, como diferentes *owned entities*, en la misma entidad propietario.
- La división de tablas se configura por convención, pero puede optar por mapear la *owned entity* a una tabla diferente usando **`ToTable`**
- Las propiedades que son *owned entities* se cargan siempre (*eager loading*) al consultar la entidad, no es necesario usar **`Include()`** en las consultas, como, por ejemplo, hace falta con las propiedades de navegación o las colecciones

Limitaciones de las *Owned Entities*:

- No puede crear un **`DbSet<T>`** de un tipo propiedad (por diseño)
- No puede llamar a **`ModelBuilder.Entity<T>()`** para *owned entities* (actualmente por diseño)
- Aún no hay colecciones de *owned entities* (pero serán soportados en versiones posteriores a EF Core 2.0)
- No hay soporte para configurarlos a través de un atributo
- No se soportan *owned entities* opcionales (*nullables*) mapeados en la misma tabla del propietario, es decir, usando *table splitting*, porque el mapeo se hace por cada propiedad de la *owned entity* y no hay un campo para manejar el null de la propiedad compleja.
- No se soporta el mapeo de herencia para las *owned entities*, pero debería poder mapear dos subtipos de la misma jerarquía como *owned types* diferentes. EF Core no evaluará el hecho de que son parte de la misma jerarquía

Diferencias principales con los tipos complejos de EF6

- El *table splitting* es opcional, es decir, se pueden mapear opcionalmente en una tabla separada y seguir siendo *owned entities*
- Pueden hacer referencia a otras entidades (es decir, pueden actuar como el lado dependiente de las relaciones con otros tipos que no son *owned entities*)

Recursos adicionales

- **Martin Fowler. ValueObject pattern**
  https://martinfowler.com/bliki/ValueObject.html
- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software.**
  (Book; includes a discussion of value objects)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/

- **Vaughn Vernon. Implementing Domain-Driven Design.** (Book; includes a discussion of value objects) https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/
- **Shadow Properties** https://docs.microsoft.com/ef/core/modeling/shadow-properties
- **Complex types and/or value objects**. Discussion in the EF Core GitHub repo (Issues tab) https://github.com/aspnet/EntityFramework/issues/246
- **ValueObject.cs.** Base value object class in eShopOnContainers. https://github.com/dotnet/eShopOnContainers/blob/masterdev/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs
- **Address class**. Sample value object class in eShopOnContainers. https://github.com/dotnet/eShopOnContainers/blob/masterdev/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Address.cs

Usando clases de Enumeración en vez de los enums de C#

Las enumeraciones (*enums* para abreviar) son un *wrapper* de lenguaje alrededor de un tipo entero. Es posible que desee limitar su uso a cuando está almacenando un valor de un conjunto cerrado de valores. La clasificación basada en el género (por ejemplo, masculino, femenino, desconocido) o los tamaños (S, M, L, XL) son buenos ejemplos. Usar enumeraciones para el flujo de control o abstracciones más robustas puede ser un *code smell* (una parte del código que no huele bien). Este tipo de uso conducirá a un código frágil, con muchas sentencias de control de flujo que verifican los valores de la enumeración.

En cambio, puede crear clases de enumeración que habiliten todas las características expresivas de un lenguaje orientado a objetos.

Sin embargo, este no es un tema crítico y, en muchos casos, por simplicidad, aún puede usar tipos enum regulares si lo prefiere.

Implementando clases de Enumeración

El microservicio de pedidos en eShopOnContainers proporciona una implementación de la clase base **`Enumeration`** de ejemplo, como se muestra en el siguiente código:

```
public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration()
    {
    }
    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }
    public override string ToString()
    {
        return Name;
    }
    public static IEnumerable<T> GetAll<T>() where T : Enumeration, new()
    {
        var type = typeof(T);
        var fields = type.GetTypeInfo().GetFields(BindingFlags.Public |
                                                  BindingFlags.Static |
                                                  BindingFlags.DeclaredOnly);
        foreach (var info in fields)
        {
            var instance = new T();
            var locatedValue = info.GetValue(instance) as T;

            if (locatedValue != null)
            {
                yield return locatedValue;
            }
        }
    }


    public override bool Equals(object obj)
    {
        var otherValue = obj as Enumeration;

        if (otherValue == null)
        {
            return false;
        }

        var typeMatches = GetType().Equals(obj.GetType());
        var valueMatches = Id.Equals(otherValue.Id);

        return typeMatches && valueMatches;
    }
```

```
    public int CompareTo(object other)
    {
        return Id.CompareTo(((Enumeration)other).Id);
    }

    // Other utility methods ...
}
```

Puede usar esta clase como un tipo base en cualquier entidad u objeto de valor, como para la siguiente clase de Enumeración **CardType**.

```
public class CardType : Enumeration
{
    public static CardType Amex = new CardType(1, "Amex");
    public static CardType Visa = new CardType(2, "Visa");
    public static CardType MasterCard = new CardType(3, "MasterCard");

    protected CardType() { }
    public CardType(int id, string name)
        : base(id, name)
    {
    }

    public static IEnumerable<CardType> List()
    {
        return new[] { Amex, Visa, MasterCard };
    }
    // Other util methods
}
```

Recursos adicionales

- **Enum's are evil—update**
  http://www.planetgeek.ch/2009/07/01/enums-are-evil/
- **Daniel Hardman. How Enums Spread Disease — And How To Cure It**
  https://codecraft.co/2012/10/29/how-enums-spread-disease-and-how-to-cure-it/
- **Jimmy Bogard. Enumeration classes**
  https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/
- **Steve Smith. Enum Alternatives in C#**
  http://ardalis.com/enum-alternatives-in-c
- **Enumeration.cs.** Base Enumeration class in eShopOnContainers
  https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/SeedWork/Enumeration.cs
- **CardType.cs**. Sample Enumeration class in eShopOnContainers.
  https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/AggregatesModel/BuyerAggregate/CardType.cs

Diseñando las validaciones en la capa de dominio

En DDD, las reglas de validación se pueden considerar invariantes. La principal responsabilidad de un agregado es imponer invariantes entre los cambios de estado, para todas las entidades dentro de ese agregado.

Las entidades de dominio siempre deben ser entidades válidas. Hay un cierto número de invariantes para un objeto que siempre debería ser cierto. Por ejemplo, un objeto de artículo de pedido siempre debe tener una cantidad que debe ser un entero positivo, más el nombre y el precio del artículo. Por lo tanto, hacer cumplir las invariantes es responsabilidad de las entidades de dominio (especialmente de la raíz de agregación) y un objeto de entidad no debería poder existir sin ser válido. Las reglas invariables simplemente se expresan como contratos y las excepciones o notificaciones se generan cuando se infringen.

El razonamiento detrás de esto es que se producen muchos errores porque los objetos están en un estado al que nunca debieron haber llegado. La siguiente es una buena explicación de Greg Young en una discusión en línea:

> *Propongamos que ahora tenemos un SendUserCreationEmailService que toma un UserProfile ... ¿cómo podemos racionalizar en ese servicio que Name no sea nulo? ¿Lo revisamos de nuevo? O más probablemente ... simplemente no nos molestamos en verificar y "esperamos lo mejor" -espera que alguien se haya tomado la molestia de validarlo antes de enviárselo. Por supuesto, al usar TDD, una de las primeras pruebas que deberíamos escribir es que si envío un cliente con un nombre nulo debería generar un error. Pero una vez que comenzamos a escribir este tipo de pruebas una y otra vez nos damos cuenta ... "espere si nunca permitiéramos que el nombre se vuelva nulo, no tendríamos todas estas pruebas"*

Implementado validaciones en la capa del modelo de dominio

Las validaciones generalmente se implementan en constructores de entidades de dominio o en métodos que pueden actualizar la entidad. Existen múltiples formas de implementar validaciones, como verificar datos y generar excepciones si la validación falla. También hay patrones más avanzados, como el uso del patrón de Especificación para las validaciones y el patrón de Notificación para devolver una colección de errores en lugar de elevar una excepción para cada validación a medida que se produce.

Validando condiciones y elevando excepciones

El siguiente ejemplo de código muestra el enfoque más simple para la validación en una entidad de dominio al generar una excepción. En la tabla de referencias al final de esta sección, puede ver enlaces a implementaciones más avanzadas basadas en los patrones que hemos discutido anteriormente.

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
```

Un mejor ejemplo demostraría la necesidad de garantizar que el estado interno no cambiara o que todas las mutaciones de un método ocurrieran. Por ejemplo, la siguiente implementación dejaría el objeto en un estado no válido:

```
Public void SetAddress(string line1, string line2,
                       string city, string state, int zip)
{
    _shipingAddress.line1 = line1 ?? throw new ...
    _shippingAddress.line2 = line2;
    _shippingAddress.city = city ?? throw new ...
    _shippingAddress.state = (IsValid(state) ? state : throw new …);
}
```

Si el valor del estado no es válido, la primera línea de dirección y la ciudad ya se han cambiado. Eso podría hacer que la dirección sea inválida.

Se puede usar un enfoque similar en el constructor de la entidad, generando una excepción para garantizar que sea válida una vez que se haya creado.

Usando atributos de validación en el modelo, basados en anotaciones de datos

Los *data annotations*, como los atributos **`Required`** o **`MaxLength`**, se pueden utilizar para configurar las propiedades de los campos de la base de datos en EF Core, como se explica en detalle en la sección Mapeo de tablas, pero ya no funcionan para la validación de entidades en EF Core (ni el método **`IValidatableObject.Validate`** tampoco), como lo han hecho desde EF 4.x en .NET Framework.

Los *data annotations* y la interfaz **`IValidatableObject`** se pueden seguir utilizando para la validación del modelo durante el *model binding*, antes de la invocación de las acciones del controlador como siempre, pero ese modelo está pensado para ser un ViewModel o DTO y eso es una preocupación de MVC o API no del modelo de dominio.

Una vez aclarada la diferencia conceptual, todavía puede utilizar *data annotations* e **`IValidatableObject`** en la clase de entidad para la validación, si sus acciones reciben como parámetro un objeto de clase de entidad, lo que no recomendamos. En ese caso, la validación se producirá en el momento del *model binding*, justo antes de invocar la acción y se puede revisar la propiedad **`ModelState.isValid`** del controlador para comprobar el resultado, pero eso también ocurre en el controlador, no antes de persistir el objeto de la entidad en el DbContext, como se ha hecho desde EF 4.x.

Aún así, puede implementar una validación personalizada en la clase de entidad utilizando los *data annotations* y el método **`IValidatableObject.Validate`**, reemplazando el método **`SaveChanges`** del DbContext.

Puede ver un ejemplo de implementación de validación de entidades **`IValidatableObject`** en este comentario de GitHub. Ese ejemplo no hace validaciones basadas en atributos, pero deberían ser fáciles de implementar usando *reflection* en el mismo *override*.

Sin embargo, desde un punto de vista DDD, el modelo de dominio se mantiene mejor con el uso de excepciones en los métodos de comportamiento de su entidad, o implementando los patrones de Especificación y Notificación para hacer cumplir las reglas de validación.

Puede tener sentido utilizar anotaciones de datos en la capa de aplicación en clases de *ViewModel* (en lugar de entidades de dominio) que aceptarán entrada, para permitir la validación del modelo dentro de la capa de la interfaz de usuario. Sin embargo, esto no exime de hacer las validaciones dentro del modelo de dominio.

Validando entidades implementando los patrones de Especificación y Notificación

Finalmente, un enfoque más elaborado para implementar validaciones en el modelo de dominio es implementar el patrón de Especificación junto con el patrón de Notificación, como se explica en algunos de los recursos adicionales que se enumeran más adelante.

Vale la pena mencionar que también puede usar sólo uno de esos patrones, por ejemplo, validando manualmente con las sentencias de control, pero usando el patrón de notificación para apilar y devolver una lista de errores de validación.

Usando validación diferida en el dominio

Hay varios enfoques para tratar con validaciones diferidas en el dominio. Vaughn Vernon discute estos en su libro Implementing Domain-Driven Design.

Validación en dos pasos

Considere también la validación en dos pasos. Use la validación a nivel de campo en los DTOs de comandos (Objetos de Transferencia de Datos, por ejemplo, un *ViewModel* de actualización) y la validación a nivel de dominio dentro de sus entidades. Puede hacer esto devolviendo un objeto como resultado en lugar de elevar excepciones, para facilitar el tratamiento de los errores de validación.

Al usar validaciones de campo con anotaciones de datos, no se duplica la definición de validación. La ejecución, sin embargo, puede ocurrir tanto del lado del servidor como del lado del cliente en el caso de los DTOs (comandos y *ViewModels*).

Recursos adicionales

- **Rachel Appel. Introducción a la validación del modelo en MVC de ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/mvc/models/validation
- **Rick Anderson. Adición de validación**
  https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation
- **Martin Fowler. Replacing Throwing Exceptions with Notification in Validations**
  https://martinfowler.com/articles/replaceThrowWithNotification.html
- **Specification and Notification Patterns**
  https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns
- **Lev Gorodinski. Validation in Domain-Driven Design (DDD)**
  http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/
- **Colin Jack. Domain Model Validation**
  http://colinjack.blogspot.com/2008/03/domain-model-validation.html
- **Jimmy Bogard. Validation in a DDD world**
  https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/

Validación en el lado del cliente (en las capas de presentación)

Incluso cuando la fuente de la verdad es el modelo de dominio y, en última instancia, debe tener validación en el nivel de modelo de dominio, la validación aún se puede manejar tanto en el nivel de modelo de dominio (del lado del servidor) como en el lado del cliente.

La validación del lado del cliente es una gran conveniencia para los usuarios. Ahorra tiempo que de lo contrario pasarían esperando un viaje de ida y vuelta al servidor que podría devolver errores de validación. En términos del negocio, incluso unas pocas fracciones de segundos se multiplican cientos de veces al día, lo que se traduce en mucho tiempo, gastos y frustración. La validación directa e inmediata permite a los usuarios trabajar de manera más eficiente y producir entradas y salidas de mejor calidad.

Así como el *ViewModel* y el modelo de dominio son diferentes, la validación del *ViewModel* y la validación del modelo de dominio pueden ser similares, pero tienen un propósito diferente. Si le preocupa el principio DRY (Don't Repeat Yourself - no repetirse), tenga en cuenta que en este caso la reutilización del código también puede implicar acoplamiento y, en las aplicaciones empresariales, es más importante no acoplar el lado del servidor al lado del cliente que seguir el principio DRY

Incluso cuando usa la validación del lado del cliente, siempre debe validar sus comandos o DTOs de entrada en el código del servidor, ya que las API del servidor son un posible vector de ataque. Por lo general, hacer ambas cosas es la mejor opción porque si tiene una aplicación cliente, desde la perspectiva de la experiencia del usuario (UX), es mejor ser proactivo y no permitir que el usuario ingrese información inválida.

Por lo tanto, valide los *ViewModels* en el código del lado del cliente. También puede validar los DTOs o comandos de salida del cliente antes de enviarlos a los servicios.

La implementación de la validación del lado del cliente depende del tipo de aplicación cliente que esté creando. Sería diferente si está validando datos en una aplicación web MVC, con la mayor parte del código en .NET, una aplicación web SPA con esa validación codificada en JavaScript o TypeScript, o una aplicación móvil nativa con Xamarin y C#.

Recursos adicionales

Validación en aplicaciones móviles con Xamarin

- **Validate Text Input And Show Errors**
  https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/
- **Validation Callback**
  https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/

Validación en aplicaciones ASP.NET Core

- **Rick Anderson. Adición de validación**
  https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation

Validación en aplicaciones Web SPA (Angular 2, TypeScript, JavaScript)

- **Ado Kukic. Angular 2 Form Validation**
  https://scotch.io/tutorials/angular-2-form-validation

- **Form Validation**
  https://angular.io/docs/ts/latest/cookbook/form-validation.html
- **Validation.** Breeze documentation.
  http://breeze.github.io/doc-js/validation.html

En resumen, estos son los conceptos más importantes en relación con las validaciones:

- Las entidades y los agregados deben asegurar su propia consistencia y ser "siempre válidos"
- Si le parece que una entidad necesita entrar en un estado inválido, considere usar un modelo diferente, por ejemplo, un DTO temporal, hasta que cree la entidad final en el dominio.
- Si necesita crear varios objetos relacionados, como un agregado y ellos sólo son válidos una vez que estén creados todos, considere usar el patrón Factoría
- En la mayoría de los casos en beneficioso tener validaciones duplicadas del lado del cliente, porque la aplicación puede ser proactiva.

Eventos del dominio: diseño e implementación

Use eventos de dominio para implementar explícitamente los efectos secundarios de los cambios dentro de su dominio. En otras palabras, y usando la terminología DDD, use eventos de dominio para implementar explícitamente efectos secundarios en múltiples agregados. Opcionalmente, para una mejor escalabilidad y un menor impacto por los bloqueos de bases de datos, use la consistencia eventual entre los agregados dentro del mismo dominio.

¿Qué es un evento de dominio?

Un evento es algo que sucedió en el pasado. Un evento de dominio es, lógicamente, algo que sucedió en un dominio en particular y algo que desea que otras partes del mismo dominio (en el mismo proceso) conozcan y a lo que puedan reaccionar.

Un beneficio importante de los eventos de dominio es que, los efectos secundarios después de que algo sucedió en un dominio, se pueden expresar explícitamente en lugar de implícitamente. Esos efectos secundarios deben ser consistentes, por lo que todas las operaciones relacionadas con la tarea negocio sucedan, o no ocurra ninguna de ellas. Además, los eventos de dominio permiten una mejor separación de las responsabilidades entre las clases dentro del mismo dominio.

Por ejemplo, si sólo está utilizando Entity Framework y entidades o incluso agregados, si tiene que haber efectos secundarios provocados por un caso de uso, estos se implementarán como un concepto implícito en el código relacionado después de que algo sucedió (es decir, debe ver el código para saber que hay efectos secundarios). Pero, si sólo ve ese código, es posible que no sepa si el efecto que está viendo es parte de la operación principal o si realmente es un efecto secundario. Por otro lado, el uso de eventos de dominio hace que el concepto sea explícito y parte del lenguaje ubicuo. Por ejemplo, en la aplicación eShopOnContainers, crear un pedido no es sólo sobre el pedido, también se actualiza o crea un agregado de comprador, basado en el usuario original, porque el usuario no es un comprador hasta que hace un pedido. Si usa eventos de dominio, puede expresar explícitamente esa regla de dominio basada en el lenguaje ubicuo proporcionado por los expertos de dominio.

Los eventos de dominio son algo similares a los eventos de mensajería, con una diferencia importante. Con mensajería real, colas de mensajes, intermediarios de mensajes o un bus de servicio que usa AMPQ, un mensaje siempre se envía de forma asíncrona y se realiza entre procesos e, incluso, máquinas. Esto es útil para integrar múltiples *Bounded Contexts*, microservicios o incluso diferentes aplicaciones. Sin embargo, con eventos de dominio, desea generar un evento a partir de la operación que está ejecutando actualmente, pero desea que se produzcan efectos secundarios dentro del mismo dominio.

Los eventos de dominio y sus efectos secundarios (las acciones desencadenadas posteriormente que son administradas por manejadores de eventos) deben ocurrir casi de inmediato, generalmente en el mismo proceso y dentro del mismo dominio. Por lo tanto, los eventos de dominio podrían ser síncronos o asíncronos. Los eventos de integración, sin embargo, siempre deben ser asíncronos.

Eventos de dominio versus eventos de integración

Semánticamente, los eventos de dominio e integración son la misma cosa: notificaciones sobre algo que acaba de suceder. Sin embargo, su implementación debe ser diferente. Los eventos de dominio

son sólo mensajes enviados a un despachador de eventos de dominio, que podría implementarse como un mediador en memoria basado en un contenedor IoC o cualquier otro método.

Por otro lado, el propósito de los eventos de integración es propagar transacciones y actualizaciones confirmadas a subsistemas adicionales, ya sean otros microservicios, *Bounded Contexts* o incluso aplicaciones externas. Por lo tanto, deben ocurrir sólo si la entidad se persiste exitosamente, ya que en muchos escenarios si esto falla, la operación, en realidad, nunca ocurrió efectivamente.

Además, y como se mencionó, los eventos de integración se deben basar en la comunicación asíncrona entre múltiples microservicios (otros *Bounded Contexts*) o incluso sistemas/aplicaciones externas. Por lo tanto, la interfaz del bus de eventos necesita alguna infraestructura que permita la comunicación distribuida entre procesos y servicios, potencialmente remotos. Puede basarse en un bus de servicio comercial, colas, una base de datos compartida utilizada como buzón de correo o cualquier otro sistema de mensajería distribuido e, idealmente, de tipo *push*.

Eventos de dominio como la forma preferida de iniciar efectos secundarios entre múltiples agregados dentro del mismo dominio

Si la ejecución de un comando relacionado con una instancia agregada requiere que se ejecuten reglas de dominio adicionales en uno o más agregados adicionales, debería diseñar e implementar los efectos secundarios para que ser desencadenarán por eventos de dominio. Como se muestra en la figura 7-14, y como uno de los casos de uso más importantes, un evento de dominio debe usarse para propagar cambios de estado a través de múltiples agregados dentro del mismo modelo de dominio.

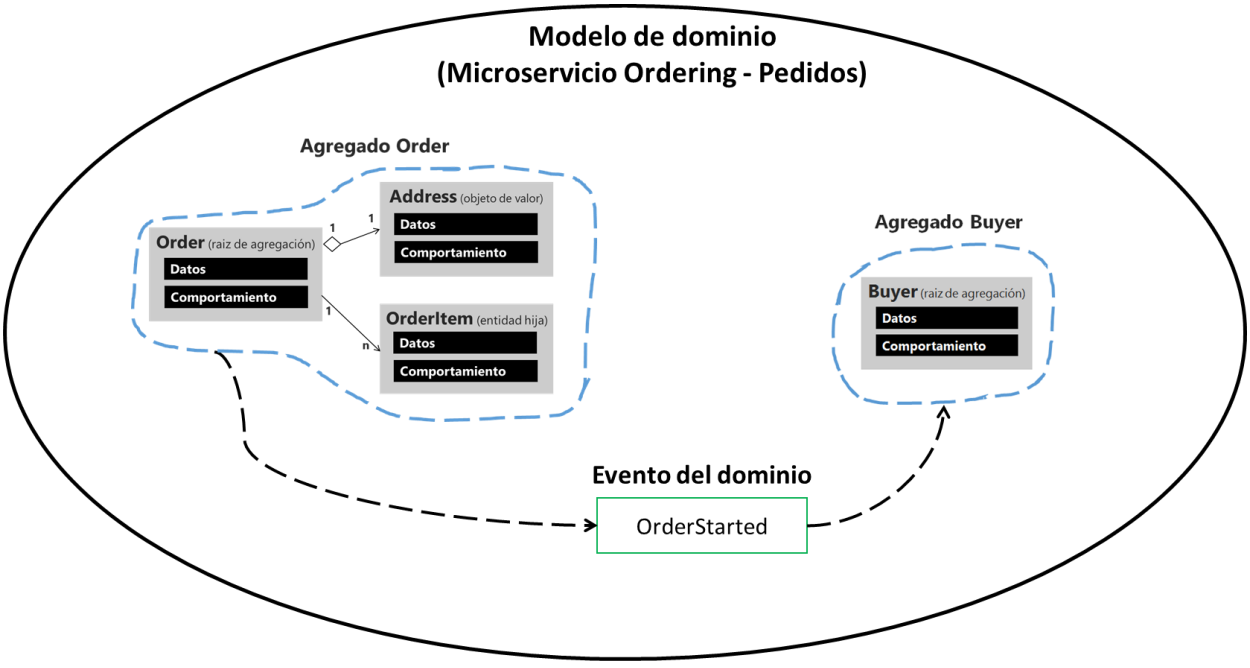


***Figura 7-14**. Eventos del dominio para reforzar la consistencia entre agregados en el mismo dominio*

En la figura, cuando el usuario inicia un pedido, el evento de dominio **`OrderStarted`** activa la creación de un objeto de **`Buyer`** en el microservicio de pedidos, basado en la información de usuario original del microservicio de identidad (con información proporcionada en el comando

CreateOrder). El evento de dominio es generado, en primera instancia, por el agregado de orden cuando se crea ésta.

Alternativamente, puede hacer que la raíz de agregación se suscriba a eventos generados por miembros de sus entidades secundarias. Por ejemplo, cada entidad secundaria **OrderItem** puede generar un evento cuando el precio del artículo es más alto que un monto específico, o cuando la cantidad del artículo del producto es demasiado alta. La raíz de agregación puede recibir esos eventos y realizar un cálculo o agregación global.

Es importante entender que esta comunicación basada en eventos no se implementa directamente dentro de los agregados; necesita implementar manejadores de eventos de dominio. El manejo de los eventos del dominio es responsabilidad de la aplicación. La capa de modelo de dominio sólo debe centrarse en la lógica de dominio, cosas que un experto en dominio entendería, no la infraestructura de aplicaciones como los manejadores y las acciones de persistencia y de efectos secundarios que usan repositorios. Por lo tanto, el nivel de la capa de aplicación es donde debe tener manejadores de eventos que desencadenen acciones cuando se genere un evento de dominio.

Los eventos de dominio también se pueden utilizar para desencadenar cualquier cantidad de acciones de la aplicación y, lo que es más importante, debe estar abierto para aumentar esa cantidad en el futuro de una manera desacoplada. Por ejemplo, cuando se inicia el pedido, es posible que desee publicar un evento de dominio para propagar esa información a otros agregados o incluso para generar acciones de aplicación, como notificaciones.

El punto clave es la cantidad abierta de acciones que se ejecutarán cuando se produzca un evento de dominio. Eventualmente, las acciones y reglas en el dominio y la aplicación crecerán. La complejidad o la cantidad de acciones con efectos secundarios cuando algo ocurra aumentará, pero si su código se combina con "pegamento" (es decir, simplemente instanciando objetos con **new** en C#), cada vez que necesite agregar una nueva acción necesitarás cambiar el código original. Esto podría generar nuevos errores, ya que con cada nuevo requerimiento necesitaría cambiar el flujo de código original. Esto va en contra del principio Open/Closed de SOLID. Además, la clase original que estaba orquestando las operaciones crecería y crecería, lo que va en contra del Principio de Responsabilidad Única (SRP).

Por otro lado, si usa eventos de dominio, puede crear una implementación desacoplada y detallada, segregando responsabilidades usando este enfoque:

1. Enviar un comando (por ejemplo, **CreateOrder**).
2. Recibir el comando en un manejador de comando.
   - Ejecutar una transacción en un agregado.
   - (Opcional) Lanzar eventos de dominio para efectos secundarios (por ejemplo, **OrderStartedDomainEvent**).
3. Manejar eventos de dominio (dentro del proceso actual) que ejecutarán una cantidad abierta de efectos secundarios en múltiples agregados o acciones de aplicación. Por ejemplo:
   - Verificar o crear comprador y método de pago.
   - Crear y enviar un evento de integración relacionado al bus de eventos, para propagar estados a través de microservicios o desencadenar acciones externas, como enviar un correo electrónico al comprador.
   - Manejar otros efectos secundarios.

Como se muestra en la figura 7-15, a partir del mismo evento de dominio, puede manejar múltiples acciones relacionadas con otros agregados en el dominio o acciones de otras aplicaciones que

necesita realizar, a través de microservicios que se conectan con eventos de integración y el bus de eventos.

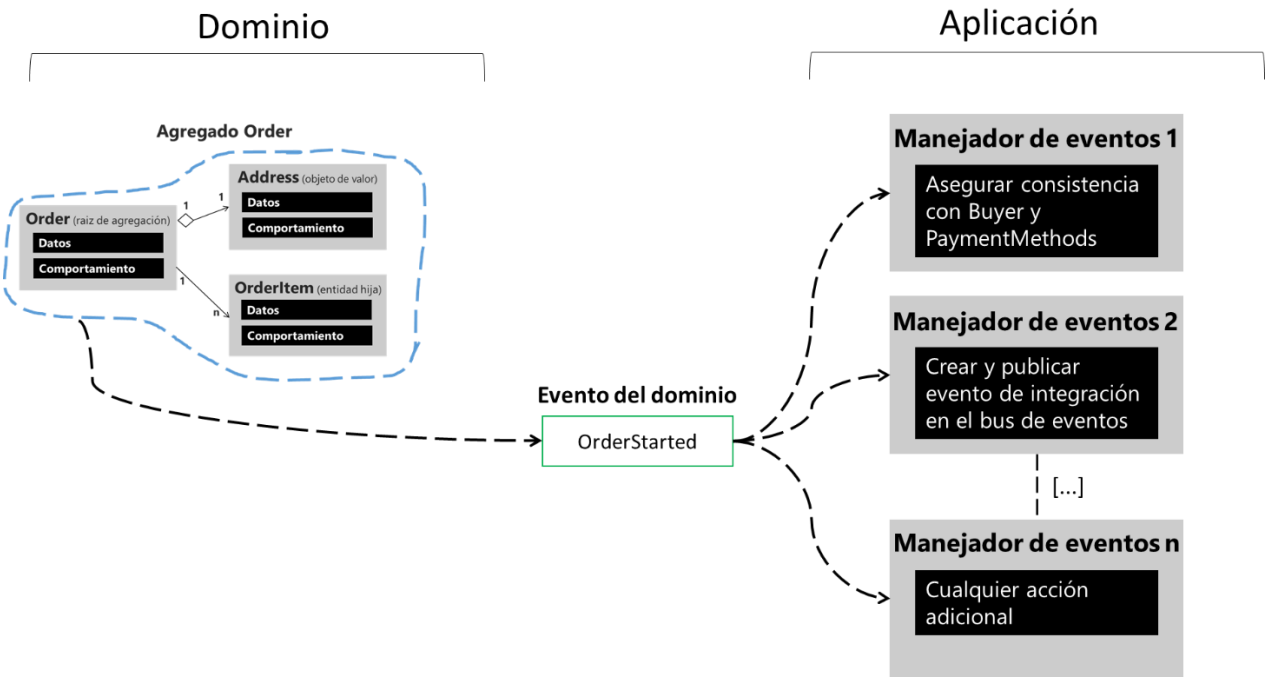


***Figura 7-15**. Manejando múltiples acciones por dominio*

Los manejadores de eventos están normalmente en la capa de aplicación, porque usará objetos de infraestructura como repositorios o una API de aplicación para el comportamiento del microservicio. En ese sentido, los manejadores de eventos son similares a los manejadores de comandos, por lo que ambos son parte de la capa de aplicación. La diferencia importante es que un comando se debe procesar sólo una vez. Un evento de dominio podría procesarse cero o muchas veces, porque podrían recibirlo múltiples receptores o manejadores de eventos con un propósito diferente cada uno.

La posibilidad de una cantidad abierta de manejadores por evento de dominio le permite agregar muchas más reglas de dominio sin afectar su código actual. Por ejemplo, la implementación de la siguiente regla comercial que debe ocurrir inmediatamente después de un evento puede ser tan fácil como agregar algunos manejadores de eventos (o, a lo mejor, sólo uno):

> *Cuando la cantidad total comprada por un cliente en la tienda, en cualquier cantidad de pedidos, exceda los $ 6,000, aplique un descuento del 10% en cada nuevo pedido y notifique al cliente con un correo electrónico sobre ese descuento para futuros pedidos.*

Implementando eventos de dominio

En C#, un evento de dominio es simplemente una estructura o clase que contiene datos, como un DTO, con toda la información relacionada con lo que acaba de ocurrir en el dominio, como se muestra en el siguiente ejemplo:

```
public class OrderStartedDomainEvent : INotification
{
    public string UserId { get; private set; }
    public int CardTypeId { get; private set; }
    public string CardNumber { get; private set; }
    public string CardSecurityNumber { get; private set; }
    public string CardHolderName { get; private set; }
    public DateTime CardExpiration { get; private set; }
    public Order Order { get; private set; }

    public OrderStartedDomainEvent(Order order,
                                   int cardTypeId, string cardNumber,
                                   string cardSecurityNumber, string cardHolderName,
                                   DateTime cardExpiration)
    {
        Order = order;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}
```

Esta es esencialmente una clase que contiene todos los datos relacionados con el evento **`OrderStarted`**.

En términos del lenguaje ubicuo del dominio, dado que un evento es algo que sucedió en el pasado, el nombre de clase del evento debería representarse como un verbo en tiempo pasado, como **`OrderStartedDomainEvent`** u **`OrderShippedDomainEvent`**. Así es como se implementa el evento de dominio en el microservicio de pedido en eShopOnContainers.

Como hemos señalado, una característica importante de los eventos es que, dado que un evento es algo que sucedió en el pasado, no debería cambiar. Por lo tanto, debe ser una clase inmutable. Puede ver en el código anterior que las propiedades son de sólo lectura desde fuera del objeto. La única forma de actualizar el objeto es mediante el constructor cuando se crea el objeto de evento.

Disparando eventos de dominio

La siguiente pregunta es, cómo disparar un evento de dominio para que llegue a los manejadores de eventos relacionados. Puede usar múltiples enfoques.

Udi Dahan originalmente propuso (por ejemplo, en varias publicaciones relacionadas, como Domain Events – Take 2) el uso de una clase estática para administrar y disparar los eventos. Esto podría incluir una clase estática llamada **`DomainEvents`** que dispararía eventos de dominio inmediatamente cuando se invoca, usando sintaxis como **`DomainEvents.Raise(Event myEvent)`**. Jimmy Bogard escribió una

publicación en su blog (Strengthening your domain: Domain Events) que recomienda un enfoque similar.

Sin embargo, cuando la clase de eventos de dominio es estática, también se distribuye a los manejadores de inmediato. Esto hace que las pruebas y la depuración sean más difíciles, ya que los controladores de eventos con lógica de efectos secundarios se ejecutan inmediatamente después de que se genera el evento. Cuando está probando y depurando, realmente quiere enfocarse sólo en lo que está ocurriendo en las clases del agregado actual, no desea ser redireccionado repentinamente a otros manejadores de eventos para realizar efectos secundarios relacionados con otros agregados o lógica de aplicación. Por esto han evolucionados otros enfoques, como se explica en la sección siguiente.

El enfoque diferido para disparar y enviar eventos

En lugar de enviar inmediatamente eventos de dominio a un manejador, un mejor enfoque es agregarlos a una colección y luego enviar esos eventos *justo antes o justo después* de la transacción (como con **SaveChanges** en EF). Este enfoque fue descrito por Jimmy Bogard en este artículo A better domain events pattern (Un patrón mejorado para eventos de dominio).

Es importante decidir si envía los eventos de dominio justo antes o después de realizar la transacción, ya que determina si incluirá los efectos secundarios como parte de la misma transacción o en transacciones diferentes. En el último caso, debe manejar la consistencia eventual entre múltiples agregados. Este tema se trata en la siguiente sección.

El enfoque diferido es lo que usa eShopOnContainers. Primero, agrega los eventos que ocurren en sus entidades en una colección o lista de eventos por entidad. Esa lista debe ser parte del objeto de la entidad, o mejor aún, parte de su clase de entidad base, como se muestra en el siguiente ejemplo de la clase base **Entity**:

```
public abstract class Entity
{
     //...
     private List<INotification> _domainEvents;
     public List<INotification> DomainEvents => _domainEvents;

     public void AddDomainEvent(INotification eventItem)
     {
         _domainEvents = _domainEvents ?? new List<INotification>();
         _domainEvents.Add(eventItem);
     }

     public void RemoveDomainEvent(INotification eventItem)
     {
         if (_domainEvents is null) return;
         _domainEvents.Remove(eventItem);
     }
     //... Additional code
}
```

Cuando desee disparar un evento, simplemente agréguelo a la colección de eventos del código en cualquier método de la entidad de raíz del agregado, como se muestra en el siguiente código, que es parte de la raíz de agregación Order en eShopOnContainers:

```
var orderStartedDomainEvent = new OrderStartedDomainEvent(this, //Order object
                                                           cardTypeId, cardNumber,
                                                           cardSecurityNumber,
                                                           cardHolderName,
                                                           cardExpiration);
this.AddDomainEvent(orderStartedDomainEvent);
```

Tenga en cuenta que lo único que está haciendo el método **AddDomainEvent** es agregar un evento a la lista. Aún no se envía ningún evento y aún no se invoca ningún manejador de eventos.

De hecho, los eventos se van a enviar más tarde, cuando guarde la transacción en la base de datos. Si está utilizando Entity Framework Core, eso significa en el método **SaveChanges** de su **DbContext**, como en el siguiente código:

```
// EF Core DbContext
public class OrderingContext : DbContext, IUnitOfWork
{
  // ...
  public async Task<bool> SaveEntitiesAsync(CancellationToken cancellationToken =
                                                          default(CancellationToken))
  {
    // Dispatch Domain Events collection.
    // Choices:
    // A) Right BEFORE committing data (EF SaveChanges) into the DB. This makes
    //    a single  transaction including side effects from the domain event
    //    handlers that are using the same DbContext with Scope lifetime
    // B) Right AFTER committing data (EF SaveChanges) into the DB. This makes
    //    multiple transactions. You will need to handle eventual consistency and
    //    compensatory actions in case of failures.
     await _mediator.DispatchDomainEventsAsync(this);

     // After this line runs, all the changes (from the Command Handler and Domain
     // event handlers) performed through the DbContext will be commited
     var result = await base.SaveChangesAsync();
  }
  // ...
}
```

Con este código, se envían los eventos de la entidad a sus respectivos manejadores. El resultado general es que ha desacoplado la creación de un evento de dominio (un simple agregado en una lista en la memoria) para que no sea enviado a un manejador de eventos. Además, dependiendo del tipo de emisor que esté utilizando, puede distribuir los eventos de forma síncrona o asíncrona.

Tenga en cuenta que los límites transaccionales tienen un rol significativo aquí. Si su unidad de trabajo y transacción puede abarcar más de un agregado (como cuando se usa EF Core y una base de datos relacional), esto puede funcionar bien. Pero si la transacción no puede abarcar agregados, como cuando está utilizando una base de datos NoSQL como Azure DocumentDB, debe implementar pasos adicionales para lograr consistencia. Esta es otra razón por la cual la ignorancia de la persistencia no es universal; depende del sistema de almacenamiento que use.

Transacciones únicas entre agregados versus consistencia eventual entre agregados

La cuestión de si realizar una sola transacción o confiar en la consistencia eventual entre los agregados, es controvertida. Muchos autores de DDD como Eric Evans y Vaughn Vernon defienden la regla de que una transacción = un agregado y, por lo tanto, argumentan a favor de la consistencia eventual entre los agregados. Por ejemplo, en su libro *Domain-Driven Design*, Eric Evans dice esto:

> *No se espera que las reglas que abarcan agregados estén actualizadas en todo momento. A través del procesamiento de eventos, el procesamiento por lotes u otros mecanismos de actualización, otras dependencias se pueden resolver dentro de un tiempo específico. (pg. 128)*

Vaughn Vernon dice lo siguiente en Effective Aggregate Design. Part II: Making Aggregates Work Together (Diseño efectivo de agregados, parte II, haciendo que los agregados trabajen juntos):

> *Por lo tanto, si la ejecución de un comando en un agregado requiere que se ejecuten reglas del negocio adicionales en uno o más agregados, use consistencia eventual [...] Hay una forma práctica de soportar la consistencia eventual en un modelo DDD. Un método de un agregado publica un evento de dominio que es entregado a tiempo a uno o más suscriptores asíncronos.*

Este razonamiento se basa en manejar transacciones pequeñas, detalladas, en lugar de transacciones que abarcan muchos agregados o entidades. La idea es que, en el segundo caso, la cantidad de bloqueos de base de datos puede ser significativo en aplicaciones de gran escala con altas necesidades de escalabilidad. Aceptar el hecho de que las aplicaciones de alta escalabilidad no necesitan tener una consistencia transaccional instantánea entre múltiples agregados, ayuda a aceptar el concepto de consistencia eventual. Los cambios atómicos a menudo no son necesarios para la empresa y, en cualquier caso, es responsabilidad de los expertos del dominio decir si determinadas operaciones necesitan transacciones atómicas o no. Si una operación siempre necesita una transacción atómica entre múltiples agregados, debería preguntarse si su agregado debería ser más grande o si no fue diseñado correctamente.

Sin embargo, otros desarrolladores y arquitectos como Jimmy Bogard están de acuerdo en abarcar una sola transacción en varios agregados, pero sólo cuando esos agregados adicionales están relacionados con los efectos secundarios del mismo comando original. Por ejemplo, en un mejor patrón de eventos de dominio, Bogard dice esto:

> *Normalmente, deseo que los efectos secundarios de un evento de dominio ocurran dentro de la misma transacción lógica, pero no necesariamente en el mismo ámbito de disparar el evento de dominio [...] justo antes de guardar nuestra transacción, enviamos nuestros eventos a sus respectivos manejadores.*

Si envía los eventos de dominio justo antes de guardar la transacción original, es porque desea que los efectos secundarios de esos eventos se incluyan en la misma transacción. Por ejemplo, si el método **`SaveChanges`** del **`DbContext`** falla, la transacción revertirá todos los cambios, incluido el resultado de cualquier operación de efecto secundario implementada por los manejadores de eventos de dominio relacionados. Esto se debe a que la vida de DbContext está definido por defecto como "*scoped* (en el ámbito)". Por lo tanto, el objeto DbContext se comparte entre todos los repositorios que se instancien dentro del mismo ámbito o grafo del objeto. Esto coincide con el ámbito del **`HttpRequest`** al desarrollar aplicaciones API Web o MVC.

En realidad, ambos enfoques (transacción atómica única y consistencia eventual) pueden ser correctos. Realmente depende de los requisitos de su dominio o negocio y de lo que dicen los expertos del dominio. También depende de qué tan escalable necesite que sea el servicio (las transacciones más granulares tienen menos impacto con respecto a los bloqueos de base de datos). Y depende de qué tanto quiera invertir en su código, ya que la consistencia eventual tiene más complejidad para detectar posibles inconsistencias entre los agregados y la necesidad de implementar acciones compensatorias. Tenga en cuenta que, si realiza cambios en el agregado original y luego, cuando se envían los eventos, hay un problema y los manejadores no pueden guardar sus efectos secundarios, tendrá inconsistencias entre los agregados.

Una forma de permitir acciones compensatorias sería almacenar los eventos de dominio en tablas de base de datos adicionales, para que puedan ser parte de la transacción original. Después, puede tener un proceso por lotes que detecta inconsistencias y ejecuta acciones compensatorias al comparar la lista de eventos con el estado actual de los agregados. Las acciones compensatorias son parte de un tema complejo que requerirá un análisis profundo de su parte, lo que incluye debatirlo con el usuario del negocio y los expertos en el dominio.

En cualquier caso, puede elegir el enfoque que necesite. Pero el enfoque diferido propuesto (disparar los eventos antes de guardar, para usar una sola transacción) es el enfoque más simple cuando se usa EF Core y una base de datos relacional. Es más fácil de implementar y válido en muchos casos del negocio. También es el enfoque utilizado en el microservicio de pedidos en eShopOnContainers.

¿Pero cómo enviar esos eventos a sus respectivos manejadores de eventos? ¿Cuál es el objeto **`_mediator`** mostrado en el ejemplo anterior? Eso tiene que ver con las técnicas y artefactos que puede usar para mapear entre los eventos y sus manejadores de eventos.

El *dispatcher* de eventos de dominio: mapeando eventos a manejadores

Una vez que pueda enviar o publicar los eventos, necesita algún tipo de artefacto que publique el evento para que cada manejador relacionado pueda obtenerlo y procesar los efectos secundarios en función de ese evento.

Un enfoque es un sistema de mensajería real o incluso un bus de eventos, posiblemente basado en un bus de servicio en lugar de eventos en memoria. Sin embargo, para el primer caso, la mensajería real sería excesiva para procesar eventos de dominio, ya que sólo necesita procesar esos eventos dentro del mismo proceso (es decir, dentro del mismo dominio y la misma capa de aplicación).

Otra forma de asignar eventos a varios manejadores es mediante el uso de registros de tipos en un contenedor IoC, para que pueda inferir dinámicamente a dónde enviar los eventos. En otras palabras, necesita saber qué manejadores de eventos necesitan recibir un evento específico. La figura 7-16 muestra un enfoque simplificado para eso.

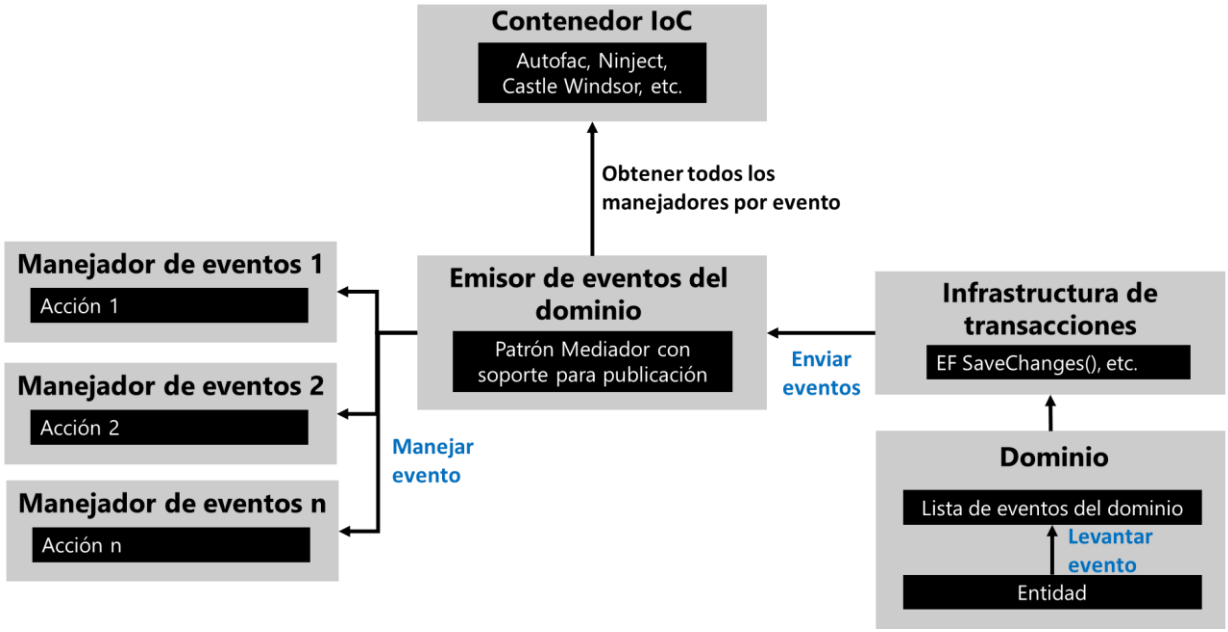


Figura 7-16. *Canalizador de eventos del dominio usando IoC*

Puede construir toda la infraestructura necesaria para implementar ese enfoque usted mismo. Sin embargo, también puede usar librerías disponibles como MediatR, que por debajo usa su contenedor IoC. Por lo tanto, puede usar directamente las interfaces predefinidas y los métodos de publicación/envío del objeto **`_mediator`**.

En este código, primero necesita registrar los tipos de manejadores de eventos en su contenedor IoC, como se muestra en el siguiente ejemplo en el microservicio de pedidos (Ordering) en eShopOnContainers:

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        // Other registrations ...
        // Register the DomainEventHandler classes (they implement
        // IAsyncNotificationHandler<>) in assembly holding the Domain Events

    builder.RegisterAssemblyTypes(
                typeof(ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler)
                            .GetTypeInfo().Assembly)
                              .AsClosedTypesOf(typeof(IAsyncNotificationHandler<>));

        // Other registrations ...
        //...
    }

}
```

El código identifica primero el ensamblado que contiene los manejadores de eventos de dominio al ubicar el ensamblado que contiene cualquiera de los manejadores (usando **`typeof(ValidateOrAddBuyerAggregateWhenXxxx)`**, pero podría haber elegido cualquier otro

manejador de eventos para localizar el ensamblado). Como todos los manejadores de eventos implementan la interfaz **IAysncNotificationHandler**, el código simplemente busca esos tipos y registra todos los manejadores de eventos.

Cómo suscribirse a eventos de dominio

Cuando utiliza MediatR, cada manejador de eventos debe usar un tipo de evento que se indica en el parámetro genérico de la interfaz **IAsyncNotificationHandler**, como puede ver en el siguiente código:

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
                        : IAsyncNotificationHandler<OrderStartedDomainEvent>
```

En función de la relación entre el evento y el manejador de eventos, que se puede considerar como la suscripción, el artefacto MediatR puede descubrir todos los manejadores de eventos para cada evento y activar cada uno de ellos.

Cómo manejar eventos de dominio

Finalmente, el manejador de eventos usualmente implementa código de la capa de aplicación, que usa repositorios de infraestructura para obtener los agregados adicionales requeridos, para ejecutar la lógica de dominio de los efectos secundarios. El siguiente código de manejador de eventos de dominio en eShopOnContainers, muestra un ejemplo de implementación.

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
                    : INotificationHandler<OrderStartedDomainEvent>
{
   private readonly ILoggerFactory _logger;
   private readonly IBuyerRepository<Buyer> _buyerRepository;
   private readonly IIdentityService _identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
                    ILoggerFactory logger,
                    IBuyerRepository<Buyer> buyerRepository,
                    IIdentityService identityService)
    {
        // ...Parameter validations...
    }

    public async Task Handle(OrderStartedDomainEvent orderStartedEvent)
    {
        var cardTypeId = (orderStartedEvent.CardTypeId != 0) ?
                         orderStartedEvent.CardTypeId : 1;

        var userGuid = _identityService.GetUserIdentity();

        var buyer = await _buyerRepository.FindAsync(userGuid);
        bool buyerOriginallyExisted = (buyer == null) ? false : true;

        if (!buyerOriginallyExisted)
        {
            buyer = new Buyer(userGuid);
        }

        buyer.VerifyOrAddPaymentMethod(cardTypeId,
                                       $"Payment Method on {DateTime.UtcNow}",
                                       orderStartedEvent.CardNumber,
                                       orderStartedEvent.CardSecurityNumber,
                                       orderStartedEvent.CardHolderName,
                                       orderStartedEvent.CardExpiration,
                                       orderStartedEvent.Order.Id);

        var buyerUpdated = buyerOriginallyExisted ? _buyerRepository.Update(buyer) :
                                                    _buyerRepository.Add(buyer);

        await _buyerRepository.UnitOfWork.SaveEntitiesAsync();

       // Logging code using buyerUpdated info, etc.
    }
}
```

El código de manejador de eventos anterior se considera código de la capa de aplicación, porque usa repositorios de infraestructura, como se explica en la siguiente sección de la capa de persistencia. Los manejadores de eventos también podrían usar otros componentes de infraestructura.

Los eventos de dominio pueden generar eventos de integración que se publiquen fuera del microservicio

Finalmente, es importante mencionar que a veces es posible que desee propagar eventos entre múltiples microservicios. Esto se considera un evento de integración y podría publicarse a través de un bus de eventos desde cualquier manejador de eventos.

Conclusiones sobre los eventos de dominio

Como se dijo, use eventos de dominio para implementar explícitamente los efectos secundarios de los cambios dentro de su dominio. Para usar la terminología DDD, use eventos de dominio para implementar explícitamente efectos secundarios en uno o varios agregados. Además, para una mejor escalabilidad y un menor impacto en los bloqueos de bases de datos, use la consistencia eventual entre los agregados dentro del mismo dominio.

Recursos adicionales

- **Greg Young. What is a Domain Event?**
  http://codebetter.com/gregyoung/2010/04/11/what-is-a-domain-event/
- **Jan Stenberg. Domain Events and Eventual Consistency**
  https://www.infoq.com/news/2015/09/domain-events-consistency
- **Jimmy Bogard. A better domain events pattern**
  https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/
- **Vaughn Vernon. Effective Aggregate Design Part II: Making Aggregates Work Together**
  http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Jimmy Bogard. Strengthening your domain: Domain Events**
  https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/
- **Tony Truong. Domain Events Pattern Example**
  http://www.tonytruong.net/domain-events-pattern-example/
- **Udi Dahan. How to create fully encapsulated Domain Models**
  http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/
- **Udi Dahan. Domain Events – Take 2**
  http://udidahan.com/2008/08/25/domain-events-take-2/
- **Udi Dahan. Domain Events – Salvation**
  http://udidahan.com/2009/06/14/domain-events-salvation/
- **Jan Kronquist. Don't publish Domain Events, return them!**
  https://blog.jayway.com/2013/06/20/dont-publish-domain-events-return-them/
- **Cesar de la Torre. Domain Events vs. Integration Events in DDD and microservices architectures**
  https://blogs.msdn.microsoft.com/cesardelatorre/2017/02/07/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/

Diseñando la capa de infraestructura de persistencia

Los componentes de persistencia de datos proporcionan acceso a los datos alojados dentro de los límites de un microservicio (es decir, una base de datos del microservicio). Contienen la implementación real de componentes como las clases de repositorios y Unidades de Trabajo, como EF DBContexts personalizados.

El patrón de Repositorio

Los repositorios son clases o componentes que encapsulan la lógica necesaria para acceder a las fuentes de datos. Centralizan la funcionalidad común de acceso a datos, facilitando el mantenimiento y desacoplando la infraestructura o tecnología utilizada para acceder a las bases de datos desde las capas del modelo de dominio y de la aplicación. Si usa un ORM como Entity Framework, se simplifica el código que debe implementar, gracias a LINQ y el tipado fuerte. Esto le permite enfocarse en la lógica de persistencia de datos en lugar de en la fontanería del acceso a datos.

El patrón Repositorio es una forma de trabajar con una fuente de datos, que está bien documentada. En el libro Patterns of Enterprise Application Architecture, Martin Fowler describe un repositorio de la siguiente manera:

> *Un repositorio realiza las tareas de un intermediario entre las capas de modelo de dominio y el mapeo de datos, actuando de forma similar a un conjunto de objetos de dominio en la memoria. Los objetos cliente pueden construir consultas de forma declarativa y luego enviarlas a los repositorios para obtener los resultados. Conceptualmente, un repositorio encapsula un conjunto de objetos almacenados en la base de datos y las operaciones que se pueden realizar sobre ellos, proporcionando una forma más cercana a la capa de persistencia. Los repositorios también soportan el propósito de separar, claramente y en una dirección, la dependencia entre el dominio de trabajo y el mapeo de datos.*

Defina un repositorio por agregado

Debe crear un repositorio para cada agregado o raíz de agregación. En un microservicio basado en patrones DDD, el único canal que debe usar para actualizar la base de datos deben ser los repositorios. Esto se debe a que tienen una relación de uno a uno con la raíz de agregación, que controla las invariantes del agregado y la consistencia transaccional. Está bien consultar la base de datos a través de otros canales (como puede hacer siguiendo un enfoque CQRS), porque las consultas no cambian el estado de la base de datos. Sin embargo, el área transaccional (las actualizaciones) siempre debe estar controlada por los repositorios y las raíces de agregación.

Básicamente, un repositorio le permite llenar datos en la memoria, en la forma de entidades del dominio, que provienen de la base de datos. Una vez que las entidades están en la memoria, se pueden cambiar y luego volver a la base de datos mediante transacciones.

Como se señaló anteriormente, si está utilizando el patrón arquitectónico CQS/CQRS, las consultas iniciales se realizarán a través consultas secundarias fuera del modelo de dominio, realizadas mediante sentencias SQL simples utilizando Dapper. Este enfoque es mucho más flexible que los repositorios, porque puede consultar y hacer *join* con cualquier tabla que necesite y estas consultas no están restringidas por las reglas de los agregados. Esa información irá a la capa de presentación o aplicación cliente.

Si el usuario realiza cambios, los datos que se actualizarán vendrán desde la aplicación cliente o la capa de presentación hasta la capa de la aplicación (como un servicio API Web). Cuando recibe un comando (con datos) en un manejador de comandos, se usan repositorios para obtener los datos que desea actualizar desde la base de datos. Usted lo actualiza en la memoria con la información pasada con los comandos y luego agrega o actualiza los datos (en las entidades de dominio) hacia la base de datos a través de una transacción.

Debemos enfatizar nuevamente que sólo se debe definir un repositorio para cada raíz de agregación, como se muestra en la Figura 7-17. Para lograr el objetivo de que la raíz de agregación debe mantener la consistencia transaccional entre todos los objetos dentro del agregado, nunca debe crear un repositorio para cada tabla en la base de datos.

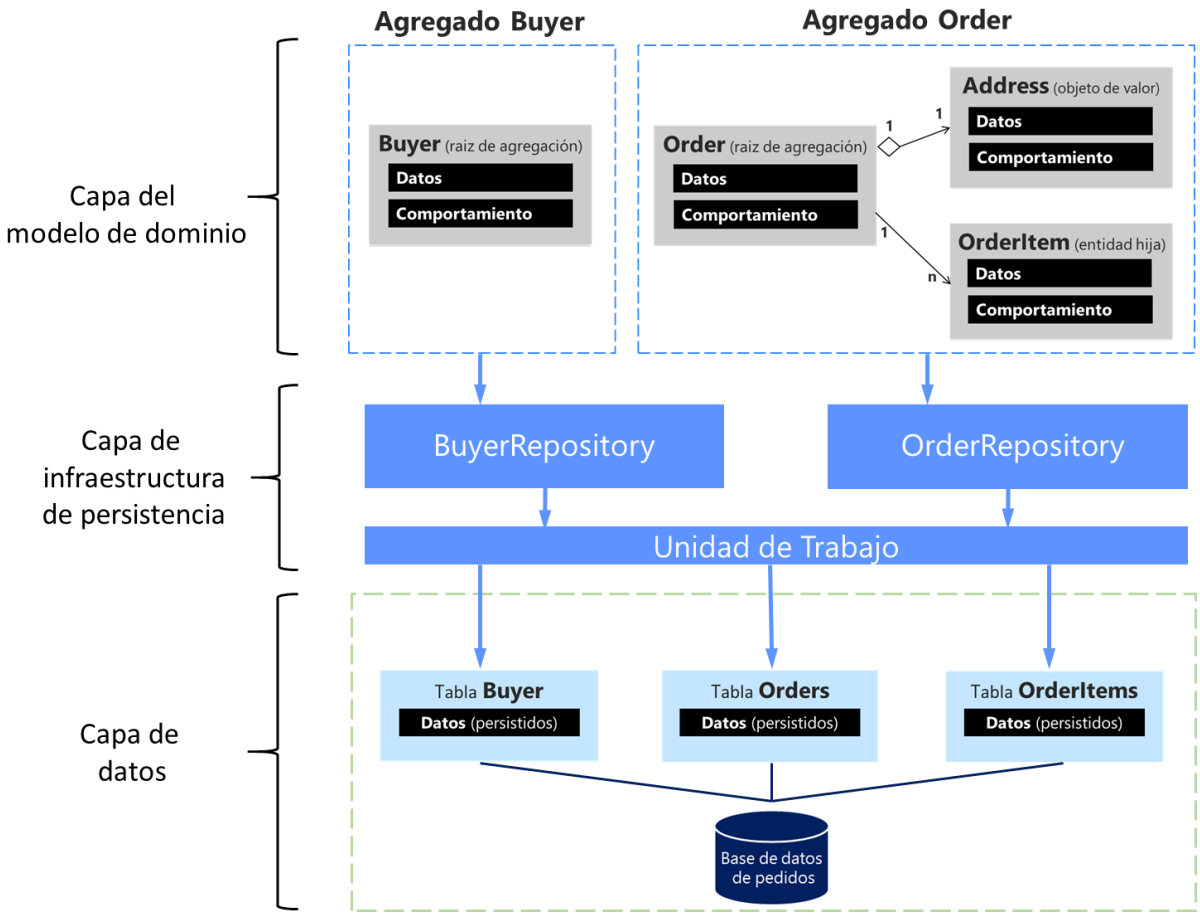


***Figura 7-17**. La relación entre repositorios, agregados y tablas de la base de datos*

Reforzando la regla de una raíz de agregación por repositorio

Puede aportar valor el implementar su diseño de repositorio para asegurar que se cumpla la regla de que sólo las raíces de agregación deben tener repositorios. Puede crear un tipo de depósito genérico o base que restrinja el tipo de entidades con las que trabaja para garantizar que tengan la interfaz del marcador **`IAggregateRoot`**.

Por lo tanto, cada clase de repositorio en la capa de infraestructura implementa su propio contrato o interfaz, como se muestra en el siguiente código:

```
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
```

Cada interfaz de repositorio específico implementa la interfaz genérica de **IRepository**:

```
    public interface IOrderRepository : IRepository<Order>
    {
        Order Add(Order order);
        // ...
    }
```

Sin embargo, una mejor forma de hacer que el código haga cumplir la convención de que cada repositorio debe estar relacionado con un agregado único sería implementar un tipo de repositorio genérico, por lo que es explícito que está utilizando un repositorio para apuntar a un agregado específico. Eso se puede hacer fácilmente implementando ese genérico en la interfaz base de **IRepository**, como en el siguiente código:

```
    public interface IRepository<T> where T : IAggregateRoot
```

El patrón de repositorio simplifica las pruebas de la lógica de la aplicación

El patrón Repositorio le permite probar fácilmente su aplicación con pruebas unitarias. Recuerde que las pruebas unitarias sólo prueban su código, no la infraestructura, por lo que las abstracciones del repositorio hacen que sea más fácil lograr ese objetivo.

Como se señaló en una sección anterior, se recomienda definir y colocar las interfaces del repositorio en la capa del modelo de dominio para que la capa de la aplicación (por ejemplo, su servicio de API Web) no dependa directamente de la capa de infraestructura donde haya implementado los repositorios. Al hacer esto y usar Inyección de Dependencias en los controladores de su API Web, puede implementar repositorios simulados que devuelvan datos falsos en lugar de datos de la base de datos. Ese enfoque desacoplado le permite crear y ejecutar pruebas unitarias para verificar sólo la lógica de la aplicación, sin requerir conectividad con la base de datos.

Las conexiones a las bases de datos pueden fallar y, lo que es más importante, ejecutar cientos de pruebas en una base de datos es malo por dos razones. En primer lugar, puede llevar mucho tiempo debido a la gran cantidad de pruebas. En segundo lugar, los registros de la base de datos pueden afectar los resultados de sus pruebas, de modo que no sean consistentes. Las pruebas contra la base de datos no son pruebas unitarias, sino pruebas de integración. Debería ejecutar muchas pruebas unitarias rápidamente, pero menor cantidad de pruebas de integración contra las bases de datos.

En términos de separación de intereses para las pruebas unitarias, su lógica opera en entidades de dominio en la memoria. Se supone que el repositorio las ha entregado. Una vez que su lógica modifica las entidades de dominio, se supone que el repositorio las almacenará correctamente. El punto importante aquí es crear pruebas unitarias contra su modelo y su lógica de dominio. Las raíces de agregación son los límites principales de consistencia en DDD.

La diferencia entre el patrón de repositorio y el viejo patrón de las clases de acceso a datos (clases DAL)

Un objeto de acceso a datos realiza directamente el acceso a los datos y las operaciones de persistencia contra el almacenamiento. Un repositorio marca los datos con las operaciones que desea realizar en la memoria de una unidad de trabajo (como al usar el DbContext en EF), pero estas actualizaciones no se realizarán de inmediato.

Una unidad de trabajo se refiere a una transacción única que incluye múltiples operaciones de inserción, actualización o eliminación. En términos simples, significa que para una acción específica del usuario (por ejemplo, el registro en un sitio web), todas las transacciones de inserción, actualización y eliminación se manejan en una sola transacción. Esto es más eficiente que manejar múltiples transacciones de bases de datos.

Estas operaciones de persistencia múltiple se realizarán más adelante en una sola acción, cuando el código de la capa de aplicación lo ordene. La decisión sobre la aplicación de los cambios en memoria en el almacenamiento real de la base de datos generalmente se basa en el patrón de Unidad de Trabajo. En EF, el patrón de unidad de trabajo se implementa como el DBContext.

En muchos casos, este patrón o forma de aplicar operaciones contra el almacenamiento puede aumentar el rendimiento de la aplicación y reducir la posibilidad de inconsistencias. Además, reduce el bloqueo de transacciones en las tablas de la base de datos, porque todas las operaciones previstas se guardan como parte de una transacción. Esto es más eficiente en comparación con la ejecución de muchas operaciones aisladas en la base de datos. Por lo tanto, el ORM seleccionado podrá optimizar la ejecución contra la base de datos agrupando varias acciones de actualización dentro de la misma transacción, a diferencia realizar muchas de transacciones pequeñas y separadas.

Los repositorios no son obligatorios

Los repositorios personalizados son útiles por las razones citadas anteriormente y ese es el enfoque para el microservicio de pedido en eShopOnContainers. Sin embargo, no es un patrón esencial para implementar en un diseño DDD o incluso en desarrollo general en .NET.

Por ejemplo, Jimmy Bogard, al hacer comentarios directos para esta guía, dijo lo siguiente:

> *Esta será probablemente mi opinión más larga. Realmente no soy fan de los repositorios, principalmente porque ocultan los detalles importantes del mecanismo de persistencia subyacente. Es por eso que también voy por la vía de comandos con MediatR. Puedo usar todo el poder de la capa de persistencia e insertar todo ese comportamiento de dominio en mis raíces de agregación. Normalmente no quiero usar maquetas de mis repositorios, aún necesito tener esa prueba de integración con la realidad. Ir con CQRS significaba que ya no necesitábamos repositorios.*

Encontramos que los repositorios son útiles, pero reconocemos que no son críticos para su diseño DDD, de la forma como lo son el patrón Agregado y el modelo de dominio expresivo. Por lo tanto, use el patrón Repositorio o no, como le parezca mejor.

Recursos adicionales

El patrón de repositorio

- **Edward Hieatt and Rob Mee. Repository pattern.**
  http://martinfowler.com/eaaCatalog/repository.html

- **The Repository pattern**
  https://msdn.microsoft.com/library/ff649690.aspx
- **Repository Pattern: A data persistence abstraction**
  http://deviq.com/repository-pattern/
- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software.**
  (Book; includes a discussion of the Repository pattern)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/

El patrón Unidad de Trabajo

- **Martin Fowler. Unit of Work pattern.**
  http://martinfowler.com/eaaCatalog/unitOfWork.html
- **Implementar el repositorio y una unidad de patrones de trabajo en una aplicación de MVC de ASP.NET**
  https://docs.microsoft.com/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

El patrón de Especificación

El patrón de Especificación (su nombre completo sería Patrón de especificación de consulta) es un patrón de Diseño Orientado por el Dominio, diseñado como el lugar donde se puede poner la definición de una consulta con lógica de ordenación y paginación opcional. El patrón de especificación define una consulta en un objeto. Por ejemplo, para encapsular una consulta paginada que busque algunos productos, puede crear una especificación **`PagedProduct`** que tome los parámetros de entrada necesarios (pageNumber, pageSize, filter, etc.). Entonces, cualquier método del repositorio (generalmente una sobrecarga de **`List()`**) aceptaría una **`ISpecification`** y ejecutaría la consulta esperada basada en esa especificación.

Hay varios beneficios para este enfoque.

- La especificación tiene un nombre (a diferencia de sólo un montón de expresiones LINQ) sobre la que se puede discutir.
- Se pueden hacer pruebas unitarias de la especificación para garantizar que sea correcta. también se puede reutilizar fácilmente si necesita un comportamiento similar (por ejemplo, en una vista MVC View o una acción API Web, así como en diversos servicios).
- También se puede usar una especificación para describir la forma de los datos que se devolverán, de modo que las consultas puedan devolver sólo los datos que solicitaron. Esto elimina la necesidad de *lazy loading* (ejecutar las consultas SQL en la medida que se van necesitando versus consolidar las consultas para eficiencia y rendimiento) en las aplicaciones web (que generalmente no es una buena idea) y ayuda a evitar que las implementaciones del repositorio se llenen de estos detalles.

Un ejemplo de una interfaz de especificación genérica es el siguiente código de eShopOnWeb.

```
// https://github.com/dotnet-architecture/eShopOnWeb
public interface ISpecification<T>
{
    Expression<Func<T, bool>> Criteria { get; }
    List<Expression<Func<T, object>>> Includes { get; }
    List<string> IncludeStrings { get; }
}
```

En las próximas secciones se explica cómo implementar el patrón de Especificación con Entity Framework Core 2.0 y cómo usarlo desde cualquier clase de Repositorio.

Nota importante: el patrón de especificación es un patrón antiguo que se puede implementar de muchas maneras diferentes, como en los recursos adicionales a continuación. Como patrón/idea, es bueno conocer los enfoques anteriores, pero tenga en cuenta que las implementaciones más antiguas no aprovechan las capacidades modernas del lenguaje como Linq y expresiones.

Recursos adicionales

- **The Specification pattern.**
  http://deviq.com/specification-pattern/
- **Evans, Eric (2004). Domain-Driven Design. Addison-Wesley. p. 224.**
- **Specifications. Martin Fowler**
  https://www.martinfowler.com/apsupp/spec.pdf

Implementando la persistencia en la capa de infraestructura con Entity Framework Core

Cuando utiliza bases de datos relacionales como SQL Server, Oracle o PostgreSQL, un enfoque recomendado es implementar la capa de persistencia basada en Entity Framework (EF). EF soporta LINQ y proporciona objetos fuertemente tipados para su modelo, así como una persistencia simplificada en su base de datos.

Entity Framework tiene una larga historia como parte de .NET Framework. Cuando usa .NET Core, también debería usar Entity Framework Core, que se ejecuta tanto en Windows como en Linux de la misma forma. EF Core se reescribió por completo desde cero, no es una migración del Entity Framework tradicional, se implementó con una huella de memoria mucho más pequeña y mejoras importantes en el rendimiento.

Introducción a Entity Framework Core

Entity Framework (EF) Core es una versión ligera, extensible y multiplataforma de la popular tecnología de acceso a datos de Entity Framework. Fue presentado con .NET Core a mediados de 2016.

Ya que existe una introducción a EF Core disponible en la documentación de Microsoft, aquí simplemente proporcionamos enlaces a esa información.

Recursos adicionales

- **Entity Framework Core**
  https://docs.microsoft.com/ef/core/
- **Introducción a ASP.NET Core MVC y Entity Framework Core con Visual Studio**
  https://docs.microsoft.com/aspnet/core/data/ef-mvc/
- **DbContext Class**
  https://docs.microsoft.com/ef/core/api/microsoft.entityframeworkcore.dbcontext
- **Compare EF Core & EF6.x**
  https://docs.microsoft.com/ef/efcore-and-ef6/index

La infraestructura en Entity Framework Core desde la perspectiva DDD

Desde un punto de vista DDD, una capacidad importante de EF es la capacidad de usar entidades de dominio POCO, también conocidas, en la terminología EF, como entidades POCO *code-first*. Si usa entidades de dominio POCO, las clases de su modelo de dominio son ignorantes de persistencia, por lo que siguen los principios de Ignorancia de la Persistencia e Ignorancia de la Infraestructura.

Según los patrones DDD, debe encapsular el comportamiento y las reglas del dominio dentro de la propia clase de entidad, de modo que pueda controlar invariantes, validaciones y reglas al acceder a cualquier colección. Por lo tanto, no es una buena práctica en DDD permitir el acceso público a colecciones de entidades secundarias o *value objects*. En su lugar, debe exponer métodos que controlan cómo y cuándo se pueden actualizar sus campos y colecciones y qué comportamiento y acciones deberían ocurrir cuando pase eso.

A partir de EF Core 1.1, se pueden tener campos simples en las entidades, en vez de propiedades públicas, para satisfacer esos requisitos de DDD. Si no desea que un campo de una entidad sea

accesible externamente, puede simplemente crear el atributo o campo en lugar de una propiedad. También puede usar *setters* privados.

De forma similar, ahora puede tener acceso de sólo lectura a las colecciones, al usar una propiedad pública tipeada como **IReadOnlyCollection<T>**, que está respaldada por un campo privado para la colección (como una **Lista<T>**) en su entidad, que utiliza EF para la persistencia. Las versiones anteriores de Entity Framework requerían propiedades que soportaran **ICollection<T>**, para manejar colecciones, lo que significaba que cualquier desarrollador que utilizara la clase de la entidad padre podía agregar o eliminar elementos a través de sus colecciones. Esa posibilidad es contraria a los patrones recomendados en DDD.

Ahora puede usar una colección privada mientras expone un objeto **IReadOnlyCollection<T>** de sólo lectura, como se muestra en el siguiente ejemplo de código:

```
public class Order : Entity
{
    // Using private fields, allowed since EF Core 1.1
    private DateTime _orderDate;
    // Other fields ...

    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    protected Order() { }
    public Order(int buyerId, int paymentMethodId, Address address)
    {
          // Initializations ...
    }

    public void AddOrderItem(int productId, string productName,
                             decimal unitPrice, decimal discount,
                             string pictureUrl, int units = 1)
    {
        // Validation logic...

        var orderItem = new OrderItem(productId, productName,
                                      unitPrice, discount,
                                      pictureUrl, units);
        _orderItems.Add(orderItem);
    }
}
```

Tenga en cuenta que sólo se puede acceder a la propiedad **OrderItems** como de sólo lectura utilizando **IReadOnlyCollection<OrderItem>**. Como este tipo es de solo lectura, está protegido contra actualizaciones externas regulares.

EF Core proporciona una forma de mapear el modelo de dominio a la base de datos física sin "contaminar" el modelo de dominio. Es un código .NET POCO puro, porque la acción de mapeo se implementa en la capa de persistencia. En esa acción de mapeo, necesita configurar la asignación de campos a la base de datos. En el siguiente ejemplo de un método **OnModelCreating**, el código resaltado le dice a EF Core que acceda a la propiedad **OrderItems** a través de su campo.

```
// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   // ...
   modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
   // Other entities' configuration ...
   // ...
}


// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
        // Other configuration
        // ...

        var navigation =
              orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));

        //EF access the OrderItem collection property through its backing field
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

        // Other configuration
        // ...
    }
}
```

Cuando utiliza campos en lugar de propiedades, la entidad **OrderItem** se conserva igual que si tuviera una propiedad **List<OrderItem>**. Sin embargo, expone un único acceso, el método **AddOrderItem**, para agregar nuevos elementos al pedido. Como resultado, el comportamiento y los datos se unen y serán consistentes a través de cualquier código de aplicación que use el modelo de dominio.

Implementando repositorios personalizados con Entity Framework Core

A nivel de implementación, un repositorio es simplemente una clase con código de persistencia de datos, coordinada por una unidad de trabajo (DBContext en EF Core) cuando se realizan actualizaciones, como se muestra en la siguiente clase:

```
// using statements...
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository
    {
        private readonly OrderingContext _context;
        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }

        public BuyerRepository(OrderingContext context)
        {
            _context = context ?? throw new ArgumentNullException(nameof(context));
        }

        public Buyer Add(Buyer buyer)
        {
            return _context.Buyers.Add(buyer).Entity;
        }

        public async Task<Buyer> FindAsync(string BuyerIdentityGuid)
        {
            var buyer = await _context.Buyers
                .Include(b => b.Payments)
                .Where(b => b.FullName == BuyerIdentityGuid)
                .SingleOrDefaultAsync();

            return buyer;
        }
    }
}
```

Tenga en cuenta que la interfaz **IBuyerRepository** proviene de la capa del modelo de dominio como un contrato. Sin embargo, la implementación del repositorio se realiza en la capa de persistencia e infraestructura.

El DbContext viene a través del constructor a través de Inyección de Dependencias. Se comparte entre varios repositorios dentro del mismo ámbito de la petición HTTP, gracias a su duración por defecto (**ServiceLifetime.Scoped**) en el contenedor IoC (que también se puede establecer explícitamente con **services.AddDbContext<>**).

Métodos a implementar en un repositorio (actualizaciones o transacciones versus consultas)

Dentro de cada repositorio, debe colocar los métodos de persistencia que actualicen el estado de las entidades contenidas en el agregado. Recuerde que existe una relación uno-a-uno entre un agregado y su repositorio relacionado. Tenga en cuenta que un objeto de la raíz de agregación podría tener entidades hijo dentro de su grafo en EF. Por ejemplo, un comprador puede tener varios métodos de pago como entidades secundarias relacionadas.

Como el enfoque para el microservicio de pedidos en eShopOnContainers también se basa en CQS/CQRS, la mayoría de las consultas no se implementan en repositorios personalizados. Los desarrolladores tienen libertad de crear las consultas, incluyendo *joins*, que necesitan para la capa de presentación, sin las restricciones impuestas por los agregados, los repositorios y DDD en general. La mayoría de los repositorios personalizados sugeridos por esta guía tienen varios métodos de actualización o transaccionales, pero sólo los métodos de consulta necesarios para actualizar los datos. Por ejemplo, el repositorio **`BuyerRepository`** implementa un método **`FindAsync`**, porque la aplicación necesita saber si existe un comprador en particular antes de crear un nuevo comprador relacionado con el pedido.

Sin embargo, los métodos de consulta reales para enviar datos a la capa de presentación o aplicaciones de cliente se implementan, como se mencionó, en las consultas de CQRS basadas en consultas flexibles que usan Dapper.

Usando un repositorio personalizado versus usar el DbContext de EF directamente

La clase **`DbContext`** de Entity Framework se basa en los patrones de Unidad de trabajo y Repositorio, y se puede usar directamente desde su código, por ejemplo, desde un controlador ASP.NET Core MVC. Esa es la forma en que puede crear el código más simple, como en el microservicio del catálogo CRUD en eShopOnContainers. En los casos en que desee el código más simple posible, es posible que desee utilizar directamente la clase DbContext, como lo hacen muchos desarrolladores.

Sin embargo, la implementación de repositorios personalizados proporciona varios beneficios al implementar microservicios o aplicaciones más complejos. Los patrones de Unidad de trabajo y Repositorio están destinados a encapsular la capa de persistencia de la infraestructura, por lo que está desacoplada de las capas del modelo de aplicación y dominio. La implementación de estos patrones puede facilitar el uso de maquetas de repositorios para las pruebas unitarias del sistema.

En la Figura 7-18 puede ver las diferencias entre no usar repositorios (usar directamente el DbContext de EF) versus usar repositorios que hacen que sea más fácil crear maquetas de esos repositorios.

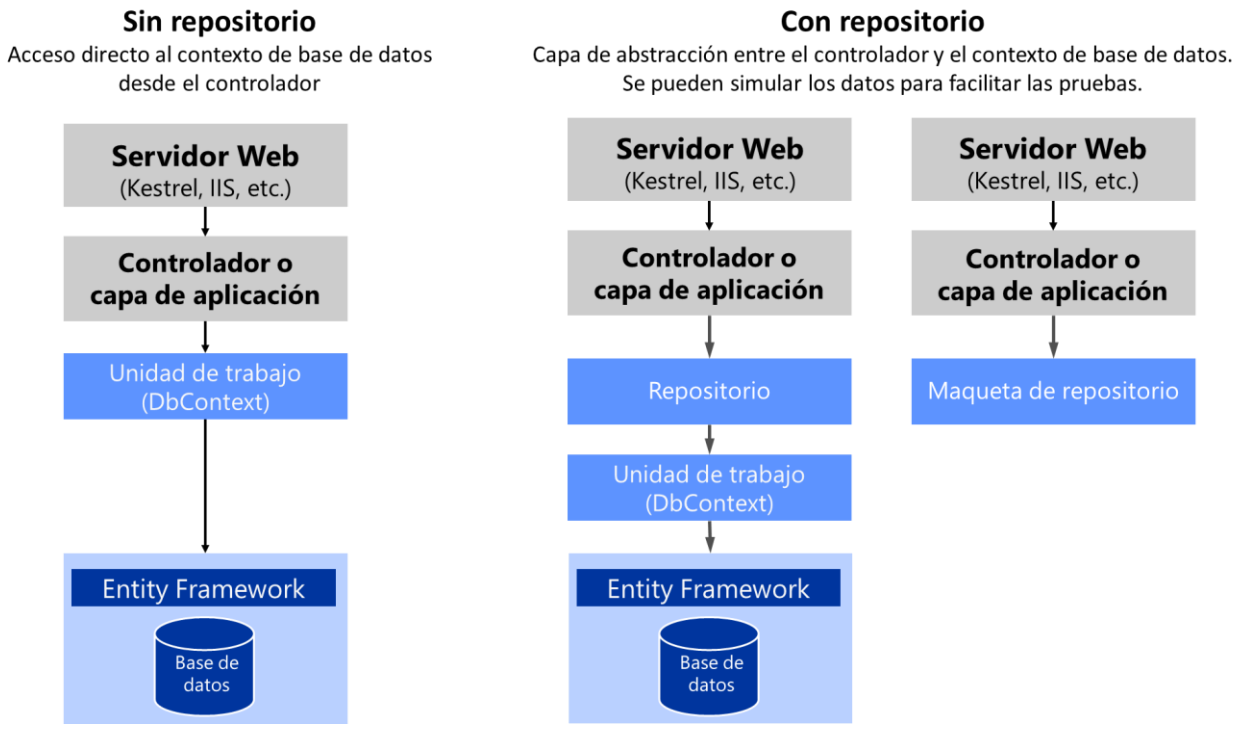


***Figura 7-18**. Usando repositorios personalizados versus un DbContext sencillo*

Hay múltiples alternativas para hacer sustitutos para las pruebas. Se podrían hacer los repositorios o de toda una unidad de trabajo. Por lo general es suficiente hacerlas de los repositorios, ya que la complejidad para abstraer y simular una unidad de trabajo completa generalmente no es necesaria.

Más adelante, cuando nos enfoquemos en la capa de aplicación, veremos cómo funciona la Inyección de Dependencias en ASP.NET Core y cómo se implementa al usar repositorios.

En resumen, usar repositorios personalizados le facilita realizar pruebas unitarias, que no se ven afectadas por el estado de la capa de datos. Si ejecutara pruebas que también accedan a la base de datos real a través de Entity Framework, entonces no serían pruebas unitarias sino pruebas de integración, que son mucho más lentas.

Si estuviese utilizando el DbContext directamente tendría que manejar un substituto o ejecutar pruebas mediante un servidor SQL en memoria, con datos predecibles para las pruebas. Pero manejar los substitutos del DbContext o controlar los datos falsos para las pruebas, requiere más trabajo que manejar sustitutos a nivel del repositorio. Por supuesto, siempre se pueden probar los controladores MVC.

El ciclo de vida del DbContext y la IUnitOfWork en el contenedor de IoC

El objeto **`DbContext`** (expuesto como un objeto **`IUnitOfWork`**) debería compartirse entre varios repositorios dentro del mismo ámbito de la petición HTTP. Por ejemplo, cuando la operación que se está ejecutando debe manejar múltiples agregados o simplemente porque está utilizando múltiples instancias de repositorio. También es importante mencionar que la interfaz **`IUnitOfWork`** forma parte de su capa de dominio, no es un tipo EF Core.

Para hacer eso, la instancia del objeto DbContext debe tener la duración del servicio establecida como **ServiceLifetime.Scoped**. Esta es la duración predeterminada cuando se registra un DbContext con **services.AddDbContext** en su contenedor IoC, desde el método **ConfigureServices** del fichero **Startup.cs** en su proyecto ASP.NET Core Web API. El siguiente código ilustra esto.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc(options =>
    {
        options.Filters.Add(typeof(HttpGlobalExceptionFilter));
    }).AddControllersAsServices();
     services.AddEntityFrameworkSqlServer()
            .AddDbContext<OrderingContext>(options =>
            {
                options.UseSqlServer(Configuration["ConnectionString"],
                    sqlOptions => sqlOptions.MigrationsAssembly(typeof(Startup).
                                                        GetTypeInfo().
                                                         Assembly.GetName().Name));
            },
            ServiceLifetime.Scoped  // Note that Scoped is the default choice
                                    // in AddDbContext. It is shown here only for
                                    // pedagogic purposes.
            );
}
```

La duración de las instancias de DbContext no se debe configurar como **ServiceLifetime.Transient** o **ServiceLifetime.Singleton**.

El ciclo de vida de la instancia del repositorio en el contenedor IoC

De manera similar, el ciclo de vida del repositorio generalmente se debe establecer como por ámbito (**`InstancePerLifetimeScope`** en Autofac). También podría ser transitorio (**`InstancePerDependency`** en Autofac), pero su servicio será más eficiente en lo que respecta a la memoria cuando utilice la duración por ámbito.

```
// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
                .As<IOrderRepository>()
                .InstancePerLifetimeScope();
```

Tenga en cuenta que el uso del tiempo de vida de *singleton* para el repositorio podría causarle serios problemas de concurrencia cuando su DbContext se establece por ámbito (**`InstancePerLifetimeScope`**), que es el tiempo de vida por defecto para un **`DBContext`**.

Recursos adicionales

- **Implementar el repositorio y una unidad de patrones de trabajo en una aplicación de MVC de ASP.NET**
  https://docs.microsoft.com/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application
- **Jonathan Allen. Implementation Strategies for the Repository Pattern with Entity Framework, Dapper, and Chain**
  https://www.infoq.com/articles/repository-implementation-strategies
- **Cesar de la Torre. Comparing ASP.NET Core IoC container service lifetimes with Autofac IoC container instance scopes**
  https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Mapeo de tablas

El mapeo de una tabla identifica los datos de la tabla a consultar y guardar en la base de datos. Anteriormente vimos cómo se pueden usar las entidades de dominio (por ejemplo, un dominio de productos o pedidos) para generar un esquema de base de datos relacionado. EF está fuertemente diseñado en torno al concepto de convenciones. Las convenciones abordan preguntas como "¿Cuál será el nombre de una tabla?" O "¿Qué propiedad es la clave principal?" Las convenciones generalmente se basan en nombres frecuentes, por ejemplo, es típico que la clave principal sea una propiedad que termina **`Id`**.

Por convención, cada entidad se configurará para mapearse a una tabla con el mismo nombre que la propiedad **`DbSet<TEntity>`** que expone a la entidad en el contexto derivado. Si no se proporciona un valor **`DbSet<TEntity>`** para la entidad dada, se utiliza el nombre de la clase.

Anotaciones de datos versus *Fluent API*

Existen muchas convenciones adicionales de EF Core y la mayoría de ellas se pueden cambiar mediante el uso de anotaciones de datos o de la *Fluent API*, implementadas dentro del método **`OnModelCreating`**.

Las anotaciones de datos se deben usar en las propias clases del modelo de entidad, lo cual es una forma más intrusiva desde un punto de vista DDD. Esto se debe a que está contaminando su modelo con anotaciones de datos relacionadas con la base de datos de infraestructura. Por otro lado, la *Fluent API* es una forma conveniente de cambiar la mayoría de las convenciones y mapeos dentro de su capa de infraestructura de persistencia de datos, por lo que el modelo de entidad estará limpio y desacoplado de la infraestructura de persistencia.

Fluent API y el método OnModelCreating

Como se mencionó, para cambiar convenciones y mapeos, puede usar el método **`OnModelCreating`** en la clase DbContext.

El microservicio de pedidos en eShopOnContainers implementa mapeo y configuración explícitos, cuando es necesario, como se muestra en el siguiente código.

```
// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   // Other entities' configuration ...
   modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
}

// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
            orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
            orderConfiguration.HasKey(o => o.Id);
            orderConfiguration.Ignore(b => b.DomainEvents);

            orderConfiguration.Property(o => o.Id)
                .ForSqlServerUseSequenceHiLo("orderseq",
                                             OrderingContext.DEFAULT_SCHEMA);
            //Address Value Object persisted as owned entity supported since EF Core 2.0
            orderConfiguration.OwnsOne(o => o.Address);
            orderConfiguration.Property<DateTime>("OrderDate").IsRequired();
            orderConfiguration.Property<int?>("BuyerId").IsRequired(false);
            orderConfiguration.Property<int>("OrderStatusId").IsRequired();
            orderConfiguration.Property<int?>("PaymentMethodId").IsRequired(false);
            orderConfiguration.Property<string>("Description").IsRequired(false);

            var navigation =
                   orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));
            navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

            orderConfiguration.HasOne<PaymentMethod>()
                .WithMany()
                .HasForeignKey("PaymentMethodId")
                .IsRequired(false)
                .OnDelete(DeleteBehavior.Restrict);
            orderConfiguration.HasOne<Buyer>()
                .WithMany()
                .IsRequired(false)
                .HasForeignKey("BuyerId");
            orderConfiguration.HasOne(o => o.OrderStatus)
                .WithMany()
```

```
                .HasForeignKey("OrderStatusId");
    }
}
```

Puede establecer todas las asignaciones de la *Fluent API* dentro del mismo método **`OnModelCreating`**, pero es aconsejable dividir ese código y tener múltiples clases de configuración, una por entidad, como se muestra en el ejemplo. Especialmente para modelos particularmente grandes, es aconsejable tener clases de configuración separadas para cada tipo de entidad.

El código en el ejemplo muestra algunas declaraciones y mapeos explícitos. Sin embargo, las convenciones de EF Core realizan muchas de esas asignaciones automáticamente, por lo que el código real que necesitaría en su caso podría ser más pequeño.

El algoritmo Hi/Lo en EF Core

Un aspecto interesante del código en el ejemplo anterior es que utiliza el algoritmo Hi/Lo como la estrategia de generación de claves.

El algoritmo Hi/Lo es útil cuando necesita claves únicas. Como resumen, el algoritmo Hi/Lo asigna identificadores únicos a las filas de la tabla, pero no depende de guardar la fila en la base de datos de inmediato. Esto le permite comenzar a utilizar los identificadores de inmediato, como sucede con las ID de bases de datos secuenciales regulares.

El algoritmo Hi/Lo describe un mecanismo para generar identificaciones seguras en el lado del cliente, en lugar de en la base de datos. *Seguro* en este contexto significa sin colisiones. Este algoritmo es interesante por estas razones:

- No rompe el patrón de Unidad de trabajo.
- No requiere viajes de ida y vuelta como lo hacen los generadores de secuencia en otros DBMSs.
- Genera un identificador legible por humanos, a diferencia de las técnicas que usan GUIDs.

EF Core soporta HiLo con el método **`ForSqlServerUseSequenceHiLo`**, como se muestra en el ejemplo anterior.

Mapeando campos en vez de propiedades

Con esta característica, disponible a partir de EF Core 1.1, puede asignar directamente columnas a campos. Es posible no usar propiedades en la clase de entidad y sólo asignar columnas de una tabla a campos. Un uso común para eso sería campos privados para cualquier estado interno al que no se necesite acceder desde fuera de la entidad.

Puede hacerlo con campos individuales o también con colecciones, como un campo **`List<>`**. Este punto se mencionó anteriormente cuando discutimos el modelado de las clases de modelo de dominio, pero aquí se puede ver cómo se realiza esa asignación con la configuración de **`PropertyAccessMode.Field`** resaltada en el código anterior.

Usando propiedades *shadow* en EF Core, escondidas a nivel de infraestructura

Las propiedades *shadow* en EF Core son propiedades que no existen en el modelo de clase de entidad. Los valores y estados de estas propiedades se mantienen puramente en la clase **`ChangeTracker`** en el nivel de infraestructura.

Implementando el patrón de Especificación

Como se presentó anteriormente en la sección de diseño, el patrón de Especificación (su nombre completo sería Patrón de especificación de consulta) es un patrón de DDD, diseñado como el lugar donde puede poner la definición de una consulta con ordenamiento opcional y lógica de paginación.

El patrón de especificación define una consulta en un objeto. Por ejemplo, para encapsular una consulta paginada que busca algunos productos, puede crear una especificación de **`PagedProduct`** que tome los parámetros de entrada necesarios (**`pageNumber`**, **`pageSize`**, **`filter`**, etc.). Luego, dentro de cualquier método de repositorio (generalmente una sobrecarga de **`List()`**), aceptaría una **`ISpecification`** y ejecutaría la consulta esperada basada en esa especificación.

Un ejemplo de una interfaz de especificación genérica es el siguiente código de eShopOnWeb.

```
// GENERIC SPECIFICATION INTERFACE
// https://github.com/dotnet-architecture/eShopOnWeb

public interface ISpecification<T>
{
    Expression<Func<T, bool>> Criteria { get; }
    List<Expression<Func<T, object>>> Includes { get; }
    List<string> IncludeStrings { get; }
}
```

Entonces, la implementación de una clase base de especificación genérica es la siguiente.

```
// GENERIC SPECIFICATION IMPLEMENTATION (BASE CLASS)
// https://github.com/dotnet-architecture/eShopOnWeb

public abstract class BaseSpecification<T> : ISpecification<T>
{
    public BaseSpecification(Expression<Func<T, bool>> criteria)
    {
        Criteria = criteria;
    }
    public Expression<Func<T, bool>> Criteria { get; }

    public List<Expression<Func<T, object>>> Includes { get; } =
                                            new List<Expression<Func<T, object>>>();

    public List<string> IncludeStrings { get; } = new List<string>();

    protected virtual void AddInclude(Expression<Func<T, object>> includeExpression)
    {
        Includes.Add(includeExpression);
    }

    // string-based includes allow for including children of children
    // e.g. Basket.Items.Product
    protected virtual void AddInclude(string includeString)
    {
        IncludeStrings.Add(includeString);
    }
}
```

La siguiente especificación carga una entidad de carrito de compras individual, ya sea con la identificación del carrito o la identificación del comprador al que pertenece el carrito y se cargará de una vez (*eager loading*) la colección de artículos del carrito.

```
// SAMPLE QUERY SPECIFICATION IMPLEMENTATION

public class BasketWithItemsSpecification : BaseSpecification<Basket>
{
    public BasketWithItemsSpecification(int basketId)
        : base(b => b.Id == basketId)
    {
        AddInclude(b => b.Items);
    }
    public BasketWithItemsSpecification(string buyerId)
        : base(b => b.BuyerId == buyerId)
    {
        AddInclude(b => b.Items);
    }
}
```

Por último, puede ver a continuación cómo un repositorio de EF genérico puede usar dicha especificación para filtrar y cargar de una vez los datos relacionados con una entidad dada tipo T.

```
// GENERIC EF REPOSITORY WITH SPECIFICATION
// https://github.com/dotnet-architecture/eShopOnWeb

public IEnumerable<T> List(ISpecification<T> spec)
{
    // fetch a Queryable that includes all expression-based includes
    var queryableResultWithIncludes = spec.Includes
        .Aggregate(_dbContext.Set<T>().AsQueryable(),
            (current, include) => current.Include(include));

    // modify the IQueryable to include any string-based include statements
    var secondaryResult = spec.IncludeStrings
        .Aggregate(queryableResultWithIncludes,
            (current, include) => current.Include(include));

    // return the result of the query using the specification's criteria expression
    return secondaryResult
                    .Where(spec.Criteria)
                    .AsEnumerable();
}
```

Además de encapsular la lógica de filtrado, la especificación puede especificar la forma de los datos que se devolverán, incluidas las propiedades que se rellenarán.

Aunque no se recomienda devolver **`IQueryable`** desde un repositorio, está perfectamente bien usarlos dentro del repositorio para crear un conjunto de resultados. Puede ver este enfoque, usado en el método de Lista anterior, que utiliza expresiones **`IQueryable`** intermedias para crear la lista de inclusiones de la consulta antes de ejecutarla con los criterios de la especificación en la última línea.

Recursos adicionales

- **Asignación de tabla**
  https://docs.microsoft.com/ef/core/modeling/relational/tables

- **Use HiLo to generate keys with Entity Framework Core**
  http://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/
- **Campos de respaldo**
  https://docs.microsoft.com/ef/core/modeling/backing-field
- **Steve Smith. Encapsulated Collections in Entity Framework Core**
  http://ardalis.com/encapsulated-collections-in-entity-framework-core
- **Shadow Properties**
  https://docs.microsoft.com/ef/core/modeling/shadow-properties
- **The Specification pattern**
  http://deviq.com/specification-pattern/

Usando bases de datos NoSQL como infraestructura de persistencia

Cuando utiliza bases de datos NoSQL para la infraestructura de datos, normalmente no utiliza un ORM como Entity Framework Core. En su lugar, utiliza la API proporcionada por el motor NoSQL, como Azure Cosmos DB, MongoDB, Cassandra, RavenDB, CouchDB o Azure Storage Tables.

Sin embargo, cuando utiliza una base de datos NoSQL, especialmente una base de datos orientada a documentos como Azure Cosmos DB, CouchDB o RavenDB, la forma en que diseña su modelo con agregados DDD es parcialmente similar a como puede hacerlo en EF Core, con respecto a la identificación de raíces de agregación, entidades hijo y *value objects*. Pero, en última instancia, la selección de la base de datos tendrá un impacto en su diseño.

Cuando utiliza una base de datos orientada a documentos, se implementa un agregado como un documento único, serializado en JSON u otro formato. Sin embargo, el uso de la base de datos es transparente desde el punto de vista de un código de modelo de dominio. Al usar una base de datos NoSQL, todavía está usando clases de entidad y clases raíz de agregados, pero con más flexibilidad que cuando usa EF Core porque la persistencia no es relacional.

La diferencia está en cómo se persiste ese modelo. Si implementó su modelo de dominio basado en clases de entidad POCO, ignorantes de infraestructura de persistencia, podría parecer que se debería poder pasar a una infraestructura de persistencia diferente, incluso de relacional a NoSQL. Sin embargo, ese no debería ser su objetivo. Siempre hay restricciones en las diferentes bases de datos que le harán retroceder, por lo que no podrá tener el mismo modelo para bases de datos relacionales o NoSQL. Cambiar los modelos de persistencia no sería trivial, porque las transacciones y las operaciones de persistencia serán muy diferentes.

Por ejemplo, en una base de datos orientada a documentos, está bien que una raíz de agregación tenga múltiples colecciones de entidades secundarias. En una base de datos relacional, consultar varias de colecciones de este tipo es muy poco eficiente, porque se obtiene una sentencia SQL UNION ALL de EF. Tener el mismo modelo de dominio para bases de datos relacionales o bases de datos NoSQL no es simple y no debe intentarlo. Realmente debe diseñar su modelo con una comprensión de cómo se usarán los datos en cada base de datos en particular.

Un beneficio al usar bases de datos NoSQL es que las entidades están más desnormalizadas, por lo que no se establece una asignación de tablas. Su modelo de dominio puede ser más flexible que cuando usa una base de datos relacional.

Cuando diseñe su modelo de dominio basado en agregados, moverse a NoSQL y bases de datos orientadas a documentos podría ser incluso más fácil que usar una base de datos relacional, porque los agregados que diseña son similares a los documentos serializados en una base de datos orientada a documentos. Luego puede incluir en esas "bolsas" toda la información que pueda necesitar para ese agregado.

Por ejemplo, el siguiente código JSON es un ejemplo de un agregado de pedido, cuando se usa una base de datos orientada a documentos. Es similar al agregado de pedido que implementamos en el ejemplo de eShopOnContainers, pero sin usar EF Core.

```
{
    "id": "2017001",
    "orderDate": "2/25/2017",
    "buyerId": "1234567",
    "address": [
        {
            "street": "100 One Microsoft Way",
            "city": "Redmond",
            "state": "WA",
            "zip": "98052",
            "country": "U.S."
        }
    ],
    "orderItems": [
        {"id": 20170011, "productId": "123456", "productName": ".NET T-Shirt",
             "unitPrice": 25, "units": 2, "discount": 0},
        {"id": 20170012, "productId": "123457", "productName": ".NET Mug",
             "unitPrice": 15, "units": 1, "discount": 0}
    ]
}
```

Introducción a Azure Cosmos DB la API nativa de Cosmos DB

Azure Cosmos DB es el servicio de base de datos distribuido globalmente de Microsoft para aplicaciones de misión crítica. Azure Cosmos DB proporciona distribución global llave en mano, escalado elástico de rendimiento y almacenamiento en todo el mundo, latencias de un dígito en milisegundos en el percentil 99, cinco niveles de consistencia bien definidos y alta disponibilidad garantizada, todos respaldados por SLA líderes en la industria. Azure Cosmos DB indexa automáticamente los datos sin necesidad de que se ocupe de la administración de esquema ni de los índices. Es multi modelo y soporta modelos de documentos, clave-valor, grafos y columnas.

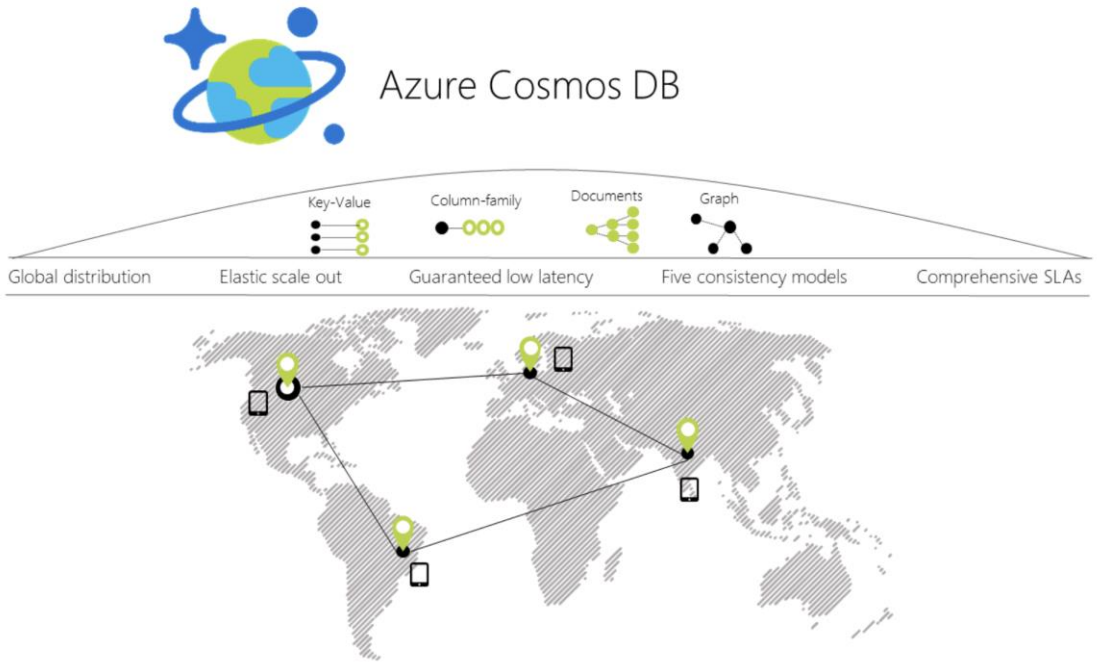


***Figura 7-19**. Distribución global de Azure Cosmos DB*

Cuando utiliza un modelo C# para implementar el agregado que usará la API de Azure Cosmos DB, el agregado puede ser similar a las clases POCO de C# utilizadas con EF Core. La diferencia está en la forma de usarlos desde las capas de aplicación e infraestructura, como en el siguiente código:

```
// C# EXAMPLE OF AN ORDER AGGREGATE BEING PERSISTED WITH AZURE COSMOS DB API
// *** Domain Model Code ***
// Aggregate: Create an Order object with its child entities and/or value objects.
// Then, use AggregateRoot's methods to add the nested objects & apply invariants.
Order orderAggregate = new Order
{
    Id = "2017001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "PO18009186470"
}
Address address = new Address
{
   Street = "100 One Microsoft Way",
   City = "Redmond",
   State = "WA",
   Zip = "98052",
   Country = "U.S."
}

orderAggregate.UpdateAddress(address);

OrderItem orderItem1 = new OrderItem
{
   Id = 20170011,
   ProductId = "123456",
   ProductName = ".NET T-Shirt",
   UnitPrice = 25,
   Units = 2,
   Discount = 0;
};

//Using methods with domain logic within the entity. No anemic-domain model
orderAggregate.AddOrderItem(orderItem1);
// *** End of Domain Model Code ***

// *** Infrastructure Code using Cosmos DB Client API ***
Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName,
                                                           collectionName);
await client.CreateDocumentAsync(collectionUri, order);

// As your app evolves, let's say your object has a new schema. You can insert
// OrderV2 objects without any changes to the database tier.
Order2 newOrder = GetOrderV2Sample("IdForSalesOrder2");
await client.CreateDocumentAsync(collectionUri, newOrder);
```

Puede ver que la forma en que trabaja con su modelo de dominio puede ser similar a la que usa cuando la infraestructura es EF. Aún usa los mismos métodos de raíz de agregación para garantizar la consistencia, invariantes y validaciones dentro del agregado.

Sin embargo, cuando persiste su modelo en la base de datos NoSQL, el código y la API cambian dramáticamente en comparación con el código EF Core o cualquier otro código relacionado con las bases de datos relacionales.

Implementando código .NET apuntando a MongoDB y Azure Cosmos DB

Usando Azure Cosmos DB desde contenedores .NET

Puede acceder a las bases de datos de Azure Cosmos DB desde el código .NET que se ejecuta en contenedores, como desde cualquier otra aplicación .NET. Por ejemplo, los microservicios Locations.API y Marketing.API en eShopOnContainers se implementan para que puedan consumir las bases de datos de Azure Cosmos DB.

Sin embargo, hay una limitación en Azure Cosmos DB, desde el punto de vista del entorno de desarrollo Docker. Incluso aunque hay un emulador de bases de datos Azure Cosmos DB que puede ejecutarse en una máquina de desarrollo local (como una PC), la menos hasta finales de 2017, sólo soporta Windows, no Linux.

También existe la posibilidad de ejecutar este emulador en Docker, pero sólo en Contenedores Windows, no Linux. Esa es una desventaja inicial para el entorno de desarrollo si su aplicación se implementa como contenedores Linux, ya que, actualmente, no puede implementar Contenedores Linux y Windows en Docker para Windows al mismo tiempo. O bien todos los contenedores que se implementan tienen que ser para Linux o para Windows.

El despliegue ideal y más sencillo para una solución de desarrollo/pruebas es poder desplegar sus sistemas de bases de datos como contenedores junto con sus contenedores personalizados, para que sus entornos de desarrollo/pruebas sean siempre consistentes.

Use el API de MongoDB para contenedores locales Linux/Windows para desarrollo/pruebas más Azure Cosmos DB

Las bases de datos de Cosmos DB soportan la API de MongoDB para .NET, así como el protocolo nativo MongoDB. Esto significa que, al usar los controladores existentes, la aplicación escrita para MongoDB ahora puede comunicarse con Cosmos DB y usar las bases de datos de Cosmos DB en lugar de las bases de datos de MongoDB, como se muestra en la Figura 7-20.

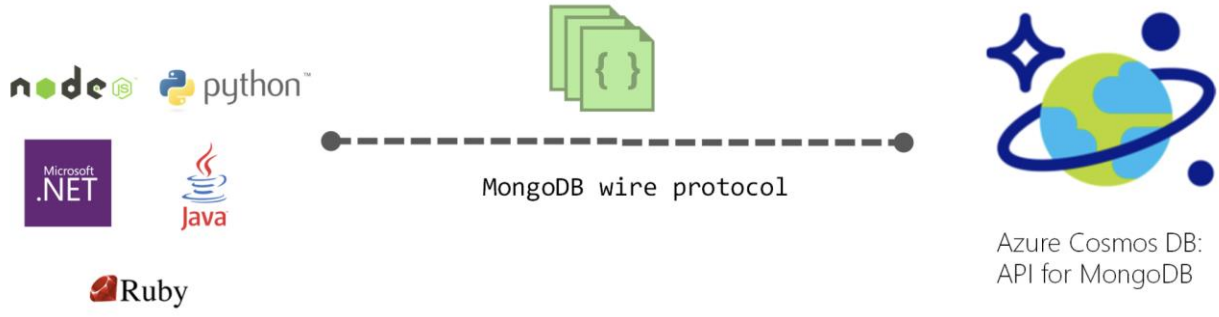


***Figura 7-20**. Usando la API y el protocolo de MongoDB para acceder a Azure Cosmos DB*

Este es un enfoque muy conveniente para las pruebas de conceptos en entornos Docker con contenedores Linux porque la imagen MongoDB Docker es una imagen multi-arch que soporta contenedores Docker Linux y Windows.

Como se muestra en la imagen 9-21, al usar la API de MongoDB, eShopOnContainers soporta contenedores MongoDB Linux y Windows para el entorno de desarrollo local, pero luego puede pasar a una solución en la nube PaaS escalable como Azure Cosmos DB simplemente cambiando la cadena de conexión de MongoDB para apuntar a Azure Cosmos DB.

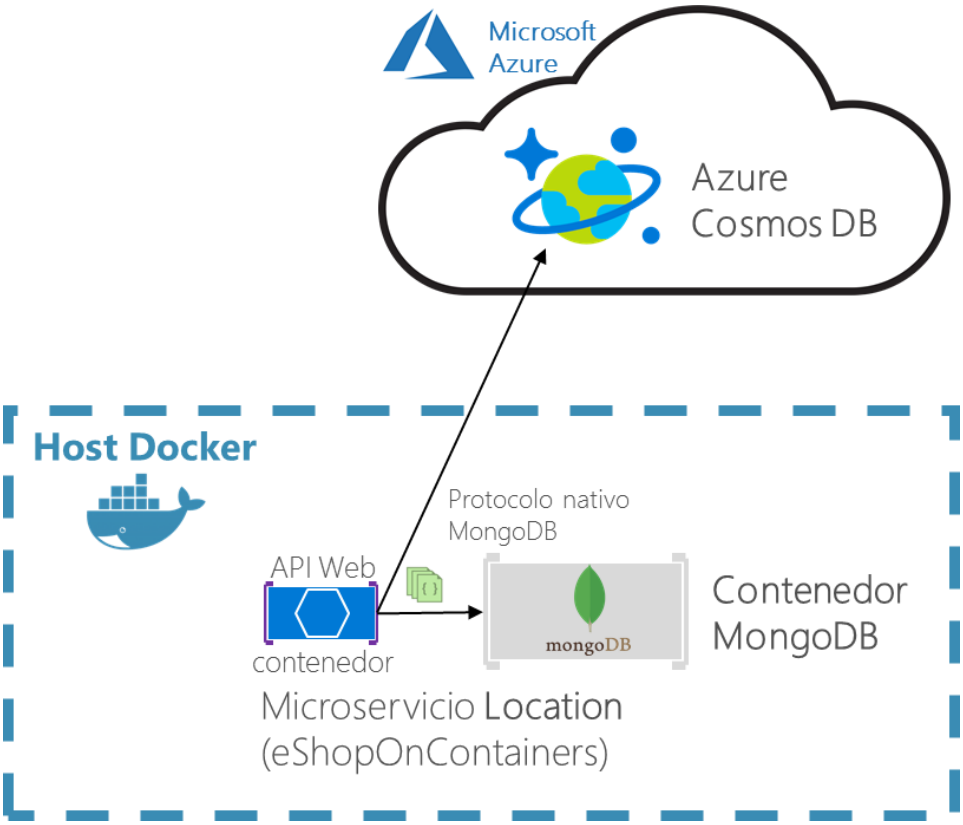


Figura 7-21. *eShopOnContainers usando contenedores MongoDB para desarrollo y Azure Cosmos DB para producción*

La instancia de producción de Azure Cosmos DB se estaría ejecutando en la nube de Azure, como un servicio PaaS escalable.

Sus contenedores .NET Core personalizados pueden ejecutarse en un *host* Docker de desarrollo local (que usa Docker para Windows en una máquina con Windows 10) o desplegarse en un entorno de producción, como Kubernetes en Azure AKS o Azure Service Fabric. En este segundo entorno, sólo desplegaría los contenedores personalizados .NET Core pero no el contenedor MongoDB, ya que usaría Azure Cosmos DB en la nube para manejar los datos en producción.

Un beneficio claro si utiliza la API de MongoDB es que su solución podría ejecutarse en ambos motores de base de datos, MongoDB o Azure Cosmos DB, por lo que las migraciones a diferentes entornos deberían ser fáciles. Sin embargo, a veces vale la pena utilizar una API nativa (que es la API nativa de Cosmos DB) para aprovechar al máximo las capacidades de un motor de base de datos específico.

Para una mejor comparación entre simplemente usar MongoDB versus Cosmos DB en la nube, consulte los beneficios de usar Azure Cosmos DB en esta página.

Analice su enfoque para aplicaciones en producción: el API de MongoDB versus el API de Cosmos DB

En eShopOnContainers estamos utilizando la API de MongoDB porque nuestra prioridad era fundamentalmente tener un entorno consistente de desarrollo/pruebas, utilizando una base de datos NoSQL que también pudiese funcionar con Azure Cosmos DB.

Sin embargo, si planea usar la API de MongoDB para acceder a Azure Cosmos DB en Azure para aplicaciones de producción, primero debe analizar cuáles son las diferencias en capacidades y rendimiento al usar la API de MongoDB para acceder a Azure Cosmos DB, en comparación con el uso de la API nativa de Azure Cosmos DB. Dependiendo del caso, podría ser similar para que pueda seguir usando la API de MongoDB y obtenga la ventaja de soportar dos motores de base de datos NoSQL al mismo tiempo, pero también podría ser diferente si hay alguna capacidad especial en Azure Cosmos DB que no funciona de la misma manera cuando se usa la API de MongoDB.

Por el contrario, también podría usar los *clusters* de MongoDB como la base de datos de producción en la nube de Azure, también el con MongoDB Azure Service. Pero ese no es un servicio PaaS provisto por Microsoft. En este caso, Azure solo está hospedando esa solución proveniente de MongoDB.

Básicamente, esto es sólo un descargo de responsabilidad para aclarar que no siempre se debe usar la API de MongoDB contra Azure Cosmos DB, como lo estamos haciendo en eShopOnContainers, porque era una opción conveniente para los contenedores de Linux. La decisión debe basarse en las necesidades y pruebas específicas que debe hacer para su aplicación en producción.

El código: usando MongoDB en aplicaciones .NET Core

La API de MongoDB para .NET se basa en paquetes NuGet que debe agregar a sus proyectos, como en Locations.API que se muestra en la siguiente imagen.

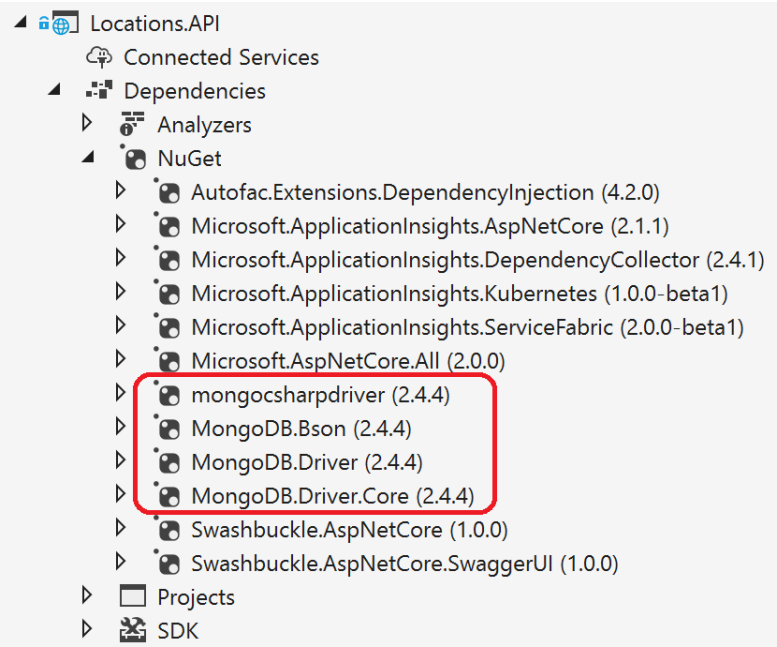


***Figura 7-22**. Referencias a los paquetes NuGet de MongoDB en un proyecto .NET Core*

Con respecto al código, hay varias áreas para investigar, como se explica en las siguientes secciones.

Un modelo usado por la API de MongoDB

En primer lugar, debe definir un modelo, que contendrá los datos provenientes de la base de datos, en el espacio de memoria de su aplicación. Aquí hay un ejemplo del modelo utilizado para **`Locations`** en eShopOnContainers.

```
using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Driver.GeoJsonObjectModel;
using System.Collections.Generic;

public class Locations
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    public int LocationId { get; set; }
    public string Code { get; set; }
    [BsonRepresentation(BsonType.ObjectId)]
    public string Parent_Id { get; set; }
    public string Description { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public GeoJsonPoint<GeoJson2DGeographicCoordinates> Location
                                                        { get; private set; }
    public GeoJsonPolygon<GeoJson2DGeographicCoordinates> Polygon
                                                        { get; private set; }
    public void SetLocation(double lon, double lat) => SetPosition(lon, lat);
    public void SetArea(List<GeoJson2DGeographicCoordinates> coordinatesList)
                                                  => SetPolygon(coordinatesList);

    private void SetPosition(double lon, double lat)
    {
        Latitude = lat;
        Longitude = lon;
        Location = new GeoJsonPoint<GeoJson2DGeographicCoordinates>(
            new GeoJson2DGeographicCoordinates(lon, lat));
    }

    private void SetPolygon(List<GeoJson2DGeographicCoordinates> coordinatesList)
    {
        Polygon = new GeoJsonPolygon<GeoJson2DGeographicCoordinates>(
                  new GeoJsonPolygonCoordinates<GeoJson2DGeographicCoordinates>(
                  new GeoJsonLinearRingCoordinates<GeoJson2DGeographicCoordinates>(
                                                            coordinatesList)));
    }
}
```

Puede ver que hay algunos atributos y tipos provenientes de los paquetes NuGet de MongoDB.

Las bases de datos NoSQL suelen ser muy adecuadas para trabajar con datos no relacionales, pero jerárquicos. En este ejemplo, estamos usando tipos de MongoDB especialmente diseñados para geo-localizaciones, como **GeoJson2DGeographicCoordinates**.

Recuperar la base de datos y la colección

En eShopOnContainers, hemos creado un contexto de base de datos personalizado donde implementamos el código para recuperar la base de datos y las MongoCollections, como en el siguiente código.

```
public class LocationsContext
{
    private readonly IMongoDatabase _database = null;

    public LocationsContext(IOptions<LocationSettings> settings)
    {
        var client = new MongoClient(settings.Value.ConnectionString);
        if (client != null)
            _database = client.GetDatabase(settings.Value.Database);
    }
    public IMongoCollection<Locations> Locations
    {
        get
        {
            return _database.GetCollection<Locations>("Locations");
        }
    }
}
```

Recuperar los datos

En C#, como los controladores de API Web o repositorios personalizados, se puede escribir un código similar al siguiente cuando realiza una consulta a través de la API de MongoDB, siendo el objeto **_context** una instancia de la clase **LocationsContext** anterior.

```
public async Task<Locations> GetAsync(int locationId)
{
    var filter = Builders<Locations>.Filter.Eq("LocationId", locationId);
    return await _context.Locations
                         .Find(filter)
                         .FirstOrDefaultAsync();
}
```

Use una env-var en el fichero docker-compose.override.yml para la cadena de conexión a MongoDB

Al crear un objeto **MongoClient**, se necesita un parámetro fundamental, que es precisamente el **ConnectionString** apuntando, en el caso de eShopOnContainers, a un contenedor MongoDB Docker local o a la base de datos de "producción" de Azure Cosmos DB. Esa cadena de conexión proviene de las variables de entorno definidas en los ficheros **docker-compose.override.yml** cuando se implementa con **docker-compose** o Visual Studio, como en el siguiente código yml.

```
# docker-compose.override.yml
version: '3'
services:
  # Other services
  locations.api:
    environment:
      # Other settings
      - ConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosql.data}
```

La variable de entorno **ConnectionString** se resuelve de está forma: si está definida la variable global **ESHOP_AZURE_COSMOSDB** en el fichero .env con la cadena de conexión de Azure Cosmos DB, la usará para acceder a esta base de datos en la nube. Si no está definida, tomará el valor **mongodb://nosql.data** y usará el contenedor de desarrollo de mongodb.

En el código siguiente puede ver el fichero .env con la variable de entorno global con la cadena de conexión al Azure Cosmos DB, como está implementado en eShopOnContainers.

```
# .env file, in eShopOnContainers root folder
# Other Docker environment variables
ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost
ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=<YourDockerHostIP>

#ESHOP_AZURE_COSMOSDB=<YourAzureCosmosDBConnData>

#Other environment variables for additional Azure infrastructure assets
#ESHOP_AZURE_REDIS_BASKET_DB=<YourAzureRedisBasketInfo>
#ESHOP_AZURE_SERVICE_BUS=<YourAzureServiceBusInfo>
```

Debe eliminar el comentario de la línea **ESHOP_AZURE_COSMOSDB** y actualizarla con su cadena de conexión real para Azure Cosmos DB, obtenida del Azure Portal como se explica en este tutorial: Conectar una aplicación MongoDB a Azure Cosmos DB.

Si la variable global **ESHOP_AZURE_COSMOSDB** está vacía, lo que significa que está comentada en el fichero .env, usará una cadena de conexión por defecto a MongoDB, que apunta al contenedor MongoDB local implementado en eShopOnContainers, que se llama **nosql.data** y se definió en el fichero **docker-compose**, como se muestra en el siguiente código .yml.

```
# docker-compose.yml
version: '3'
services:
  # ...Other services...
  nosql.data:
    image: mongo
```

Recursos adicionales

- **Modelado de datos del documento para bases de datos NoSQL**

https://docs.microsoft.com/azure/cosmos-db/modeling-data

- **Vaughn Vernon. The Ideal Domain-Driven Design Aggregate Store?**
  https://vaughnvernon.co/?p=942
- **Introducción a Azure Cosmos DB: API para MongoDB**
  https://docs.microsoft.com/azure/cosmos-db/mongodb-introduction
- **Azure Cosmos DB: Compilar una aplicación web de API MongoDB con .NET y Azure Portal**
  https://docs.microsoft.com/azure/cosmos-db/create-mongodb-dotnet
- **Uso del Emulador de Azure Cosmos DB para desarrollo y pruebas de forma local**
  https://docs.microsoft.com/azure/cosmos-db/local-emulator
- **Conectar una aplicación de MongoDB a Azure Cosmos DB**
  https://docs.microsoft.com/azure/cosmos-db/connect-mongodb-account
- **The Cosmos DB Emulator Docker image (Windows Container)**
  https://hub.docker.com/r/microsoft/azure-cosmosdb-emulator/
- **The MongoDB Docker image (Linux and Windows Container)**
  https://hub.docker.com/r/_/mongo/
- **Uso de MongoChef con una cuenta de Azure Cosmos DB: API para MongoDB**
  https://docs.microsoft.com/azure/cosmos-db/mongodb-mongochef

Diseñando el microservicio de la capa de aplicación con API Web

Usando los principios SOLID y la Inyección de Dependencias

Los principios SOLID son técnicas clave para ser utilizadas en cualquier aplicación moderna y de misión crítica, como el desarrollo de un microservicio con patrones DDD. SOLID es un acrónimo que agrupa cinco principios fundamentales:

- Single Responsibility principle - Principio de Responsabilidad Única
- Open/closed principle – Principio abierto/cerrado
- Liskov substitution principle - Principio de sustitución de Liskov
- Interface Segregation principle - Principio de segregación de interfaces
- Dependency Inversion principle - Principio de inversión de dependencia

SOLID es más acerca de cómo diseñar su aplicación o capas internas de microservicio y sobre cómo desacoplar dependencias entre ellas. No está relacionado con el dominio, sino con el diseño técnico de la aplicación. El principio final, el principio de Inversión de Dependencia (DI), le permite desacoplar la capa de infraestructura del resto de las capas, lo que permite una mejor implementación desacoplada de las capas DDD.

La Inyección de Dependencias (también DI) es una forma de implementar el principio de Inversión de Dependencia. Es una técnica para lograr un acoplamiento flexible entre los objetos y sus dependencias. En lugar de crear instancias directas de los colaboradores o usar referencias estáticas, los objetos que una clase necesita para realizar sus acciones se proporcionan (o "inyectan") a la clase. Muy a menudo, las clases declararán sus dependencias a través de su constructor, permitiéndoles seguir el principio de Dependencias Explícitas. La Inyección de Dependencias generalmente se basa en contenedores específicos de Inversión de Control (IoC). ASP.NET Core proporciona un contenedor IoC integrado simple, pero también puede usar su contenedor IoC favorito, como Autofac o Ninject.

Siguiendo los principios SOLID, sus clases tenderán naturalmente a ser pequeñas, bien factorizadas y fáciles de probar. Pero, ¿cómo puede saber si se están inyectando demasiadas dependencias en sus clases? Si usa DI a través del constructor, será fácil detectarlo simplemente mirando el número de parámetros del constructor. Si hay demasiadas dependencias, esto generalmente es una señal (un *code smell* – algo que no huele bien) de que su clase está tratando de hacer demasiado y probablemente esté violando el principio de Responsabilidad Única.

Tomaría otra guía para cubrir SOLID en detalle. Por lo tanto, esta guía requiere que sólo tenga un conocimiento mínimo de estos temas y esté consciente de ellos.

Recursos adicionales

- **SOLID: Fundamental OOP Principles**
  http://deviq.com/solid/
- **Inversion of Control Containers and the Dependency Injection pattern**
  https://martinfowler.com/articles/injection.html
- **Steve Smith. New is Glue**
  http://ardalis.com/new-is-glue

Implementando el microservicio de la capa de aplicación usando API Web

Usando Inyección de Dependencias para usar objetos de infraestructura en la capa de la aplicación

Como se mencionó anteriormente, la capa de aplicación se puede implementar tanto como parte del artefacto (ensamblado) que está construyendo, como dentro de un proyecto de API Web o un proyecto de aplicación web MVC. En el caso de un microservicio creado con ASP.NET Core, la capa de aplicación generalmente será su librería API Web. Si desea separar lo que proviene de ASP.NET Core (su infraestructura más sus controladores) del código de capa de aplicación personalizada, también puede colocar su capa de aplicación en una librería de clases separada, pero eso es opcional.

Por ejemplo, el código de la capa de aplicación del microservicio de pedidos se implementa directamente como parte del proyecto **Ordering.API** (un proyecto ASP.NET Core Web API), como se muestra en la figura 7-23.

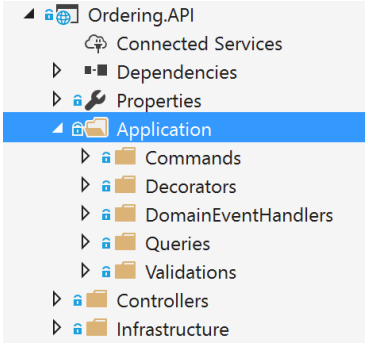


***Figura 7-23**. La capa de aplicación en el proyecto ASP.NET Core Web API Ordering.API*

ASP.NET Core incluye un contenedor IoC simple integrado (representado por la interfaz **`IServiceProvider`**) que soporta por defecto la inyección por constructor y ASP.NET hace que ciertos servicios estén disponibles a través de DI. ASP.NET Core utiliza el término *servicio* para cualquiera de los tipos que registra y que se inyectará a través de DI. Los servicios del contenedor integrado se configuran en el método **`ConfigureServices`** en la clase **`Startup`** de su aplicación. Sus dependencias se implementan en los servicios que necesitan los tipos y que también se registran en el contenedor IoC.

Normalmente, desea inyectar dependencias que implementen objetos de infraestructura. Una dependencia muy típica para inyectar es un repositorio. Pero podría inyectar cualquier otra dependencia de infraestructura que pueda tener. Para implementaciones más simples, puede inyectar directamente su objeto de patrón de unidad de trabajo (el objeto DbContext de EF), porque el **`DBContext`** es también la implementación de los objetos de persistencia de su infraestructura.

En el siguiente ejemplo, puede ver cómo .NET Core está inyectando los objetos requeridos del repositorio a través del constructor. La clase es un manejador de comandos, que trataremos en la siguiente sección.

```
// Sample command handler

public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                     IOrderRepository orderRepository,
                                     IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
                           throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
                           throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
                                throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Create the Order AggregateRoot
        // Add child entities and value objects through the Order aggregate root
        // methods and constructor so validations, invariants, and business logic
        // make sure that consistency is preserved across the whole aggregate

        var address = new Address(message.Street, message.City, message.State,
                                  message.Country, message.ZipCode);
        var order = new Order(message.UserId, address, message.CardTypeId,
                              message.CardNumber, message.CardSecurityNumber,
                              message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                               item.Discount, item.PictureUrl, item.Units);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}
```

La clase usa los repositorios inyectados para ejecutar la transacción y persistir los cambios de estado. No importa si esa clase es un manejador de comandos, un método de controlador ASP.NET Web API o un servicio de aplicación DDD. En última instancia, es una clase simple que usa repositorios, entidades del dominio y otras, para coordinar la aplicación, de forma similar a un manejador de

comandos. La Inyección de Dependencias (DI) funciona de la misma manera para todas las clases mencionadas, como en el ejemplo, que usa DI basado en el constructor.

Registrando los tipos de implementación de dependencias y las interfaces o abstracciones

Antes de usar los objetos inyectados a través de los constructores, necesita saber dónde registrar las interfaces y las clases que producen los objetos inyectados en las clases de la aplicación, a través de DI. (Usando DI basada en el constructor, como se mostró anteriormente).

Usando el contenedor IoC integrado de ASP.NET Core

Cuando utiliza el contenedor integrado de IoC proporcionado por ASP.NET Core, debe registrar los tipos que desea inyectar en el método **`ConfigureServices`** en el fichero **`Startup.cs`**, como en el siguiente código:

```
// Registration of types into ASP.NET Core built-in container
public void ConfigureServices(IServiceCollection services)
{
    // Register out-of-the-box framework services.
    services.AddDbContext<CatalogContext>(c =>
    {
        c.UseSqlServer(Configuration["ConnectionString"]);
    },
    ServiceLifetime.Scoped
    );
    services.AddMvc();
    // Register custom application dependencies.
    services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();
}
```

El patrón más común al registrar tipos en un contenedor IoC es registrar un par de tipos: una interfaz y su clase de implementación relacionada. Luego, cuando solicita un objeto desde el contenedor IoC a través de cualquier constructor, solicita un objeto de cierto tipo de interfaz. Por ejemplo, en el código anterior, la última línea indica que cuando alguno de sus constructores tiene una dependencia en **`IMyCustomRepository`** (interfaz o abstracción), el contenedor IoC inyectará una instancia de la clase de implementación **`MyCustomSQLServerRepository`**.

Usando la librería Scrutor para registro automático de tipos

Al usar DI en .NET Core, es posible que desee poder escanear un ensamblado y registrar automáticamente sus tipos por convención. Esta función no está disponible actualmente en ASP.NET Core. Sin embargo, puede usar la librería Scrutor para eso. Este enfoque es conveniente cuando tiene docenas de tipos que deben registrarse en su contenedor IoC.

Recursos adicionales

- **Matthew King. Registering services with Scrutor**
  https://mking.io/blog/registering-services-with-scrutor
- **Kristian Hellang. Scrutor.** GitHub repo.
  https://github.com/khellang/Scrutor

Usando Autofac como contenedor IoC

También puede usar contenedores IoC adicionales y conectarlos a la línea de proceso de peticiones (*pipeline)* de ASP.NET Core, como en el servicio de microservicio de pedidos en eShopOnContainers, que usa Autofac. Al usar Autofac, generalmente registra los tipos a través de módulos, lo que le permite dividir los tipos de registro entre múltiples ficheros dependiendo de dónde estén sus tipos, del mismo modo que podría tener los tipos de aplicaciones distribuidos en múltiples librerías de clases.

Por ejemplo, el siguiente es el módulo Autofac de la aplicación para el proyecto API Web Ordering.API con los tipos que serán inyectados.

```
public class ApplicationModule : Autofac.Module
{
    public string QueriesConnectionString { get; }

    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }
    protected override void Load(ContainerBuilder builder)
    {
        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>()
            .InstancePerLifetimeScope();

        builder.RegisterType<BuyerRepository>()
            .As<IBuyerRepository>()
            .InstancePerLifetimeScope();

        builder.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();

        builder.RegisterType<RequestManager>()
           .As<IRequestManager>()
           .InstancePerLifetimeScope();
    }
}
```

Autofac también tiene facilidades para escanear ensamblados y registrar tipos por convenciones de nombre.

El proceso y los conceptos de registro son muy similares a la forma en que puede registrar tipos con el contenedor ASP.NET Core iOS integrado, pero la sintaxis al usar Autofac es un poco diferente.

En el código de ejemplo, la abstracción **`IOrderRepository`** se registra junto con la clase de implementación **`OrderRepository`**. Esto significa que siempre que un constructor declare una dependencia a través de la abstracción o interfaz **`IOrderRepository`**, el contenedor IoC inyectará una instancia de la clase **`OrderRepository`**.

El tipo de ámbito de instancia determina cómo se comparte una instancia entre las peticiones para el mismo servicio o dependencia. Cuando se realiza la solicitud para una dependencia, el contenedor IoC puede devolver lo siguiente:

- Una instancia única por la vida del ámbito (**`InstancePerLifetimeScope`** - a la que se hace referencia en el contenedor IoC de ASP.NET Core como *scoped*).
- Una nueva instancia por dependencia (**`InstancePerDependency`** - a la que se hace referencia en el contenedor IoC de ASP.NET Core como *transient*).
- Una instancia única compartida en todos los objetos que utilizan el contenedor IoC (**`SingleInstance`** - al que se hace referencia en el contenedor IoC de ASP.NET Core como *singleton*).

Recursos adicionales

- **Introducción a la Inyección de Dependencias en ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/fundamentals/dependency-injection
- **Autofac.** Official documentation.
  http://docs.autofac.org/en/latest/
- **Comparing ASP.NET Core IoC container service lifetimes with Autofac IoC container instance scopes - Cesar de la Torre**
  https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Implementando los patrones Comando y Manejador de Comandos

En el ejemplo de DI a través del constructor mostrado en la sección anterior, el contenedor IoC estaba inyectando repositorios a través de un constructor en una clase. Pero, exactamente ¿dónde fueron inyectados? En una API Web simple (por ejemplo, el microservicio de catálogo en eShopOnContainers), los inyecta en el nivel de controladores MVC, en un constructor de controladores, como parte del *request pipeline* de ASP.NET Core. Sin embargo, en el código inicial de esta sección (la clase `CreateOrderCommandHandler` del servicio **Ordering.API** en eShopOnContainers), la inyección de dependencias se realiza mediante el constructor de un manejador de comandos en particular. Permítanos explicarle qué es un manejador de comando y por qué le gustaría usarlo.

El patrón de comando está intrínsecamente relacionado con el patrón de CQRS que se introdujo anteriormente en esta guía. CQRS tiene dos lados. El primero es las consultas, usando consultas simplificadas con el micro ORM de Dapper, que explicamos anteriormente. El segundo lado son los comandos, que son el punto de partida para las transacciones y el canal de entrada desde el exterior del servicio.

Como se muestra en la figura 7-24, el patrón se basa en aceptar comandos del lado del cliente, procesarlos según las reglas del modelo de dominio y finalmente persistir los estados con transacciones.

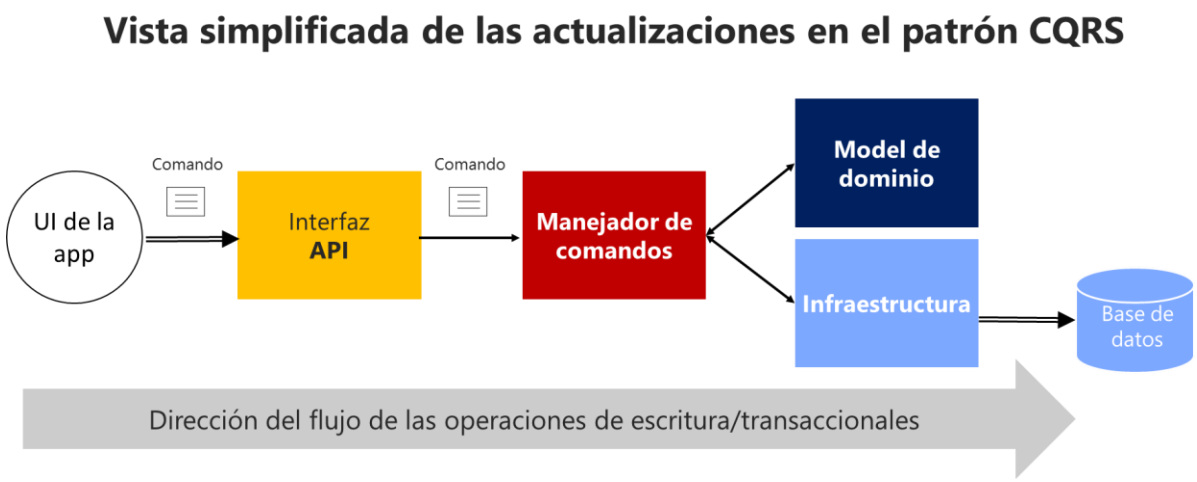


***Figura 7-24**. Vista simplificada, de alto nivel, de los comandos o el "lado transaccional" en el patrón CQRS*

La clase Comando

Un comando es una solicitud para que el sistema realice una acción que cambie el estado del sistema. Los comandos son imprescindibles y se deben procesar sólo una vez.

Dado que los comandos son imperativos, normalmente se nombran con un verbo en el modo infinitivo (por ejemplo, "crear" o "actualizar") y pueden incluir el tipo agregado, como **`CreateOrderCommand`**. A diferencia de un evento, un comando no es un hecho del pasado, es sólo una solicitud y por lo tanto puede ser rechazada.

Los comandos se pueden originar desde la interfaz de usuario, como resultado de un usuario que inicia una *petición* o desde un administrador de procesos, cuando éste le está pidiendo a un agregado que realice una acción.

Una característica importante de un comando es que se debe procesar sólo una vez por un solo receptor. Esto se debe a que un comando es una acción o transacción única que se desea realizar en la aplicación. Por ejemplo, el mismo comando de creación de orden no se debe procesar más de una vez. Esta es una diferencia importante entre comandos y eventos. Los eventos pueden procesarse varias veces, porque muchos sistemas o microservicios pueden estar interesados en el evento.

Además, en caso de que el comando no sea idempotente, es importante que sólo se procese una vez. Un comando es idempotente si se puede ejecutar varias veces sin cambiar el resultado, ya sea por la naturaleza del comando o por la forma en que el sistema lo maneja.

Es una buena práctica hacer que sus comandos y actualizaciones sean idempotentes, cuando tenga sentido bajo las reglas en invariantes del negocio en su dominio. Por ejemplo, y para reutilizar el mismo anterior, si por alguna razón (reintento de lógica, intentos de piratería, etc.) el mismo comando **`CreateOrderCommand`** llega a su sistema varias veces, debe poder identificarlo y asegurarse de no crear varias órdenes. Para hacerlo, debe adjuntar algún tipo de identidad en las operaciones e identificar si el comando o la actualización ya se procesaron.

Un comando se envía a un solo receptor, los comandos no se publican. La publicación es para eventos que establecen un hecho: que algo ha sucedido y podría ser interesante para los manejadores de

eventos. En el caso de los eventos, a quien los publica no le interesa quiénes son los receptores ni lo que hacen con ellos. Pero los eventos del dominio o de integración son una historia diferente ya presentada en secciones anteriores.

Un comando se implementa con una clase que contiene campos de datos o colecciones, con toda la información necesaria para ejecutarlo. Un comando es un tipo especial de *Data Transfer Object* (DTO), que se usa específicamente para solicitar cambios o transacciones. El comando en sí se basa exactamente en la información necesaria para procesarlo y nada más.

El siguiente ejemplo muestra la clase **`CreateOrderCommand`** simplificada. Este es un comando inmutable que se utiliza en el microservicio de pedido en eShopOnContainers.

```
// DDD and CQRS patterns comment
// Note that it is recommended that you implement immutable commands
// In this case, immutability is achieved by having all the setters as private
// plus being able to update the data just once, when creating the object
// through the constructor.

// References on immutable commands:
// http://cqrs.nu/Faq
// https://docs.spine3.org/motivation/immutability.html
// http://blog.gauffin.org/2012/06/griffin-container-introducing-command-support/
// https://msdn.microsoft.com/library/bb383979.aspx

[DataContract]
public class CreateOrderCommand : IRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;
    [DataMember]
    public string City { get; private set; }
    [DataMember]
    public string Street { get; private set; }
    [DataMember]
    public string State { get; private set; }
    [DataMember]
    public string Country { get; private set; }
    [DataMember]
    public string ZipCode { get; private set; }
    [DataMember]
    public string CardNumber { get; private set; }
    [DataMember]
    public string CardHolderName { get; private set; }
    [DataMember]
    public DateTime CardExpiration { get; private set; }
    [DataMember]
    public string CardSecurityNumber { get; private set; }
    [DataMember]
    public int CardTypeId { get; private set; }
    [DataMember]
    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;

    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }
```

```
    public CreateOrderCommand(List<BasketItem> basketItems, string city,
        string street,
        string state, string country, string zipcode,
        string cardNumber, string cardHolderName, DateTime cardExpiration,
        string cardSecurityNumber, int cardTypeId) : this()
    {
        _orderItems = MapToOrderItems(basketItems);
        City = city;
        Street = street;
        State = state;
        Country = country;
        ZipCode = zipcode;
        CardNumber = cardNumber;
        CardHolderName = cardHolderName;
        CardSecurityNumber = cardSecurityNumber;
        CardTypeId = cardTypeId;
        CardExpiration = cardExpiration;
    }
    public class OrderItemDTO
    {
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public decimal Discount { get; set; }
        public int Units { get; set; }
        public string PictureUrl { get; set; }
    }
}
```

Básicamente, la clase de comando contiene todos los datos necesarios para realizar una transacción del negocio, usando los objetos del modelo de dominio. Por lo tanto, los comandos son simplemente estructuras de datos que contienen datos de sólo lectura y ningún comportamiento. El nombre del comando indica su propósito. En muchos lenguajes como C#, los comandos se representan como clases, pero no son clases verdaderas en el sentido real de orientado a objetos.

Como característica adicional, los comandos son inmutables, porque el uso esperado es que sean procesados directamente por el modelo de dominio. No necesitan cambiar durante su vida proyectada. En una clase C#, la inmutabilidad se puede lograr al no tener ningún setter u otros métodos que cambien el estado interno.

Por ejemplo, la clase de comando para crear un pedido probablemente sea similar en términos de datos de pedido que desea crear, pero probablemente no necesite los mismos atributos. Por ejemplo, **`CreateOrderCommand`** no tiene una ID de pedido, porque la orden aún no se ha creado.

Muchas clases de comando pueden ser simples, requiriendo sólo unos pocos campos sobre algún estado que necesita ser cambiado. Ese sería el caso si sólo está cambiando el estado de un pedido de "en proceso" a "pagado" o "enviado" utilizando un comando similar al siguiente:

```
[DataContract]
public class UpdateOrderStatusCommand
    :IAsyncRequest<bool>
{
    [DataMember]
    public string Status { get; private set; }
    [DataMember]
    public string OrderId { get; private set; }
    [DataMember]
    public string BuyerIdentityGuid { get; private set; }
}
```

Algunos desarrolladores hacen que sus objetos de interfaz de usuario por *petición* sean distintos de sus DTO de comando, pero eso es sólo una cuestión de preferencia. Es una separación tediosa, con poco valor agregado y los objetos son casi exactamente iguales. Por ejemplo, en eShopOnContainers, algunos comandos vienen directamente del lado del cliente.

La clase Manejador de Comando

Debería implementar una clase específica de manejador de comandos para cada comando. Así es como funciona el patrón y es donde usará el objeto del comando, los objetos de dominio y los objetos del repositorio de infraestructura. El manejador de comandos es, de hecho, el corazón de la capa de aplicación en términos de CQRS y DDD. Sin embargo, toda la lógica de dominio debe estar contenida dentro de las clases de dominio, dentro de las raíces de agregación (entidades raíz), entidades secundarias o servicios de dominio, pero no dentro del manejador de comandos, que es una clase de la capa de aplicación.

Un manejador de comando recibe un comando y obtiene un resultado del agregado que se utiliza. El resultado debería ser una ejecución exitosa del comando o una excepción. En el caso de una excepción, el estado del sistema no debe modificarse.

El manejador de comandos generalmente sigue los siguientes pasos:

- Recibe el objeto de comando, como un DTO (del mediador u otro objeto de infraestructura).
- Valida que el comando sea válido (si no es validado por el mediador).
- Instancia la raíz de agregación destinataria del comando actual.
- Ejecuta el método en la instancia de raíz de agregación, obteniendo los datos requeridos del comando.
- Persiste el nuevo estado del agregado en su base de datos. Esta última operación es la transacción real.

Normalmente, un manejador de comandos trata con un agregado único, controlado por su raíz de agregación (entidad raíz). Si múltiples agregados se vean afectados por la recepción de un solo comando, podría usar eventos de dominio para propagar estados o acciones a través de múltiples agregados.

El punto importante aquí es que cuando se procesa un comando, toda la lógica de dominio debe estar dentro del modelo de dominio (los agregados), completamente encapsulada y lista para las pruebas unitarias. El manejador de comandos solo actúa como un coordinador, para obtener el modelo de dominio de la base de datos y, como último paso, indicar a la capa de infraestructura (repositorios) que persista en los cambios cuando se modifique el modelo. La ventaja de este enfoque es que puede refactorizar la lógica de dominio en un modelo de dominio de comportamiento aislado, completamente encapsulado, rico y expresivo, sin tener que cambiar el código en las capas de aplicación o infraestructura, que son el nivel de fontanería (manejadores de comandos, API web, repositorios, etc.).

Cuando los manejadores de comandos se vuelven complejos, con demasiada lógica, eso puede ser un *code smell*. Revíselos y, si encuentra lógica de dominio, refactorice el código para mover ese comportamiento a los métodos de los objetos del dominio (la entidad raíz y las secundarias).

Como ejemplo de una clase de manejador de comandos, el siguiente código muestra la misma clase **CreateOrderCommandHandler** que vimos al principio de este capítulo. En este caso, queremos resaltar el método **Handle** y las operaciones con los objetos/agregados del modelo de dominio.

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                     IOrderRepository orderRepository,
                                     IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
                           throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
                           throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
                                throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {

        // Create the Order AggregateRoot
        // Add child entities and value objects through the Order aggregate root
        // methods and constructor so validations, invariants, and business logic
        // make sure that consistency is preserved across the whole aggregate

        var address = new Address(message.Street, message.City, message.State,
                                  message.Country, message.ZipCode);
        var order = new Order(message.UserId, address, message.CardTypeId,
                              message.CardNumber, message.CardSecurityNumber,
                              message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                               item.Discount, item.PictureUrl, item.Units);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}
```

Estos son pasos adicionales que ocurren en un manejador de comandos:

- Use los datos del comando para operar con los métodos y el comportamiento de la raíz de agregación.
- Internamente, dentro de los objetos de dominio, se generan eventos mientras se ejecuta la transacción, pero eso es transparente desde el punto de vista de un manejador de comandos.

- Si el resultado de la operación del agregado es exitoso y después de que finaliza la transacción, dispare los eventos de integración. (Estos también pueden ser disparados por clases de infraestructura como repositorios).

Recursos adicionales

- **Mark Seemann. At the Boundaries, Applications are Not Object-Oriented**
  http://blog.ploeh.dk/2011/05/31/AttheBoundaries,ApplicationsareNotObject-Oriented/
- **Commands and events**
  http://cqrs.nu/Faq/commands-and-events
- **What does a command handler do?**
  http://cqrs.nu/Faq/command-handlers
- **Jimmy Bogard. Domain Command Patterns – Handlers**
  https://jimmybogard.com/domain-command-patterns-handlers/
- **Jimmy Bogard. Domain Command Patterns – Validation**
  https://jimmybogard.com/domain-command-patterns-validation/

La línea de procesamiento de comandos: como activar un manejador de comandos

La siguiente pregunta es cómo invocar un manejador de comandos. Puede llamarlo manualmente desde cada controlador ASP.NET Core relacionado. Sin embargo, ese enfoque estaría demasiado acoplado y no es lo ideal.

Las otras dos opciones principales, que son las recomendadas, son:

- A través de un artefacto del patrón Mediador en memoria.
- Con una cola de mensajes asíncronos, entre controladores y manejadores.

Usando el patrón Mediador en la línea de procesamiento de comandos (en la memoria)

Como se muestra en la figura 7-25, en un enfoque CQRS se usa un mediador inteligente, similar a un bus en memoria, que es lo suficientemente listo como para redirigir al manejador de comandos correcto, según el tipo de comando o DTO que se recibe. Las flechas unidireccionales representan las dependencias entre los objetos (en muchos casos, inyectados a través de DI) con sus interacciones relacionadas.

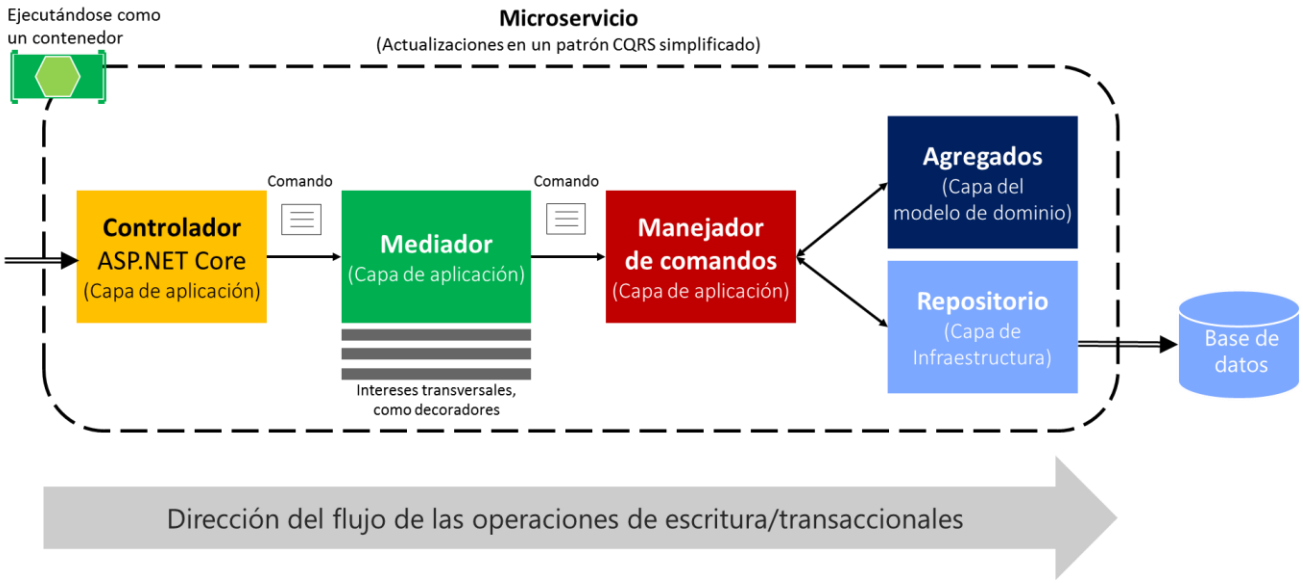


Figura 7-25. *Usando el patrón Mediador en un microservicio CQRS*

La razón por la que tiene sentido usar el patrón Mediador es que, en las aplicaciones empresariales, las solicitudes de procesamiento pueden complicarse. Desea poder manejar una cantidad indeterminada de intereses transversales como registro, validaciones, auditoría y seguridad. En estos casos, puede confiar en una línea de proceso de mediadores (consulte el patrón Mediador) para proporcionar una forma de manejar esos comportamientos adicionales o intereses transversales.

Un mediador es un objeto que encapsula el "cómo" de este proceso: coordina la ejecución en función del estado, la forma como se invoca el manejador de comandos o la data que proporciona al manejador. Con un componente mediador puede manejar intereses transversales de forma centralizada y transparente mediante la aplicación de decoradores (o comportamientos del *pipeline*, a partir de MediatR v3). (Para más información, vea el patrón Decorador).

Los decoradores y los comportamientos son similares a la Programación Orientada a Aspectos (AOP), sólo se aplica a una línea de proceso específica, administrada por el componente mediador. Los aspectos de AOP, que manejan intereses transversales, se aplican en función de los *aspect weavers* (“tejedores de aspectos”), inyectados en el tiempo de compilación o en base a la intercepción de llamadas a objetos. A veces se dice que los enfoques típicos de AOP funcionan "como magia", porque no es fácil ver cómo AOP hace su trabajo. Cuando se trata de problemas graves o errores, AOP puede ser difícil de depurar. Por otro lado, estos decoradores/comportamientos son explícitos y se aplican sólo en el contexto del mediador, por lo que la depuración es mucho más predecible y fácil.

Por ejemplo, en el servicio de microservicio de eShopOnContainers, implementamos dos comportamientos de ejemplo, una clase LogBehavior y una clase ValidatorBehavior. La implementación de los comportamientos se explica en la siguiente sección mostrando cómo eShopOnContainers usa los comportamientos de MediatR 3.

Usando colas de mensajes (procesos independientes) en la línea de proceso de comandos

Otra opción es usar mensajes asíncronos basados en intermediarios o colas de mensajes, como se muestra en la figura 7-26. Esa opción también se puede combinar con el componente mediador, justo antes del controlador de comandos.

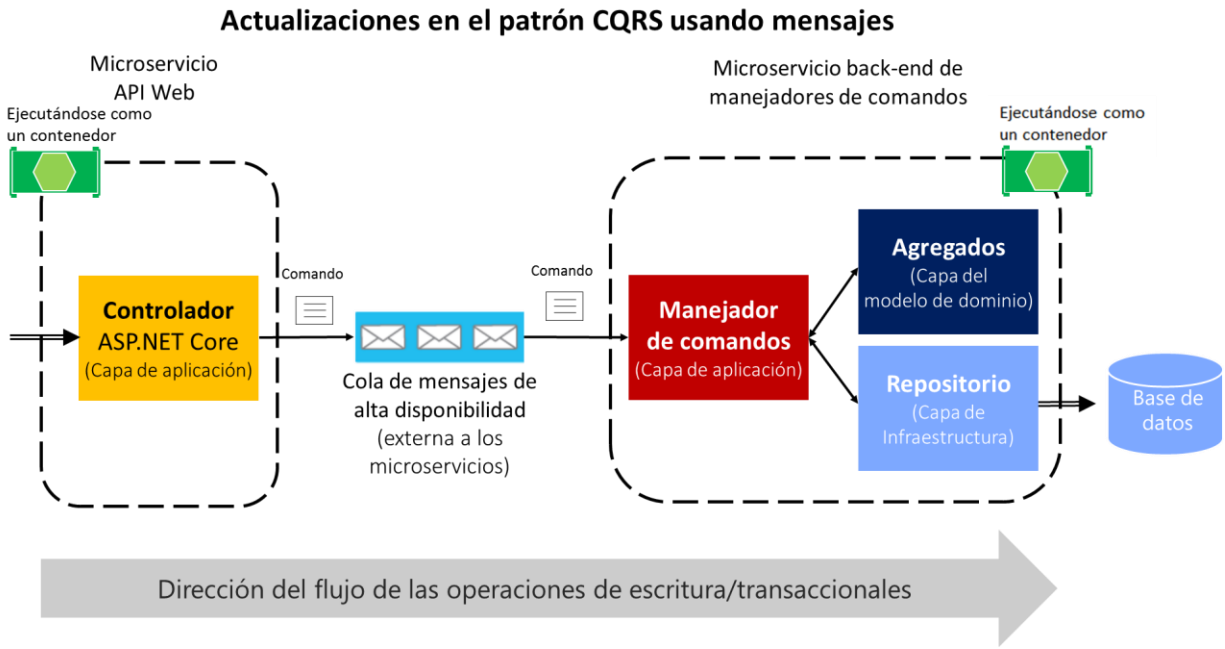


***Figura 7-26**. Usando colas de mensajes (procesos independientes) con comandos CQRS*

El uso de colas de mensajes para aceptar los comandos puede complicar aún más la línea de procesamiento de los comandos, ya que probablemente deba dividirla en dos procesos conectados a través de una cola de mensajes externos. Aun así, debe usarlas si necesita tener una mayor escalabilidad y rendimiento, basados en la mensajería asíncrona. Tenga en cuenta que en el caso de Figura 7-26, el controlador sólo envía el mensaje de comando a la cola y se regresa. Luego, los manejadores de comandos procesan los mensajes a su propio ritmo. Esa es una gran ventaja de las colas: la cola de mensajes puede actuar como un búfer en los casos en los que se necesita una "hiperescalabilidad", como en el caso de las transacciones en los mercados de valores o cualquier otro escenario con un gran volumen de datos de entrada.

Sin embargo, debido a la naturaleza asíncrona de las colas de mensajes, debe resolver cómo comunicarse con la aplicación cliente sobre el éxito o el fracaso del proceso del comando. Como regla general, nunca debe usar comandos de "disparar y olvidar". Cada aplicación del negocio necesita saber si un comando fue procesado con éxito o, al menos, validado y aceptado.

Por lo tanto, ser capaz de responder al cliente después de validar un mensaje de comando que se envió a una cola asíncrona, esto agrega complejidad al sistema, en comparación con un proceso del comando en memoria, que devuelve el resultado después de ejecutar la transacción. Con las colas, es posible que deba devolver el resultado del proceso a través de otros mensajes de resultados de otras operaciones, lo que requerirá componentes adicionales y comunicación personalizada en su sistema.

Además, los comandos asíncronos son unidireccionales, que en muchos casos podrían no ser necesarios, como se explica en el siguiente intercambio interesante entre Burtsev Alexey y Greg Young en una conversación en línea:

> *[Burtsev Alexey] He encontrado mucho código donde la gente usa el manejo de comandos asíncronos o mensajes en una dirección, sin ninguna razón para hacerlo (no están haciendo una operación larga, no están ejecutando código asíncrono externo, ni siquiera cruzan el límite de la aplicación para estar usando el bus de mensajes). ¿Por qué introducen esta complejidad innecesaria? Y, de hecho, hasta ahora no he visto un ejemplo de código CQRS con manejadores de comandos bloqueadores, aunque debería funcionar bien en la mayoría de los casos.*
>
> *[Greg Young] [...] no existe un comando asíncrono; en realidad es otro evento. Si debo aceptar lo que me envías y disparar un evento si no estoy de acuerdo, ya no me estás diciendo que haga algo* [es decir, ya no es un comando]. *Eres tú diciéndome que algo se ha hecho. Esto parece una ligera diferencia al principio, pero tiene muchas implicaciones.*

Los comandos asíncronos aumentan en gran medida la complejidad de un sistema, porque no hay una forma simple de indicar fallas. Por lo tanto, no se recomiendan los comandos asíncronos excepto cuando hay requisitos de escalabilidad o en casos especiales, cuando se comunican los microservicios internos a través de mensajes. En esos casos, debe diseñar un sistema de informe y recuperación separado para las fallas.

En la versión inicial de eShopOnContainers, decidimos utilizar un procesamiento de comandos sincrónico, iniciado a partir de *peticiones* HTTP y manejado con el patrón Mediador. Eso le permite fácilmente devolver el éxito o el fracaso del proceso, como en la implementación de `CreateOrderCommandHandler`.

En cualquier caso, esta debe ser una decisión basada en los requisitos del negocio de su aplicación o microservicio.

Implementando la línea de procesamiento de comandos con el patrón mediador (MediatR)

Como una implementación de ejemplo, esta guía propone usar el *pipeline* en memoria (en el mismo proceso), basado en el patrón de Mediador para gestionar la recepción de comandos y enrutarlos a los manejadores de comandos adecuados. La guía también propone aplicar comportamientos para atender los intereses transversales.

Para la implementación en .NET Core, hay múltiples librerías *open source* disponibles que implementan el patrón Mediador. La librería utilizada en esta guía es MediatR, que es open source y fue creada por Jimmy Bogard, pero podría usar otro enfoque. MediatR es una librería pequeña y simple que le permite procesar en memoria mensajes como los comandos y permite aplicar decoradores o comportamientos.

El uso del patrón Mediador le ayuda a reducir el acoplamiento y a aislar los otros intereses del trabajo solicitado, mientras se conecta automáticamente al manejador que realiza ese trabajo, en este caso, a los manejadores de comandos.

Otra buena razón para utilizar el patrón Mediador fue explicada por Jimmy Bogard al revisar esta guía:

Creo que vale la pena mencionar las pruebas aquí: proporciona una ventana buena y consistente al comportamiento de su sistema. Petición de entrada, respuesta de salida. Hemos encontrado ese aspecto bastante valioso en la construcción de pruebas de comportamiento consistentes.

En primer lugar, echemos un vistazo a un código de controlador API Web de ejemplo, donde realmente usaría el objeto mediador. Si no estuviera usando el objeto mediador, necesitaría inyectar todas las dependencias para ese controlador, cosas como un objeto **logger** y otros. Por lo tanto, el constructor sería bastante complicado. Por otro lado, si usa el objeto mediador, el constructor puede ser mucho más simple, con sólo unas pocas dependencias en lugar de las muchas que tendría si tuviera una por operación transversal, como puede apreciar en el siguiente ejemplo:

```
public class MyMicroserviceController : Controller
{
    public MyMicroserviceController(IMediator mediator,
                                    IMyMicroserviceQueries microserviceQueries)
    // ...
```

Puede ver que el mediador proporciona un constructor de controlador API Web limpio y delgado. Además, dentro de los métodos del controlador, el código para enviar un comando al mediador es casi una línea:

```
[Route("new")]
[HttpPost]
public async Task<IActionResult> ExecuteBusinessOperation([FromBody]RunOpCommand
                                                          runOperationCommand)
{
    var commandResult = await _mediator.SendAsync(runOperationCommand);

    return commandResult ? (IActionResult)Ok() : (IActionResult)BadRequest();
}
```

Implementando comandos idempotentes

En eShopOnContainers, un ejemplo más avanzado que el anterior, sería cuando se presente un objeto **CreateOrderCommand** desde el microservicio de pedido. Pero, dado que el proceso del negocio de pedidos es un poco más complejo y, en nuestro caso, realmente comienza en el servicio del carrito de compras, esta acción de enviar el objeto **CreateOrderCommand** se realiza desde un manejador de eventos de integración llamado **UserCheckoutAcceptedIntegrationEvent.cs** en lugar de un controlador API Web simple, llamado desde la aplicación cliente como en el ejemplo anterior más sencillo.

Sin embargo, la acción de enviar el comando a MediatR es bastante similar, como se muestra en el siguiente código.

```
var createOrderCommand = new CreateOrderCommand(eventMsg.Basket.Items,
                                                 eventMsg.UserId, eventMsg.City,
                                                 eventMsg.Street, eventMsg.State,
                                                 eventMsg.Country, eventMsg.ZipCode,
                                                 eventMsg.CardNumber,
                                                 eventMsg.CardHolderName,
                                                 eventMsg.CardExpiration,
                                                 eventMsg.CardSecurityNumber,
                                                 eventMsg.CardTypeId);

var requestCreateOrder = new IdentifiedCommand<CreateOrderCommand,bool>(
                                                                 createOrderCommand,
                                                                 eventMsg.RequestId);

result = await _mediator.Send(requestCreateOrder);
```

Sin embargo, este caso también es un poco más avanzado porque también estamos implementando comandos idempotentes. El proceso **CreateOrderCommand** debe ser idempotente, por lo que, en caso de que se repita el mismo mensaje a través de la red (por cualquier motivo, como reintentos), el pedido sólo se debe procesar una vez.

Esto se implementa al empaquetar el comando del negocio (en este caso, **CreateOrderCommand**) e insertarlo en un **IdentifiedCommand** genérico, que se rastrea mediante un ID de cada mensaje que llega a través de la red, que debe ser idempotente.

En el siguiente código, puede ver que **IdentifiedCommand** no es más que un DTO con ID más el objeto de comando empresarial.

```
public class IdentifiedCommand<T, R> : IRequest<R>
    where T : IRequest<R>
{
    public T Command { get; }
    public Guid Id { get; }

    public IdentifiedCommand(T command, Guid id)
    {
        Command = command;
        Id = id;
    }
}
```

Entonces, el **CommandHandler** para **IdentifiedCommand**, llamado IdentifiedCommandHandler.cs básicamente verificará si la identificación que viene como parte del mensaje ya existe en una tabla. Si ya existe, ese comando no se procesará nuevamente, por lo que se comporta como un comando idempotente. Ese código de infraestructura se realiza mediante la llamada al método **_requestManager.ExistAsync()** a continuación.

```
// IdentifiedCommandHandler.cs

public class IdentifiedCommandHandler<T, R> :
                                    IAsyncRequestHandler<IdentifiedCommand<T, R>, R>
                                    where T : IRequest<R>
{
    private readonly IMediator _mediator;
    private readonly IRequestManager _requestManager;

    public IdentifiedCommandHandler(IMediator mediator,
                                     IRequestManager requestManager)
    {
        _mediator = mediator;
        _requestManager = requestManager;
    }

    protected virtual R CreateResultForDuplicateRequest()
    {
        return default(R);
    }

    public async Task<R> Handle(IdentifiedCommand<T, R> message)
    {
        var alreadyExists = await _requestManager.ExistAsync(message.Id);
        if (alreadyExists)
        {
            return CreateResultForDuplicateRequest();
        }
        else
        {
            await _requestManager.CreateRequestForCommandAsync<T>(message.Id);

            // Send the embeded business command to mediator
            // so it runs its related CommandHandler
            var result = await _mediator.Send(message.Command);

            return result;
        }
    }

}
```

Dado que **IdentifiedCommand** actúa como el envoltorio (*wrapper*) de un comando del negocio, cuando se necesita procesar el comando y el Id no está repetido, toma ese comando interno y lo reenvía a MediatR, como en la última parte del código que se muestra arriba al ejecutar **_mediator.Send (message.Command)**, desde IdentifiedCommandHandler.cs.

Al hacer eso, se vinculará y ejecutará el manejador de comandos del negocio, en este caso, `CreateOrderCommandHandler` que está ejecutando transacciones contra la base de datos de pedidos, como se muestra en el siguiente código.

```
// CreateOrderCommandHandler.cs

public class CreateOrderCommandHandler
                                    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
                                     IOrderRepository orderRepository,
                                     IIdentityService identityService)
    {
        _orderRepository = orderRepository ??
                           throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ??
                           throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ??
                                throw new ArgumentNullException(nameof(mediator));
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        // Add/Update the Buyer AggregateRoot
        var address = new Address(message.Street, message.City, message.State,
                                  message.Country, message.ZipCode);
        var order = new Order(message.UserId, address, message.CardTypeId,
                              message.CardNumber, message.CardSecurityNumber,
                              message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                               item.Discount, item.PictureUrl, item.Units);
        }

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}
```

Registrando los tipos usados por MediatR

Para que MediatR tenga conocimiento de las clases manejadoras de comando, necesita registrar las clases del mediador y las de los manejadores en su contenedor IoC. Por defecto, MediatR utiliza Autofac como el contenedor IoC, pero también puede usar el contenedor ASP.NET Core IoC incorporado o cualquier otro contenedor compatible con MediatR.

El siguiente código muestra cómo registrar los tipos y comandos de MediatR cuando se utilizan módulos Autofac.

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IAsyncRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(
                               typeof(CreateOrderCommand).GetTypeInfo().Assembly).
                                    AsClosedTypesOf(typeof(IAsyncRequestHandler<,>));

        // Other types registration
        //...

    }
}
```

Aquí es donde "sucede la magia" con MediatR.

Como cada controlador de comandos implementa la interfaz genérica **IAsyncRequestHandler<T>**, al registrar los ensamblados, el código registra con **RegisteredAssemblyTypes** todos los tipos asignados como **IAsyncRequestHandler** mientras relaciona los **CommandHandlers** con sus **Commands**, gracias a la relación establecida en la clase **CommandHandler**, como en el siguiente ejemplo:

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
```

Ese es el código que correlaciona los comandos con los manejadores de comando. El manejador es sólo una clase simple, pero hereda de **RequestHandler<T>**, donde **<T>** es el tipo del comando y MediatR se asegura de que se invoque con la carga útil correcta (el comando).

Atendiendo intereses transversales al procesar comandos con los *Behaviors* de MediatR

Hay una cosa más: ser capaz de aplicar los intereses transversales al *pipeline* de MediatR. También puede ver al final del código del módulo de registro Autofac, cómo se está registrando un tipo de comportamiento, específicamente, una clase **LoggingBehavior** y una clase **ValidatorBehavior**. Pero también se podrían agregar otros comportamientos personalizados.

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IAsyncRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(
                               typeof(CreateOrderCommand).GetTypeInfo().Assembly).
                                    AsClosedTypesOf(typeof(IAsyncRequestHandler<,>));

        // Other types registration
        //...

        builder.RegisterGeneric(typeof(LoggingBehavior<,>)).
                                                    As(typeof(IPipelineBehavior<,>));

        builder.RegisterGeneric(typeof(ValidatorBehavior<,>)).
                                                    As(typeof(IPipelineBehavior<,>));

    }
}
```

Esa clase LoggingBehavior se puede implementar como el siguiente código, que registra información sobre el controlador de comandos que se está ejecutando y si fue exitoso o no.

```
public class LoggingBehavior<TRequest, TResponse>
         : IPipelineBehavior<TRequest, TResponse>
{
    private readonly ILogger<LoggingBehavior<TRequest, TResponse>> _logger;
    public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> logger) =>
                                                                  _logger = logger;

    public async Task<TResponse> Handle(TRequest request,
                                        RequestHandlerDelegate<TResponse> next)
    {
        _logger.LogInformation($"Handling {typeof(TRequest).Name}");
        var response = await next();
        _logger.LogInformation($"Handled {typeof(TResponse).Name}");
        return response;
    }
}
```

Simplemente implementando esta clase de decorador y decorando el *pipeline* con ella, todos los comandos procesados a través de MediatR registrarán información sobre la ejecución.

El microservicio de pedidos de eShopOnContainers también aplica un segundo comportamiento para las validaciones básicas, la clase ValidatorBehavior que se basa en la librería FluentValidation, como se muestra en el siguiente código:

```
public class ValidatorBehavior<TRequest, TResponse>
         : IPipelineBehavior<TRequest, TResponse>
{
    private readonly IValidator<TRequest>[] _validators;
    public ValidatorBehavior(IValidator<TRequest>[] validators) =>
                                                          _validators = validators;

    public async Task<TResponse> Handle(TRequest request,
                                        RequestHandlerDelegate<TResponse> next)
    {
        var failures = _validators
            .Select(v => v.Validate(request))
            .SelectMany(result => result.Errors)
            .Where(error => error != null)
            .ToList();

        if (failures.Any())
        {
            throw new OrderingDomainException(
                $"Command Validation Errors for type {typeof(TRequest).Name}",
                        new ValidationException("Validation exception", failures));
        }

        var response = await next();
        return response;
    }
}
```

Luego, en base a la librería FluentValidation, creamos la validación para los datos pasados con **CreateOrderCommand**, como en el siguiente código:

```
public class CreateOrderCommandValidator : AbstractValidator<CreateOrderCommand>
{
    public CreateOrderCommandValidator()
    {
        RuleFor(command => command.City).NotEmpty();
        RuleFor(command => command.Street).NotEmpty();
        RuleFor(command => command.State).NotEmpty();
        RuleFor(command => command.Country).NotEmpty();
        RuleFor(command => command.ZipCode).NotEmpty();
        RuleFor(command => command.CardNumber).NotEmpty().Length(12, 19);
        RuleFor(command => command.CardHolderName).NotEmpty();
        RuleFor(command =>
command.CardExpiration).NotEmpty().Must(BeValidExpirationDate).WithMessage("Please
specify a valid card expiration date");
        RuleFor(command => command.CardSecurityNumber).NotEmpty().Length(3);
        RuleFor(command => command.CardTypeId).NotEmpty();
        RuleFor(command =>
command.OrderItems).Must(ContainOrderItems).WithMessage("No order items found");
    }

    private bool BeValidExpirationDate(DateTime dateTime)
    {
        return dateTime >= DateTime.UtcNow;
    }

    private bool ContainOrderItems(IEnumerable<OrderItemDTO> orderItems)
    {
        return orderItems.Any();
    }
}
```

Podría crear validaciones adicionales. Esta es una forma muy limpia y elegante de implementar sus validaciones de comandos.

De forma similar, podría implementar otros comportamientos para aspectos adicionales o intereses transversales que quiera aplicar a los comandos cuando los procese.

Recursos adicionales

El patrón Mediador

- **Mediator pattern**
  https://en.wikipedia.org/wiki/Mediator_pattern

El patrón Decorador

- **Decorator pattern**
  https://en.wikipedia.org/wiki/Decorator_pattern

MediatR (Jimmy Bogard)

- **MediatR.** GitHub repo.
  https://github.com/jbogard/MediatR
- **CQRS with MediatR and AutoMapper**
  https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/
- **Put your controllers on a diet: POSTs and commands.**
  https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/
- **Tackling cross-cutting concerns with a mediator pipeline**
  https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/
- **CQRS and REST: the perfect match**
  https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/
- **MediatR Pipeline Examples**
  https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/
- **Vertical Slice Test Fixtures for MediatR and ASP.NET Core**
  https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/
- **MediatR Extensions for Microsoft Dependency Injection Released**
  https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/

Fluent validation

- **Jeremy Skinner. FluentValidation.** GitHub repo.
  https://github.com/JeremySkinner/FluentValidation

Implementando Aplicaciones Resilientes

Visión

Sus aplicaciones basadas en microservicios y en la nube deben manejar las fallas parciales que con certeza ocurrirán eventualmente. Debe diseñar su aplicación para que sea resiliente a esas fallas parciales.

La resiliencia es la capacidad de recuperarse de las fallas y continuar funcionando. No se trata de evitar fallas, sino de aceptar el hecho de que las fallas ocurrirán y responder a ellas de una manera que evite la pérdida del servicio o la pérdida de datos. El objetivo de la resiliencia es devolver la aplicación a un estado completamente funcional después de una falla.

Es todo un desafío diseñar y desplegar una aplicación basada en microservicios. Pero también necesita mantener su aplicación ejecutándose en un entorno donde es seguro que habrá algún tipo de falla. Por lo tanto, su aplicación debe ser resiliente. Debe estar diseñada para hacer frente a fallas parciales, como interrupciones de la red, o nodos o máquinas virtuales que se cuelgan en la nube. Incluso los microservicios (contenedores) que se mueven a un nodo diferente dentro de un *cluster* pueden causar fallas cortas e intermitentes dentro de la aplicación.

Los muchos componentes individuales de su aplicación también deben incorporar características de monitorización de la salud. Al seguir las pautas de este capítulo, puede crear una aplicación que funcione sin problemas a pesar del tiempo transitorio sin servicio, o las interrupciones normales que ocurren en las implementaciones complejas y basadas en la nube.

Manejando fallas parciales

En sistemas distribuidos como aplicaciones basadas en microservicios, existe un riesgo constante de falla parcial. Por ejemplo, un microservicio/contenedor puede fallar o no estar disponible por un corto tiempo o una VM o servidor puede fallar. Como los clientes y los servicios son procesos separados, es posible que un servicio no pueda responder de manera oportuna a la solicitud de un cliente. Es posible que el servicio esté sobrecargado y responda con extrema lentitud a las solicitudes, o que simplemente no sea accesible durante un breve período de tiempo debido a problemas de red.

Por ejemplo, considere la página de detalles del pedido de la aplicación de ejemplo eShopOnContainers. Si el microservicio de pedido no responde cuando el usuario intenta enviar un pedido, una mala implementación del proceso del cliente (la aplicación web MVC), por ejemplo, si el código del cliente utilizara RPC síncronas sin tiempo de espera, bloquearía los hilos indefinidamente esperando por una respuesta. Además de crear una mala experiencia de usuario, cada espera que no

responde consume o bloquea un hilo, y los hilos son extremadamente valiosos en aplicaciones altamente escalables. Si hay muchos subprocesos bloqueados, con el tiempo el motor de ejecución de la aplicación puede quedarse sin subprocesos. En ese caso, la aplicación puede dejar de responder globalmente en lugar de simplemente no responder parcialmente, como se muestra en la Figura 8-1.

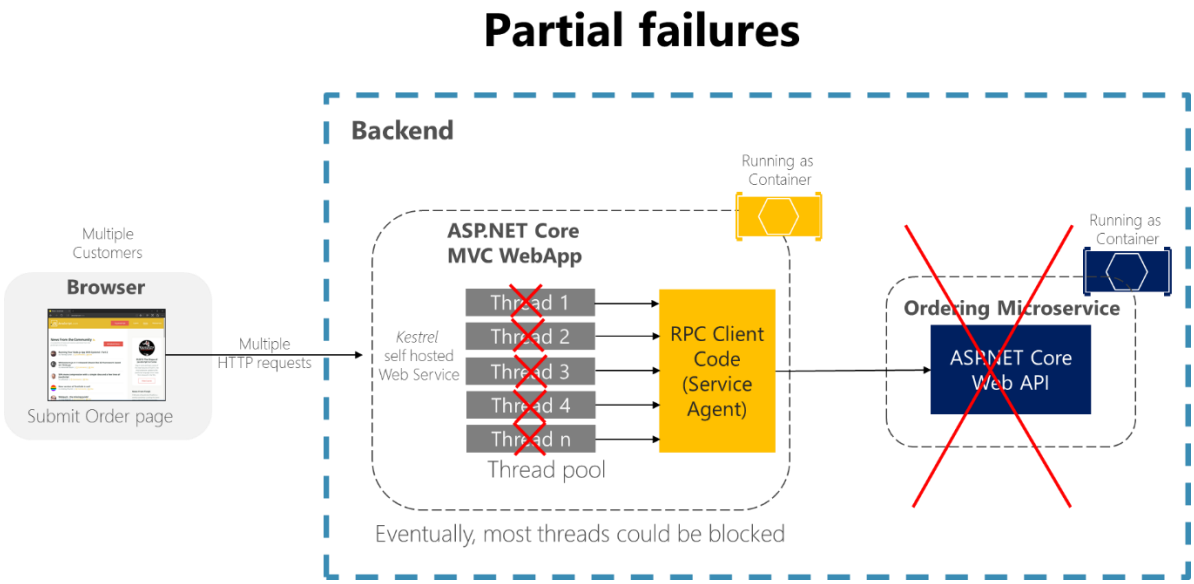


***Figura 8-1**. Fallas parciales por dependencias que afectan la disponibilidad de los hilos del servicio*

En una aplicación grande basada en microservicios, cualquier falla parcial puede amplificarse, especialmente si la mayor parte de la interacción interna de los microservicios se basa en llamadas HTTP síncronas (lo que se considera un anti patrón). Piense en un sistema que recibe millones de llamadas entrantes por día. Si su sistema tiene un diseño incorrecto que se basa en cadenas largas de llamadas HTTP síncronas, estas llamadas entrantes podrían resultar en muchos millones más de llamadas salientes (supongamos una proporción de 1: 4) a docenas de microservicios internos como dependencias síncronas. Esta situación se muestra en la Figura 8-2, especialmente en la dependencia #3.

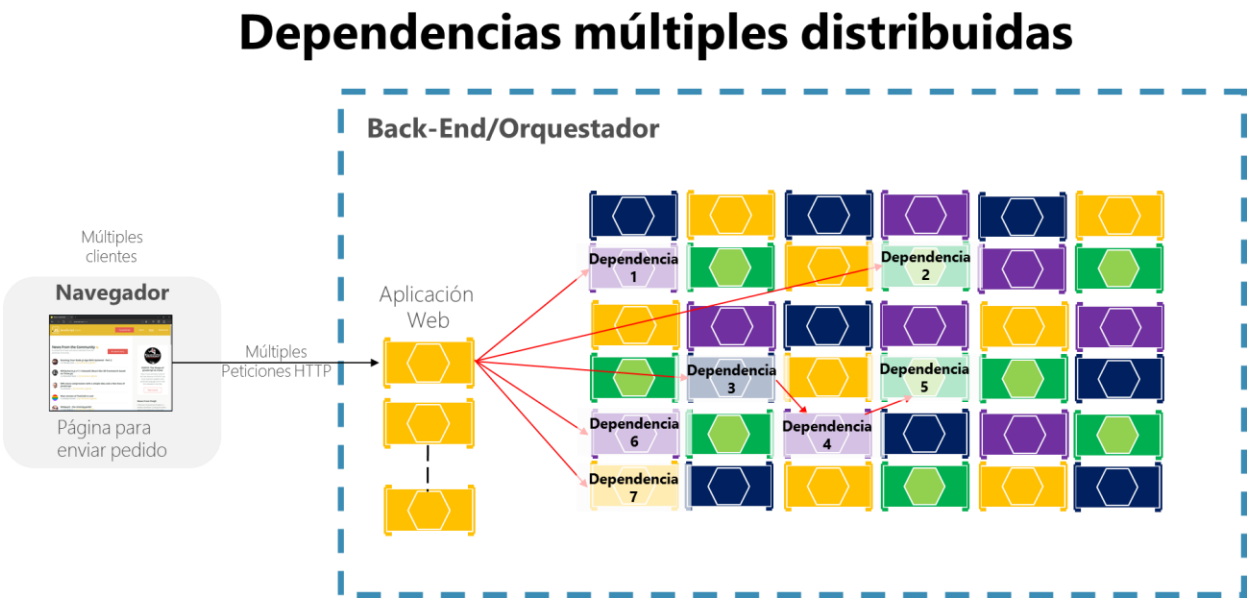


***Figura 8-2**. El impacto de un diseño incorrecto, con cadenas largas de peticiones HTTP*

La falla intermitente está prácticamente garantizada en un sistema distribuido y basado en la nube, incluso si cada dependencia en sí tiene una excelente disponibilidad. Esto es un hecho que debe tener en cuenta.

Si no diseña e implementa técnicas para garantizar la tolerancia a fallas, incluso los pequeños tiempos de inactividad pueden amplificarse. Como ejemplo, 50 dependencias cada una con 99.99% de disponibilidad resultaría en varias horas de inactividad cada mes debido a este efecto de onda expansiva. Cuando se produce un error en la dependencia de un microservicio al manejar un gran volumen de solicitudes, esa falla puede saturar rápidamente todos los hilos de solicitud disponibles en cada servicio y bloquear toda la aplicación.

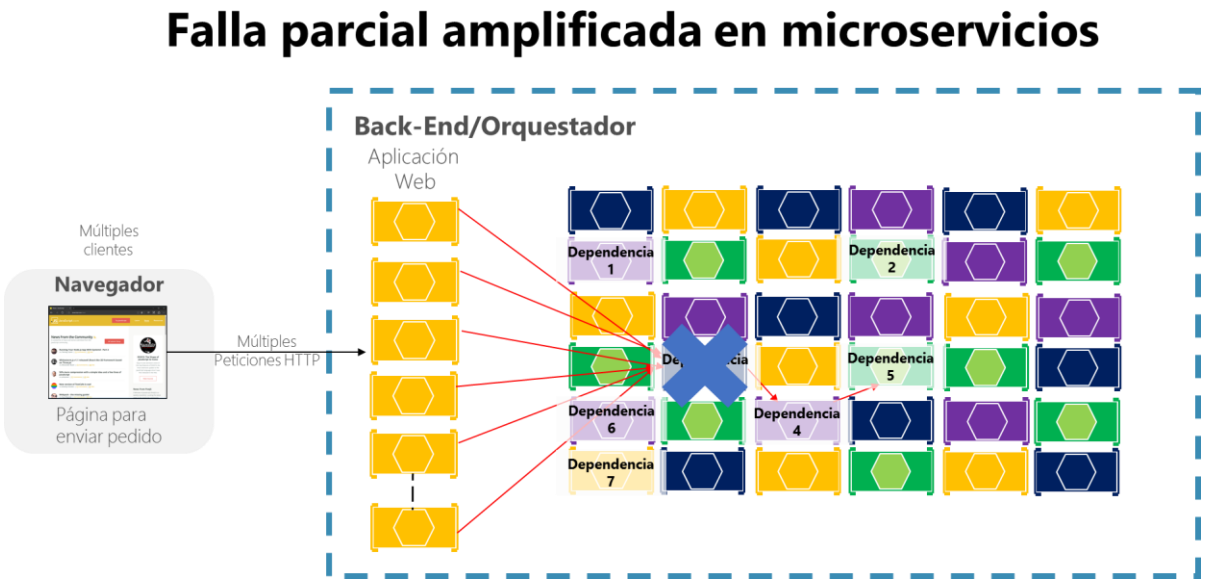


***Figura 8-3**. Falla parcial amplificada por microservicios con cadenas largas de llamadas HTTP síncronas*

Para minimizar este problema, en la sección "La integración asíncrona refuerza la autonomía de los microservicios" (en el capítulo de arquitectura), le alentamos a usar comunicación asíncrona entre los microservicios internos. Explicaremos más sobre esto, brevemente, en la siguiente sección.

Además, es esencial que diseñe sus microservicios y aplicaciones cliente para manejar fallas parciales, es decir, para construir microservicios y aplicaciones cliente resilientes.

Estrategias para manejar fallas parciales

Las estrategias para lidiar con fallas parciales incluyen lo siguiente.

Use comunicación asíncrona (por ejemplo, comunicación basada en mensajes) a través de microservicios internos. Es muy recomendable no encadenar llamadas HTTP síncronas en los microservicios internos, porque ese diseño incorrecto eventualmente se convertirá en la causa principal de fallas de servicio. Por el contrario, a excepción de las comunicaciones de *front-end* entre las aplicaciones cliente y el primer nivel de microservicios o *API Gateways* de grano fino, se recomienda usar sólo comunicación asíncrona (basada en mensajes) una vez pasado el ciclo de petición/respuesta inicial, entre los microservicios internos. La consistencia eventual y las arquitecturas basadas en eventos ayudarán a minimizar problemas por "efectos dominó". Estos enfoques refuerzan un nivel más alto de autonomía de los microservicios y, por lo tanto, evitan el problema que se menciona aquí.

Use reintentos con retroceso exponencial. Esta técnica ayuda a evitar fallas cortas e intermitentes al realizar reintentos de llamadas un cierto número de veces, en caso de que el servicio no esté disponible por período de tiempo corto. Esto puede ocurrir debido a problemas de red intermitentes o cuando un microservicio/contenedor se mueve a un nodo diferente en un *cluster*. Sin embargo, si estos reintentos no se diseñan correctamente con interruptores automáticos, pueden agravar los efectos dominantes, lo que en última instancia puede causar una denegación de servicio (DoS).

Busque paliativos a las fallas de la red. En general, los clientes deben estar diseñados para no bloquearse indefinidamente y poner siempre límites de tiempo mientras espera una respuesta. El uso de tiempos de espera garantiza que los recursos nunca estén inmovilizados indefinidamente.

Use el patrón de Interruptor Automático. En este enfoque, el proceso cliente rastrea la cantidad de solicitudes fallidas. Si la tasa de error excede un límite configurado, un "interruptor automático de circuito" se dispara para que los intentos posteriores fallen inmediatamente. (Si una gran cantidad de solicitudes están fallando, eso sugiere que el servicio no está disponible y que no tiene sentido hacer más peticiones). Después de un tiempo de espera, el cliente debe volver a intentarlo y, si las nuevas solicitudes son exitosas, desactivar el interruptor automático.

Proporcionar planes alternativos (fallbacks). En este enfoque, el proceso del cliente realiza una lógica alternativa cuando falla una solicitud, como devolver datos en caché o un valor predeterminado. Este es un enfoque adecuado para consultas, pero es más complejo para actualizaciones o comandos.

Limite el número de solicitudes en cola. Los clientes también deben imponer un límite superior a la cantidad de peticiones pendientes, que puede enviar a un servicio en particular. Si se ha alcanzado el límite, probablemente no tenga sentido realizar peticiones adicionales y esos intentos deberían fallar inmediatamente. En términos de implementación, la política de Bulkhead Isolation (asilamiento por mamparos, las divisiones internas de los barcos para contener los daños si se rompe el casco) Polly se

puede utilizar para cumplir este requisito. Este enfoque es esencialmente un regulador de paralelización con la clase SemaphoreSlim como implementación. También permite una "cola" fuera del mamparo (de protección). Puede eliminar de forma proactiva el exceso de carga incluso antes de la ejecución (por ejemplo, porque la capacidad se considere completa). Esto hace que su respuesta a ciertos escenarios de falla sea más rápida de lo que sería un interruptor automático, ya que éste espera a que ocurran las fallas. El objeto **`BulkheadPolicy`** en Polly expone cuán llenos están el mamparo y la cola y ofrece eventos en desbordamiento, por lo que también se puede usar para impulsar el escalado horizontal automático.

Recursos adicionales

- **Patrones de resiliencia**
  https://docs.microsoft.com/azure/architecture/patterns/category/resiliency
- **Adding Resilience and Optimizing Performance**
  https://msdn.microsoft.com/library/jj591574.aspx
- **Bulkhead.** GitHub repo. Implementation with Polly policy.
  https://github.com/App-vNext/Polly/wiki/Bulkhead
- **Diseño de aplicaciones resilientes de Azure**
  https://docs.microsoft.com/azure/architecture/resiliency/
- **Control de errores transitorios**
  https://docs.microsoft.com/azure/architecture/best-practices/transient-faults

Implementando reintentos con retroceso exponencial

Los reintentos con retroceso exponencial son una técnica que trata de reintentar una operación, con un tiempo de espera que aumenta exponencialmente, hasta que se alcanza un conteo máximo de reintentos (el retroceso exponencial). Esta técnica abarca el hecho de que los recursos de la nube pueden no estar disponibles intermitentemente por más de unos pocos segundos por cualquier motivo. Por ejemplo, un orquestador podría mover un contenedor a otro nodo en un *cluster* para balancear la carga. Durante ese tiempo, algunas solicitudes pueden fallar. Otro ejemplo podría ser una base de datos como SQL Azure, donde una base de datos se puede mover a otro servidor para balancear la carga, lo que hace que la base de datos no esté disponible durante unos segundos.

Hay muchos enfoques para implementar la lógica de reintentos con retroceso exponencial.

Implementar conexiones SQL resilientes en Entity Framework Core

Para Azure SQL DB, Entity Framework Core ya ofrece resiliencia interna de conexión de base de datos y lógica de reintento. Pero debe habilitar la estrategia de ejecución de Entity Framework para cada conexión DbContext si desea tener conexiones resilientes en EF Core.

Por ejemplo, el siguiente código en el nivel de conexión de EF Core permite conexiones SQL resilientes que se reintentan si la conexión falla.

```
// Startup.cs from any ASP.NET Core Web API
public class Startup
{
    // Other code ...
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
       // ...
       services.AddDbContext<OrderingContext>(options =>
          {
             options.UseSqlServer(Configuration["ConnectionString"],
                                  sqlServerOptionsAction: sqlOptions =>
                                  {
                                      sqlOptions.EnableRetryOnFailure(
                                          maxRetryCount: 5,
                                          maxRetryDelay: TimeSpan.FromSeconds(30),
                                          errorNumbersToAdd: null);
                                    });
          });
    }
//...
```

Estrategias de ejecución y transacciones explícitas usando BeginTransaction y múltiples DbContexts

Cuando los reintentos están habilitados en las conexiones de EF Core, cada operación que realice utilizando EF Core se convierte en su propia operación "reintentable". Cada consulta y cada llamada a **SaveChanges**, se reintentarán como una unidad si ocurre una falla transitoria.

Sin embargo, si su código inicia una transacción con **BeginTransaction**, está definiendo su propio grupo de operaciones que deben tratarse como una unidad: todo lo que se encuentra dentro de la transacción se revierte si ocurre una falla. Verá una excepción como la siguiente si intenta ejecutar esa transacción cuando usa una estrategia de ejecución (política de reintento) e incluye varias llamadas a **SaveChanges** desde múltiples **DbContexts** en la transacción.

```
System.InvalidOperationException: The configured execution strategy
'SqlServerRetryingExecutionStrategy' does not support user initiated
transactions. Use the execution strategy returned by
'DbContext.Database.CreateExecutionStrategy()' to execute all the operations in
the transaction as a retriable unit.
```

La solución es invocar manualmente la estrategia de ejecución de EF con un delegado que represente todo lo que se debe ejecutar. Si se produce una falla transitoria, la estrategia de ejecución invocará al delegado nuevamente. Por ejemplo, el siguiente código muestra cómo se implementa en eShopOnContainers con dos **DbContexts** múltiples (**_catalogContext** y **IntegrationEventLogContext**) al actualizar un producto y luego guarda el objeto **ProductPriceChangedIntegrationEvent**, que necesita utilizar un DbContext diferente.

```
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem
                                                productToUpdate)
{
    // Other code ...

    // Update current product
    catalogItem = productToUpdate;

    // Use of an EF Core resiliency strategy when using multiple DbContexts
    // within an explicit transaction
    // See:
    // https://docs.microsoft.com/ef/core/miscellaneous/connection-resiliency
    var strategy = _catalogContext.Database.CreateExecutionStrategy();

    await strategy.ExecuteAsync(async () =>
    {
        // Achieving atomicity between original Catalog database operation and the
        // IntegrationEventLog thanks to a local transaction
        using (var transaction = _catalogContext.Database.BeginTransaction())
        {
            _catalogContext.CatalogItems.Update(catalogItem);
            await _catalogContext.SaveChangesAsync();

            // Save to EventLog only if product price changed
            if (raiseProductPriceChangedEvent)
                await _integrationEventLogService.SaveEventAsync(priceChangedEvent);

            transaction.Commit();
        }
    });
```

El primer DbContext es **_catalogContext** y el segundo DbContext está dentro del objeto **_integrationEventLogService**. El *commit* se realiza en múltiples DbContexts utilizando una estrategia de ejecución de EF.

Recursos adicionales

- **Connection Resiliency and Command Interception with the Entity Framework in an ASP.NET MVC Application**
  https://docs.microsoft.com/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/connection-resiliency-and-command-interception-with-the-entity-framework-in-an-asp-net-mvc-application
- **Cesar de la Torre. Using Resilient Entity Framework Core Sql Connections and Transactions**
  https://blogs.msdn.microsoft.com/cesardelatorre/2017/03/26/using-resilient-entity-framework-core-sql-connections-and-transactions-retries-with-exponential-backoff/

Implementando reintento personalizado de llamadas HTTP con retroceso exponencial

Para crear microservicios resilientes, debe manejar posibles escenarios de falla HTTP. Para ese propósito, podría crear su propia implementación de reintentos con retroceso exponencial.

Además de manejar la no disponibilidad temporal de recursos, el retroceso exponencial también debe tener en cuenta que el proveedor de la nube puede reducir la disponibilidad de recursos para evitar la sobrecarga de uso. Por ejemplo, crear demasiadas solicitudes de conexión muy rápidamente puede verse como un ataque de denegación de servicio (DoS) por parte del proveedor de la nube. Como resultado, debe proporcionar un mecanismo para reducir las solicitudes cuando se ha encontrado un umbral de capacidad.

Como exploración inicial, puede implementar su propio código con una clase utilitaria para el retroceso exponencial como en RetryWithExponentialBackoff.cs, más un código como el siguiente (que también está disponible en un repositorio de GitHub).

```
public sealed class RetryWithExponentialBackoff
{
    private readonly int maxRetries, delayMilliseconds, maxDelayMilliseconds;

    public RetryWithExponentialBackoff(int maxRetries = 50,
                                       int delayMilliseconds = 200,
                                       int maxDelayMilliseconds = 2000)
    {
        this.maxRetries = maxRetries;
        this.delayMilliseconds = delayMilliseconds;
        this.maxDelayMilliseconds = maxDelayMilliseconds;
    }

    public async Task RunAsync(Func<Task> func)
    {
        ExponentialBackoff backoff = new ExponentialBackoff(this.maxRetries,
                                         this.delayMilliseconds,
                                         this.maxDelayMilliseconds);
        retry:
        try
        {
            await func();
        }
        catch (Exception ex) when (ex is TimeoutException ||
                                   ex is System.Net.Http.HttpRequestException)
        {
            Debug.WriteLine("Exception raised is: " +
                            ex.GetType().ToString() +
                            " –Message: " + ex.Message +
                            " -- Inner Message: " +
                            ex.InnerException.Message);
            await backoff.Delay();
            goto retry;
        }
    }
}
public struct ExponentialBackoff
{
    private readonly int m_maxRetries, m_delayMilliseconds, m_maxDelayMilliseconds;
    private int m_retries, m_pow;
```

```
    public ExponentialBackoff(int maxRetries, int delayMilliseconds,
                              int maxDelayMilliseconds)
    {
        m_maxRetries = maxRetries;
        m_delayMilliseconds = delayMilliseconds;
        m_maxDelayMilliseconds = maxDelayMilliseconds;
        m_retries = 0;
        m_pow = 1;
    }

    public Task Delay()
    {
        if (m_retries == m_maxRetries)
        {
            throw new TimeoutException("Max retry attempts exceeded.");
        }
        ++m_retries;
        if (m_retries < 31)
        {
            m_pow = m_pow << 1; // m_pow = Pow(2, m_retries - 1)
        }
        int delay = Math.Min(m_delayMilliseconds * (m_pow - 1) / 2,
                             m_maxDelayMilliseconds);
        return Task.Delay(delay);
    }
}
```

El uso de este código en una aplicación C# cliente (otro microservicio cliente API Web, una aplicación ASP.NET MVC o incluso una aplicación C# Xamarin) es sencillo. El siguiente ejemplo muestra cómo, usando la clase **`HttpClient`**.

```
public async Task<Catalog> GetCatalogItems(int page,int take, int? brand, int? type)
{
    _apiClient = new HttpClient();
    var itemsQs = $"items?pageIndex={page}&pageSize={take}";
    var filterQs = "";

    var catalogUrl =
        $"{_remoteServiceBaseUrl}items{filterQs}?pageIndex={page}&pageSize={take}";
    var dataString = "";
    //
    // Using HttpClient with Retry and Exponential Backoff
    //
    var retry = new RetryWithExponentialBackoff();
    await retry.RunAsync(async () =>
   {
        // work with HttpClient call
        dataString = await _apiClient.GetStringAsync(catalogUrl);
    });

    return JsonConvert.DeserializeObject<Catalog>(dataString);
}
```

Sin embargo, este código es adecuado sólo como una prueba de concepto. La siguiente sección explica cómo usar librerías más sofisticadas y comprobadas.

Implementando reintento de llamadas HTTP con retroceso exponencial usando Polly

El enfoque recomendado para los reintentos con retroceso exponencial es aprovechar librerías .NET más avanzadas, como la librería open source Polly.

Polly es una librería .NET que proporciona resiliencia y capacidad de recuperación de fallas transitorias. Puede implementar esas capacidades fácilmente aplicando las políticas de Polly, como **`Retry`**, **`Circuit Breaker`**, **`BulkheadIsolation`**, **`Timeout`** y **`Fallback`**. Polly apunta a .NET 4.x y .NET Standard Library 1.0 (que es compatible con .NET Core).

La política de reintento de Polly es el enfoque utilizado en eShopOnContainers al implementar reintentos HTTP. Puede implementar una interfaz, para poder inyectar un **`HttpClient`** estándar de **`HttpClient`** o una versión resiliente de **`HttpClient`** utilizando Polly, dependiendo de qué configuración de la directiva de reintento desee usar.

El siguiente ejemplo muestra la interfaz implementada en eShopOnContainers.

```
public interface IHttpClient
{
    Task<string> GetStringAsync(string uri, string authorizationToken = null,
                                string authorizationMethod = "Bearer");

    Task<HttpResponseMessage> PostAsync<T>(string uri, T item,
                          string authorizationToken = null, string requestId = null,
                          string authorizationMethod = "Bearer");

    Task<HttpResponseMessage> DeleteAsync(string uri,
                          string authorizationToken = null, string requestId = null,
                          string authorizationMethod = "Bearer");
    // Other methods ...
}
```

Puede utilizar la implementación estándar si no desea usar un mecanismo resiliente, como cuando desarrolla o prueba enfoques más simples. El siguiente código muestra la implementación estándar de **HttpClient** que permite, opcionalmente, solicitudes con tokens de autenticación.

```
public class StandardHttpClient : IHttpClient
{
    private HttpClient _client;
    private ILogger<StandardHttpClient> _logger;

    public StandardHttpClient(ILogger<StandardHttpClient> logger)
    {
        _client = new HttpClient();
        _logger = logger;
    }
    public async Task<string> GetStringAsync(string uri,
                                             string authorizationToken = null,
                                             string authorizationMethod = "Bearer")
    {
        var requestMessage = new HttpRequestMessage(HttpMethod.Get, uri);
        if (authorizationToken != null)
        {
            requestMessage.Headers.Authorization =
             new AuthenticationHeaderValue(authorizationMethod, authorizationToken);
        }

        var response = await _client.SendAsync(requestMessage);
        return await response.Content.ReadAsStringAsync();
    }

    public async Task<HttpResponseMessage> PostAsync<T>(string uri, T item,
                string authorizationToken = null, string requestId = null,
                string authorizationMethod = "Bearer")
    {
    // Rest of the code and other Http methods ...
```

La implementación interesante es codificar otra clase similar, pero utilizando Polly para implementar los mecanismos resilientes que desea usar, en el siguiente ejemplo, reintentos con retroceso exponencial.

```
public class ResilientHttpClient : IHttpClient
{
    private HttpClient _client;
    private PolicyWrap _policyWrapper;
    private ILogger<ResilientHttpClient> _logger;

    public ResilientHttpClient(Policy[] policies,
                               ILogger<ResilientHttpClient> logger)
    {
        _client = new HttpClient();
        _logger = logger;

        // Add Policies to be applied
        _policyWrapper = Policy.WrapAsync(policies);
    }

    private Task<T> HttpInvoker<T>(Func<Task<T>> action)
    {
        // Executes the action applying all
        // the policies defined in the wrapper
        return _policyWrapper.ExecuteAsync(() => action());
    }

    public Task<string> GetStringAsync(string uri,
                                       string authorizationToken = null,
                                       string authorizationMethod = "Bearer")
    {
        return HttpInvoker(async () =>
        {
            var requestMessage = new HttpRequestMessage(HttpMethod.Get, uri);

            // The Token's related code eliminated for clarity in code snippet

            var response = await _client.SendAsync(requestMessage);
            return await response.Content.ReadAsStringAsync();
        });
    }
    // Other Http methods executed through HttpInvoker so it applies Polly policies
    // ...
}
```

Con Polly, se define una política para reintentar, con el número de reintentos, la configuración de retroceso exponencial y las acciones a tomar cuando hay una excepción HTTP, como registrar el error. En este caso, la política está configurada, por lo que intentará la cantidad de veces especificada al registrar los tipos en el contenedor IoC. Debido a la configuración de retroceso exponencial, cada vez que el código detecta una excepción **`HttpRequestException`**, reintenta la solicitud HTTP después de esperar una cantidad de tiempo que aumenta exponencialmente según cómo se haya configurado la política.

El método más importante es **HttpInvoker**, que realiza las solicitudes HTTP en este utilitario. Ese método ejecuta internamente la solicitud HTTP con **_policyWrapper.ExecuteAsync**, que tiene en cuenta la política de reintento.

En eShopOnContainers, usted especifica las políticas de Polly cuando registra los tipos en el contenedor IoC, como en el siguiente código de la clase Startup.cs en aplicación web MVC.

```
// Startup.cs class

if (Configuration.GetValue<string>("UseResilientHttp") == bool.TrueString)
{
    services.AddTransient<IResilientHttpClientFactory,
                          ResilientHttpClientFactory>();

    services.AddSingleton<IHttpClient,
                          ResilientHttpClient>(sp =>
                                  sp.GetService<IResilientHttpClientFactory>().
                                                                  CreateResilientHttpClient());
}
else
{
    services.AddSingleton<IHttpClient, StandardHttpClient>();
}
```

Tenga en cuenta que los objetos **IHttpClient** se instancian como *singleton* en lugar de transitorios, de modo que el servicio utilice las conexiones TCP de manera eficiente y no se produzca un problema con los sockets. Además, cuando se trabaja con IHttpClient como singleton, también hay que manejar este problema con los cambios de DNS.

Pero el punto importante sobre la resiliencia es que se aplica la política **`WaitAndRetryAsync`** de Polly dentro de **`ResilientHttpClientFactory`** en el método **`CreateResilientHttpClient`**, como se muestra en el siguiente código:

```
public  ResilientHttpClient CreateResilientHttpClient()
        => new ResilientHttpClient(CreatePolicies(), _logger);
// Other code

private Policy[] CreatePolicies()
    => new Policy[]
    {
        Policy.Handle<HttpRequestException>()
        .WaitAndRetryAsync(
            // number of retries
            6,
            // exponential backoff
            retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
            // on retry
            (exception, timeSpan, retryCount, context) =>
            {
                var msg = $"Retry {retryCount} implemented with Pollys
                    RetryPolicy " +
                    $"of {context.PolicyKey} " +
                    $"at {context.ExecutionKey}, " +
                    $"due to: {exception}.";
                _logger.LogWarning(msg);
                _logger.LogDebug(msg);
            }),
    }
```

Implementando el patrón de Interruptor Automático

Como se señaló anteriormente, debe manejar las fallas que pueden tomar un tiempo variable para recuperarse, como podría ocurrir cuando intenta conectarse a un recurso o servicio remoto. El manejo de este tipo de falla puede mejorar la estabilidad y la resiliencia de una aplicación.

En un entorno distribuido, las llamadas a recursos y servicios remotos pueden fallar debido a problemas transitorios, como conexiones de red lentas y tiempos de espera o si los recursos están respondiendo lentamente o no están disponibles temporalmente. Por lo general, estas fallas se corrigen luego de un corto período de tiempo y una aplicación robusta en la nube debería estar preparada para manejarlas. utilizando una estrategia como el patrón de Reintento.

Sin embargo, también puede haber situaciones en las que las fallas se deban a eventos imprevistos, que pueden tardar mucho más en solucionarse. Estas fallas pueden variar en severidad desde una pérdida parcial de conectividad, hasta la falla completa de un servicio. En estas situaciones, puede no tener sentido que una aplicación intente continuamente una operación, que probablemente no tenga éxito. En cambio, la aplicación debe codificarse para aceptar que la operación ha fallado y manejar la falla en consecuencia.

El patrón de Interruptor Automático tiene un propósito diferente al patrón de Reintento. El patrón de Reintento permite a una aplicación reintentar una operación con la expectativa de que finalmente tendrá éxito. El patrón de Interruptor Automático impide que una aplicación realice una operación que probablemente falle. Una aplicación puede combinar estos dos patrones utilizando el patrón de

reintento para invocar una operación a través de un interruptor automático. Sin embargo, la lógica de reintento debe ser sensible a cualquier excepción devuelta por el interruptor automático y debe abandonar los reintentos si el interruptor automático indica que una falla no es transitoria.

Implementando un patrón de Interruptor Automático con Polly

Al igual que cuando se implementan reintentos, el enfoque recomendado para los interruptores automáticos es aprovechar las librerías probadas de .NET como Polly.

La aplicación eShopOnContainers usa la política Circuit Breaker de Polly, al implementar reintentos HTTP. De hecho, la aplicación aplica ambas políticas a la clase utilitaria **`ResilientHttpClient`**. Siempre que utilice un objeto de tipo **`ResilientHttpClient`** para peticiones HTTP (desde eShopOnContainers), estará aplicando ambas políticas, pero también podría añadir políticas adicionales.

La única adición aquí al código utilizado para los reintentos de llamadas HTTP es el código donde se agrega la política de Interruptor Automático a la lista de políticas a usar, como se muestra al final del siguiente código:

```
public  ResilientHttpClient CreateResilientHttpClient()
        => new ResilientHttpClient(CreatePolicies(), _logger);

private Policy[] CreatePolicies()
    => new Policy[]
    {
        Policy.Handle<HttpRequestException>()
        .WaitAndRetryAsync(
            // number of retries
            6,
            // exponential backofff
            retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
            // on retry
            (exception, timeSpan, retryCount, context) =>
            {
                var msg = $"Retry {retryCount} implemented with Polly
                    RetryPolicy " +
                    $"of {context.PolicyKey} " +
                    $"at {context.ExecutionKey}, " +
                    $"due to: {exception}.";
                _logger.LogWarning(msg);
                _logger.LogDebug(msg);
            }),
        Policy.Handle<HttpRequestException>()
        .CircuitBreakerAsync(
            // number of exceptions before breaking circuit
            5,
            // time circuit opened before retry
            TimeSpan.FromMinutes(1),
            (exception, duration) =>
            {
                // on circuit opened
                _logger.LogTrace("Circuit breaker opened");
            },
            () =>
            {
                // on circuit closed
                _logger.LogTrace("Circuit breaker reset");
            })};
    }
```

El código agrega una política al *wrapper* (envoltorio) HTTP. Esta política define un interruptor automático que se abre cuando el código detecta el número especificado de excepciones consecutivas, tal y como se indica en el parámetro **`exceptionsAllowedBeforeBreaking`** (5 en este caso). Cuando el circuito está abierto, las peticiones HTTP no funcionan, sino que se presenta una excepción.

Los interruptores automáticos también se deben utilizar para redirigir las peticiones a una infraestructura de respaldo, si existe la posibilidad de que tenga problemas en un recurso que se despliegue en un entorno diferente al de la aplicación o servicio cliente que realiza la llamada HTTP. De esta manera, si hay una interrupción en el centro de datos, que afecta sólo a sus microservicios

back-end pero no a sus aplicaciones cliente, las aplicaciones cliente pueden redirigirse a los servicios *fallback*. Polly tiene planeada una nueva política para automatizar este escenario con la política de conmutación por error (failover policy).

Por supuesto, todas esas características son para los casos en los que se está gestionando el *failover* desde el código. NET, en lugar de que Azure lo gestione automáticamente por usted, con transparencia de ubicación.

Usando la clase utilitaria ResilientHttpClient desde eShopOnContainers

Puede usar la clase de utilitaria **ResilientHttpClient** de forma similar a como usa la clase .NET **HttpClient**. En el siguiente ejemplo de la aplicación web eShopOnContainers MVC (la clase de agente **OrderingService** utilizada por **OrderController**), el objeto **ResilientHttpClient** se inyecta a través del parámetro **httpClient** del constructor. A continuación, el objeto se utiliza para realizar peticiones HTTP.

```
public class OrderingService : IOrderingService
{
    private IHttpClient _apiClient;
    private readonly string _remoteServiceBaseUrl;
    private readonly IOptionsSnapshot<AppSettings> _settings;
    private readonly IHttpContextAccessor _httpContextAccesor;

    public OrderingService(IOptionsSnapshot<AppSettings> settings,
                           IHttpContextAccessor httpContextAccesor,
                           IHttpClient httpClient)
    {
        _remoteServiceBaseUrl = $"{settings.Value.OrderingUrl}/api/v1/orders";
        _settings = settings;
        _httpContextAccesor = httpContextAccesor;
        _apiClient = httpClient;
    }

    async public Task<List<Order>> GetMyOrders(ApplicationUser user)
    {
        var context = _httpContextAccesor.HttpContext;
        var token = await context.Authentication.GetTokenAsync("access_token");

        _apiClient.Inst.DefaultRequestHeaders.Authorization = new
                 System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
        var ordersUrl = _remoteServiceBaseUrl;
        var dataString = await _apiClient.GetStringAsync(ordersUrl);
        var response = JsonConvert.DeserializeObject<List<Order>>(dataString);

        return response;
    }
    // Other methods ...

    async public Task CreateOrder(Order order)
    {
        var context = _httpContextAccesor.HttpContext;
        var token = await context.Authentication.GetTokenAsync("access_token");
        _apiClient.Inst.DefaultRequestHeaders.Authorization = new
                 System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
        _apiClient.Inst.DefaultRequestHeaders.Add("x-requestid",
```

```
                                                        order.RequestId.ToString());
        var ordersUrl = $"{_remoteServiceBaseUrl}/new";
        order.CardTypeId = 1;
        order.CardExpirationApiFormat();
        SetFakeIdToProducts(order);
        var response = await _apiClient.PostAsync(ordersUrl, order);
        response.EnsureSuccessStatusCode();
    }
}
```

Cada vez que se utiliza el campo **_apiClient**, utiliza internamente la clase "envoltorio" con las políticas de Polly: la política de reintentos, la política de interruptor automático y cualquier otra política, de la colección de políticas de Polly, que desee aplicar.

Probando los reintentos en eShopOnContainers

Siempre que inicie la solución eShopOnContainers en un servidor Docker, necesita iniciar varios contenedores. Algunos de los contenedores son más lentos para arrancar e inicializar, como el contenedor de SQL Server. Esto es especialmente cierto la primera vez que despliega la aplicación eShopOnContainers en Docker, ya que necesita configurar las imágenes y la base de datos. El hecho de que algunos contenedores empiecen más despacio que otros puede hacer que el resto de los servicios eleven inicialmente excepciones HTTP, incluso si se establecen dependencias entre contenedores al nivel de **docker-compose**, como se explicó en secciones anteriores. Esas dependencias entre contenedores se encuentran sólo a nivel del proceso. Es posible que se inicie el proceso a nivel del contenedor, pero es posible que SQL Server no esté preparado para las consultas. El resultado puede ser una cascada de errores y la aplicación puede obtener una excepción al intentar consumir de ese contenedor en particular.

Es posible que también vea este tipo de error al iniciar la aplicación cuando se está implementando en la nube. En ese caso, los orquestadores podrían estar moviendo contenedores de un nodo o máquina virtual a otro (es decir, iniciando nuevas instancias) al balancear el número de contenedores a través de los nodos del *cluster*.

eShopOnContainers resuelve este problema usando el patrón de reintento que ilustramos anteriormente. También es por eso que, al iniciar la solución, puede obtener trazas de registro o advertencias como las siguientes:

```
"Retry 1 implemented with Polly's RetryPolicy, due to:
System.Net.Http.HttpRequestException: An error occurred while sending the
request. ---> System.Net.Http.CurlException: Couldn't connect to server\n   at
System.Net.Http.CurlHandler.ThrowIfCURLEError(CURLcode error)\n   at  [...].
```

Probando el Interruptor Automático en eShopOnContainers

Hay algunas formas de abrir el circuito y probarlo con eShopOnContainers.

Una opción es reducir el número permitido de reintentos a 1 en la política de interruptor automático y redesplegar toda la solución a Docker. Con un solo reintento, existe una buena probabilidad de que una petición HTTP falle durante el despliegue, el interruptor automático se abrirá y obtendrá un error.

Otra opción es utilizar el *middleware* personalizado que se implementa en el microservicio **Basket** (el del carrito de compras). Cuando este middleware está activado, captura todas las peticiones HTTP y

devuelve el código de estado 500. Puede habilitar el middleware haciendo una petición GET al URI que falla, como se indica a continuación:

- `GET /failing`

  Esta petición devuelve el estado actual del middleware. Si el middleware está habilitado, la petición devuelve el código de estado 500. Si el middleware está desactivado, no hay respuesta.

- `GET /failing?enable`

  Esta petición habilita el middleware.

- `GET /failing?disable`

  Esta petición desactiva el middleware.

Por ejemplo, una vez que la aplicación se está ejecutando, puede habilitar el *middleware* haciendo una petición utilizando el siguiente URI en cualquier navegador. Tenga en cuenta que el microservicio de pedido utiliza el puerto **5103** en este ejemplo.

```
http://localhost:5103/failing?enable
```

Entonces puede comprobar el estado utilizando el URI http://localhost:5103/failing, como se muestra en la Figura 8-4.

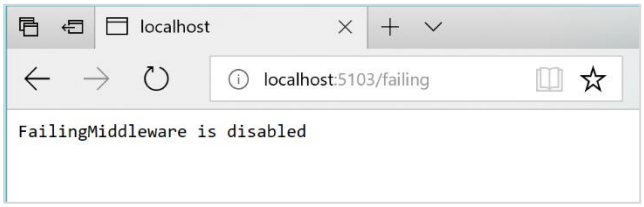


***Figura 8-4**. Revisando el estado del middleware ASP.NET Core que está "fallando", en este caso, deshabilitado.*

En este punto, el microservicio del carrito de compras responde con el código de estado 500 cada vez que se llama a invocarlo.

Una vez que el *middleware* se está ejecutando, puede intentar hacer un pedido desde la aplicación web MVC. Debido a que las solicitudes fallan, el circuito se abrirá.

En el siguiente ejemplo, puede ver que la aplicación web MVC tiene un bloque **`catch`** en la lógica para realizar un pedido. Si el código detecta una excepción de circuito abierto, muestra al usuario un mensaje sencillo diciéndole que espere.

```
public class CartController : Controller
{
    //…
    public async Task<IActionResult> Index()
    {
        try
        {
            //… Other code
        }
        catch (BrokenCircuitException)
        {
            // Catch error when Basket.api is in circuit-opened mode
            HandleBrokenCircuitException();
        }
        return View();
    }

    private void HandleBrokenCircuitException()
    {
        TempData["BasketInoperativeMsg"] = "Basket Service is inoperative, please
try later on. (Business Msg Due to Circuit-Breaker)";
    }
}
```

En resumen. La política de reintento intenta varias veces hacer la petición HTTP y obtiene errores HTTP. Cuando el número de intentos alcanza el número máximo establecido para la política de interruptor automático (en este caso, 5), la aplicación lanza una excepción **`BrokenCircuitException`**. El resultado es un mensaje amigable, como se muestra en la Figura 8-5.

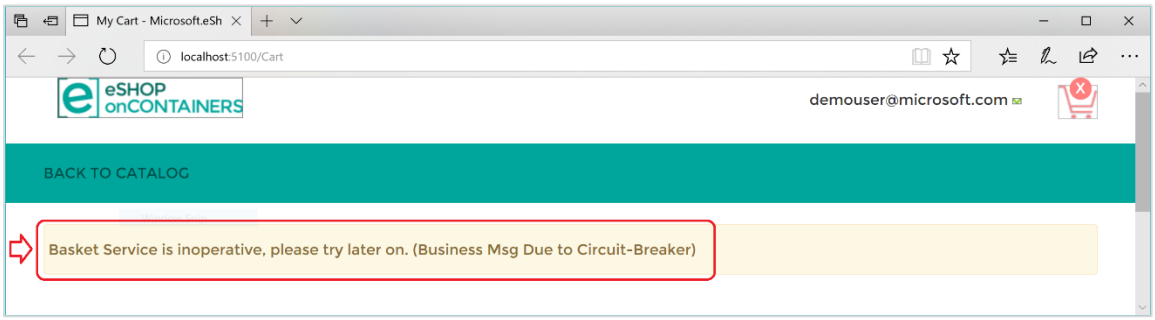


***Figura 8-5**. Interruptor automático devolviendo un error a la interfaz de usuario*

Puede implementar diferentes lógicas para cuándo abrir el circuito. También puede probar una petición HTTP contra un microservicio *back-end* diferente, si existe un centro de datos de respaldo o un sistema *back-end* redundante.

Por último, otra posibilidad para la política **`CircuitBreakerPolicy`** es usar **`Isolate`** (que fuerza a abrir y mantiene abierto el circuito) y **`Reset`** (que lo vuelve a cerrar). Éstos se pueden utilizar para crear un *endpoint* utilitario HTTP que invoque **`Isolate`** y **`Reset`** directamente en la política. Un *endpoint* HTTP de este tipo también podría utilizarse, asegurándolo adecuadamente, en producción para aislar temporalmente un sistema aguas abajo, como cuando se desea actualizarlo. O podría

disparar el circuito manualmente para proteger un sistema aguas abajo del que sospecha que está fallando.

Añadiendo una estrategia de fluctuación (*jitter*) a la política de reintento

Una política de Reintento regular puede afectar su sistema en casos de alta concurrencia y escalabilidad y con alta contención. Para superar picos de reintentos similares provenientes de muchos clientes en caso de interrupciones parciales, una buena solución es agregar una estrategia de fluctuación al algoritmo/política de reintento. Esto puede mejorar el rendimiento general del sistema añadiendo aleatoriedad al retroceso exponencial. Esto esparce los picos cuando surgen problemas. Cuando se usa Polly, el código para implementar fluctuaciones podría verse como el siguiente ejemplo:

```
Random jitterer = new Random();
Policy
  .Handle<HttpResponseException>() // etc
  .WaitAndRetry(5,    // exponential back-off plus some jitter
      retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
                    + TimeSpan.FromMilliseconds(jitterer.Next(0, 100))
  );
```

Recursos adicionales

- **Patrón Retry**
  https://docs.microsoft.com/azure/architecture/patterns/retry
- **Resiliencia de conexión** (Entity Framework Core)
  https://docs.microsoft.com/ef/core/miscellaneous/connection-resiliency
- **Polly** (.NET resilience and transient-fault-handling library)
  https://github.com/App-vNext/Polly
- **Patrón de disyuntor**
  https://docs.microsoft.com/azure/architecture/patterns/circuit-breaker
- **Marc Brooker. Jitter: Making Things Better With Randomness**
  https://brooker.co.za/blog/2015/03/21/backoff.html

Monitorización de la salud

La monitorización de la salud puede dar información casi en tiempo real sobre el estado de sus contenedores y microservicios. La monitorización de la salud es crítica para múltiples aspectos de los microservicios operativos y es especialmente importante cuando los orquestadores realizan actualizaciones parciales de la aplicación en fases, como se explica más adelante.

Las aplicaciones basadas en microservicios a menudo utilizan *heartbeats* (latidos) o chequeos de salud para permitir que sus monitores de rendimiento, programadores y orquestadores puedan llevar un registro de la multitud de servicios. Si los servicios no pueden enviar algún tipo de señal tipo "Estoy vivo", ya sea bajo demanda o según una programación, su aplicación podría correr riesgos cuando se despliegue las actualizaciones o simplemente se podrían detectar fallas demasiado tarde y no ser capaz de detener fallas en cascada, que puedan terminar en grandes interrupciones.

En el modelo típico, los servicios envían informes sobre su estado y esa información se agrega para proporcionar una visión general del estado de salud de su aplicación. Si está utilizando un

orquestador, puede proporcionar información de salud al *cluster* del orquestador, para que éste pueda actuar en consecuencia. Si usted invierte en informes de salud personalizados de alta calidad para su aplicación, puede detectar y corregir problemas en tiempo ejecución con mucha más facilidad.

Implementando controles de salud en servicios ASP.NET Core

Cuando desarrolle un microservicio o aplicación web ASP. NET Core, puede utilizar una librería llamada **HealthChecks** del equipo de ASP. NET. (versión preliminar disponible en GitHub).

Esta librería es fácil de usar y proporciona características que le permiten validar que cualquier recurso externo específico necesario para su aplicación (como una base de datos SQL Server o API remota) funciona correctamente. Cuando usted usa esta librería, también puede decidir qué significa que el recurso esté saludable, como explicaremos más adelante.

Para poder usar esta librería, primero debe referenciarla en sus microservicios. En segundo lugar, necesita una aplicación de *front-end* que pregunte por los informes de salud. Esa aplicación de *front-end* podría ser una aplicación personalizada de informes o podría ser un orquestador mismo que pueda reaccionar adecuadamente a los estados de salud.

Usando la librería HealthChecks en sus microservicios ASP.NET de *back-end*

Puede ver cómo se utiliza la librería HealthChecks en la aplicación eShopOnContainers. Para empezar, es necesario definir lo que constituye un estado saludable para cada microservicio. En la aplicación de ejemplo, los microservicios están saludables si la API del microservicio es accesible vía HTTP y si su base de datos SQL Server relacionada también está disponible.

En el futuro, podrá instalar la librería HealthChecks como un paquete NuGet. Pero, hasta el momento de escribir esta guía, necesita descargar y compilar el código como parte de su solución. Clone el código disponible en https://github.com/dotnet-architecture/HealthChecks y copie las siguientes carpetas a su solución.

```
src/common
src/Microsoft.AspNetCore.HealthChecks
src/Microsoft.Extensions.HealthChecks
src/Microsoft.Extensions.HealthChecks.SqlServer
```

También podría usar las verificaciones adicionales para Azure (`Microsoft.Extensions.HealthChecks.AzureStorage`), pero como esta versión de eShopOnContainers no tiene ninguna dependencia de Azure, no la necesita. No necesita los controles de salud de ASP.NET, porque eShopOnContainers está basada en ASP.NET Core.

La Figura 8-6 muestra la librería HealthChecks en Visual Studio, lista para ser utilizada como un bloque de construcción para cualquier microservicio.

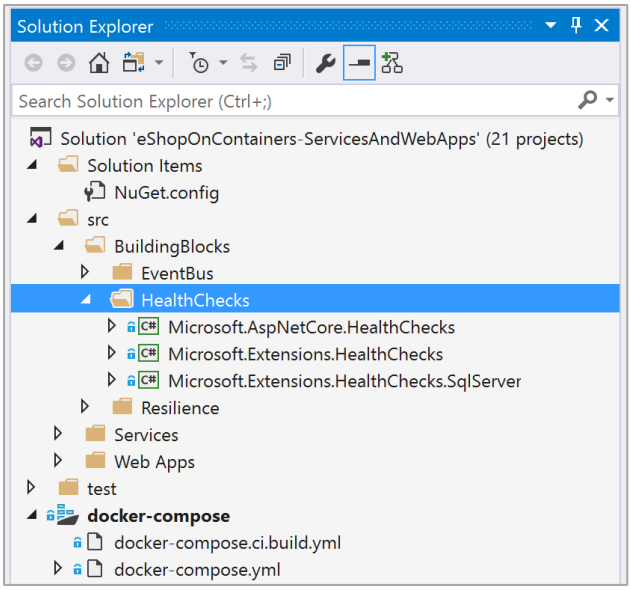


***Figura 8-6**. Código fuente de la librería ASP.NET Core HealthChecks en una solución Visual Studio*

Como se presentó anteriormente, lo primero que hay que hacer en cada proyecto de microservicio es agregar una referencia a las tres librerías de HealthChecks. Después de eso, agregue las acciones de control de salud que desea realizar en ese microservicio. Estas acciones son básicamente dependencias de otros microservicios (**`HttpUrlCheck`**) o bases de datos (actualmente **`SqlCheck*`** para bases de datos SQL Server). Añada la acción dentro de la clase Startup de cada microservicio o aplicación web ASP.NET.

Se debe configurar cada servicio o aplicación web, añadiendo todas sus dependencias HTTP o de base de datos como un método **`AddHealthCheck`**. Por ejemplo, la aplicación web MVC de eShopOnContainers depende de muchos servicios, por lo que tiene varios métodos **`AddCheck`** añadidos a los controles de salud.

Por ejemplo, en el siguiente código se puede ver cómo el microservicio de catálogo añade una dependencia a su base de datos SQL Server.

```
// Startup.cs from Catalog.api microservice
//
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    // Add framework services
    services.AddHealthChecks(checks =>
    {
        checks.AddSqlCheck("CatalogDb", Configuration["ConnectionString"]);
    });
    // Other services
  }
}
```

Sin embargo, la aplicación web MVC de eShopOnContainers tiene múltiples dependencias sobre el resto de los microservicios. Por lo tanto, llama un método **AddUrlCheck** para cada microservicio, como se muestra en el siguiente ejemplo:

```
// Startup.cs from the MVC web app
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    services.AddMvc();
    services.Configure<AppSettings>(Configuration);

    services.AddHealthChecks(checks =>
    {
        checks.AddUrlCheck(Configuration["CatalogUrl"]);
        checks.AddUrlCheck(Configuration["OrderingUrl"]);
        checks.AddUrlCheck(Configuration["BasketUrl"]);
        checks.AddUrlCheck(Configuration["IdentityUrl"]);
    });
  }
}
```

Por lo tanto, un microservicio no proporcionará un estado "saludable" hasta que todas sus verificaciones estén saludables también.

Si el microservicio no tiene una dependencia de un servicio o de SQL Server, sólo tiene que añadir un resultado de **Healthy ("Ok")**. El siguiente código es del microservicio del carrito de compras de eShopOnContainers **basket.api**. (El microservicio del carrito de compras usa el caché de Redis, pero la librería todavía no incluye un proveedor de control de salud de Redis.

```
services.AddHealthChecks(checks =>
{
   checks.AddValueTaskCheck("HTTP Endpoint", () => new
                   ValueTask<IHealthCheckResult>(HealthCheckResult.Healthy("Ok")));
});
```

Para que un servicio o aplicación web exponga el *endpoint* del control de salud, debe habilitar el *extension method* **UserHealthChecks([url_del_control_de_salud])**. Este método va a nivel del

WebHostBuilder en el método **Main** de la clase de **Program** de su servicio o aplicación web ASP.NET Core, justo después de **UseKestrel** como se muestra en el código de abajo.

```
namespace Microsoft.eShopOnContainers.WebMVC
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseHealthChecks("/hc")
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
            host.Run();
        }
    }
}
```

El proceso funciona así: cada microservicio expone el *endpoint* /hc. Ese *endpoint* es creado por el middleware de ASP.NET Core de la librería HealthChecks. Cuando se invoca ese *endpoint*, ejecuta todas las comprobaciones de estado configuradas en el método **AddHealthChecks** de la clase **Startup**.

El método **UseHealthChecks** espera un puerto o una ruta. Ese puerto o ruta es el *endpoint* que se usará utilizar para comprobar el estado de salud del servicio. Por ejemplo, el microservicio de catálogo utiliza la ruta **/hc**.

Guardando en caché las respuestas del control de salud

Dado que no desea causar una Denegación de Servicio (DoS) en sus servicios, o simplemente no desea afectar el rendimiento del servicio comprobando los recursos con demasiada frecuencia, puede almacenar en caché las devoluciones y configurar una duración del caché para cada control de salud.

De forma predeterminada, la duración de la caché se establece internamente en 5 minutos, pero puede cambiarla al configurar cada control de salud, como en el código siguiente:

```
checks.AddUrlCheck(Configuration["CatalogUrl"],1);  // 1 min as cache duration
```

Consultando sus microservicios para obtener su estado de salud

Cuando haya configurado los controles de salud como se describe aquí, una vez que el microservicio se esté ejecutando en Docker, puede comprobar directamente desde un navegador si está en buen estado. (Esto requiere que publique el puerto de contenedores fuera del servidor Docker, para poder acceder al contenedor a través de localhost o a través de la IP externa del servidor Docker. La figura 8-7 muestra una petición en un navegador y la respuesta correspondiente.

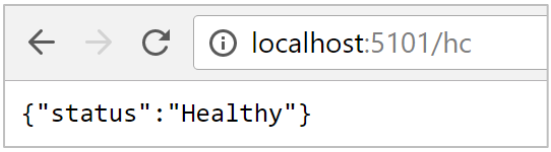


***Figura 8-7**. Revisando el estado de salud de un microservicio desde un navegador*

En esa prueba, puede ver que el microservicio **`catalog.api`** (corriendo en el puerto **5101**) está saludable, devolviendo el estado 200 de HTTP y la información de estado en JSON. También significa que el servicio también ha comprobado internamente la salud de su dependencia de la base de datos SQL Server y que el control de salud se ha reportado como saludable.

Usando guardianes (*watchdogs*)

Un guardián (*watchdog*) es un servicio separado que puede vigilar la salud y carga entre los servicios, e informar sobre la salud de los microservicios, consultando con la librería HealthChecks presentada anteriormente. Esto puede ayudar a evitar errores que no se detectarían basándose en la visualización de un servicio único. Los *watchdogs* también son un buen lugar para alojar código que puede realizar acciones de remediación para condiciones conocidas, sin la intervención del usuario.

El ejemplo de eShopOnContainers contiene una página web que muestra reportes de control de salud, como se muestra en la Figura 8-8. Este es el *watchdog* más simple que se puede tener, ya que sólo muestra el estado de los microservicios y aplicaciones web en eShopOnContainers. Por lo general, un *watchdog* también toma medidas cuando detecta servicios no saludables.

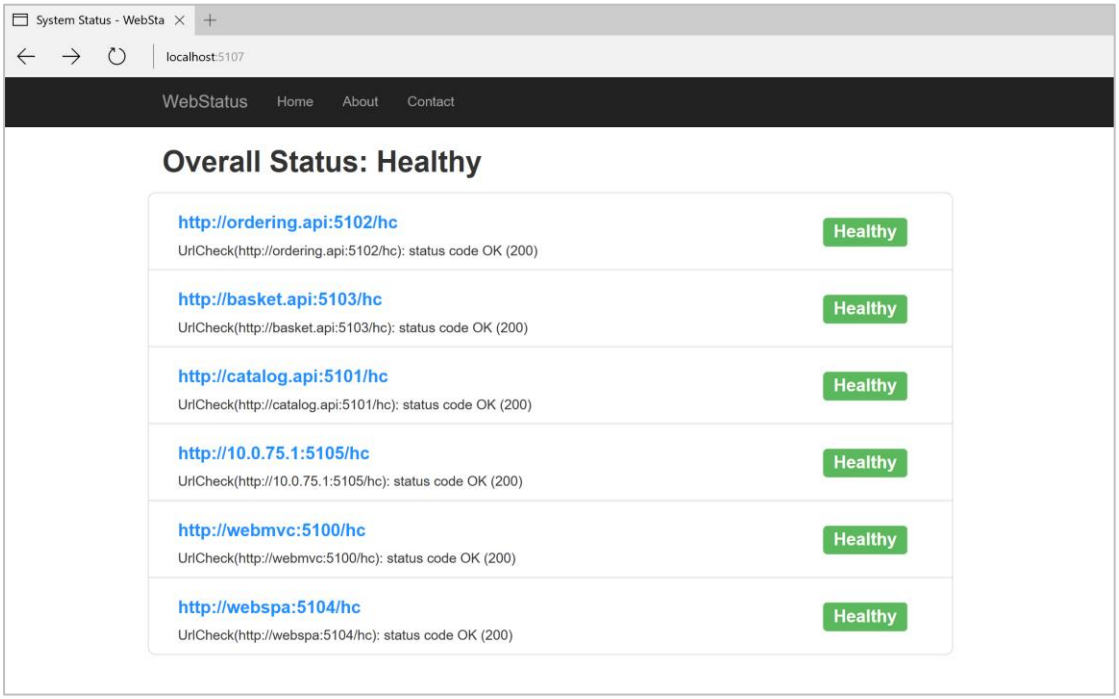


***Figura 8-8**. Ejemplo de una consulta del estado de salud en eShopOnContainers*

En resumen, el middleware ASP.NET de la librería ASP.NET Core HealthChecks, proporciona un *endpoint* único de control de salud para cada microservicio. Esto ejecutará todos los controles de salud definidos en el mismo y devolverá un estado de salud general dependiendo de todos esos controles.

La librería HealthChecks es extensible a través de nuevos controles de salud de futuros recursos externos. Por ejemplo, esperamos que en el futuro tenga controles para Redis y para otras bases de datos. La librería permite la presentación de informes de salud por parte de múltiples dependencias de servicios o aplicaciones y, a continuación, usted puede tomar medidas basadas en esos resultados si es necesario.

Controles de salud cuando se usan orquestadores

Para monitorizar la disponibilidad de los microservicios, los orquestadores como Docker Swarm, Kubernetes y Azure Service Fabric, realizan revisiones periódicas de salud, enviando peticiones para probar los microservicios. Cuando un orquestador determina que un servicio/contenedor no está saludable, deja de enrutar las peticiones a esa instancia. También suele crear una nueva instancia de ese contenedor.

Por ejemplo, la mayoría de los orquestadores pueden usar controles de salud para administrar despliegues sin ocasionar interrupciones del servicio. Sólo cuando el estado de un servicio/contenedor cambia a saludable, el orquestador comenzará a enrutar el tráfico hacia las instancias del servicio/contenedor.

La monitorización de la salud es especialmente importante cuando un orquestador realiza la actualización de la aplicación. Algunos orquestadores (como Azure Service Fabric) actualizan los servicios por fases, por ejemplo, pueden actualizar una quinta parte de la superficie del *cluster* para aplicar la actualización de cada aplicación. El conjunto de nodos que se actualiza al mismo tiempo se denomina dominio de actualización. Después de que cada dominio de actualización se haya actualizado y esté disponible para los usuarios, debe pasar las comprobaciones de estado antes de que el despliegue pase al siguiente dominio de actualización.

Otro aspecto de la salud de los servicios es reportar métricas del servicio. Esta es una capacidad avanzada del modelo de salud de algunos orquestadores, como Azure Service Fabric. Las métricas son importantes cuando se usa un orquestador, porque se usan para balancear el uso de recursos. Las métricas también pueden ser un indicador de la salud del sistema. Por ejemplo, es posible que tenga una aplicación con muchos microservicios y cada instancia reporte una métrica de peticiones por segundo (RPS – *requests per second*). Si un servicio está utilizando más recursos (memoria, procesador, etc.) que otro servicio, el orquestador podría mover las instancias de servicio en el *cluster* para intentar mantener uniforme el uso de recursos.

Tenga en cuenta que, si usted está usando Azure Service Fabric, éste proporciona su propio modelo de monitorización de la salud, que es más avanzado que los simples controles de salud.

Monitorización avanzada: visualización, análisis y alertas

La parte final de la monitorización consiste en visualizar el flujo de eventos, informar sobre el rendimiento del servicio y alertar cuando se detecte un problema. Puede utilizar diferentes soluciones para este aspecto de la monitorización.

Puede utilizar aplicaciones sencillas y personalizadas que muestren el estado de sus servicios, como la página personalizada que mostramos cuando explicamos los ASP.NET Core HealthChecks. O bien, puede utilizar herramientas más avanzadas como Azure Application Insights y Operations Management Suite para crear alertas basadas en los flujos de eventos.

Por último, si estaba almacenando todos los flujos de eventos, puede utilizar Microsoft Power BI o una solución de terceros como Kibana o Splunk para visualizar los datos.

Recursos adicionales

- **ASP.NET Core HealthChecks** (early release)
  https://github.com/aspnet/HealthChecks/
- **Introducción a la supervisión del mantenimiento de Service Fabric**
  https://docs.microsoft.com/azure/service-fabric/service-fabric-health-introduction
- **Azure Application Insights**
  https://azure.microsoft.com/services/application-insights/
- **Microsoft Operations Management Suite**
  https://www.microsoft.com/cloud-platform/operations-management-suite

Securizando Aplicaciones Web y Microservicios .NET

Implementando la autenticación en aplicaciones web y microservicios .NET

A menudo es necesario que los recursos y las APIs expuestas por un servicio se limiten a ciertos usuarios o clientes de confianza. El primer paso para tomar este tipo de decisiones de confianza a nivel de API es la autenticación. La autenticación es el proceso que consiste en determinar de forma fiable la identidad del usuario.

En los escenarios de microservicio, la autenticación suele gestionarse de forma centralizada. Si está utilizando una API Gateway, éste es un buen lugar para autenticarse, como se muestra en la Figura 9-1. Si utiliza este enfoque, asegúrese de que los microservicios individuales no pueden ser contactados directamente (sin la API Gateway) a menos que haya algún sistema de seguridad adicional para autenticar los mensajes, independientemente de que provengan o no del *gateway*.

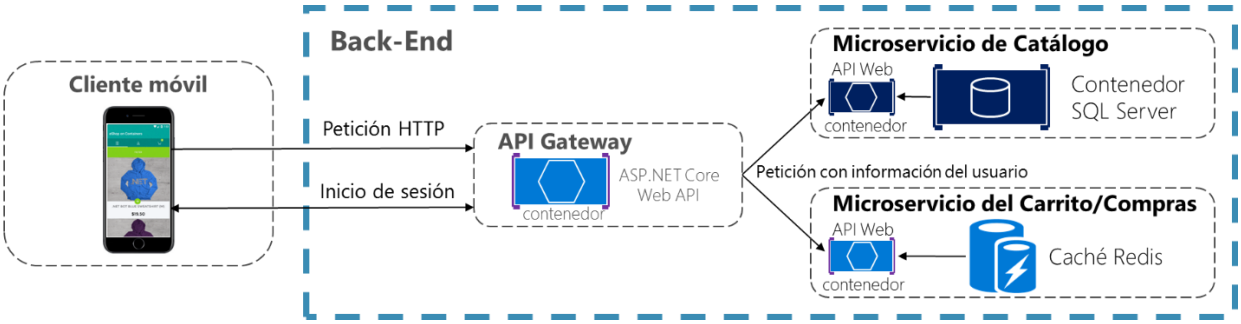


Figura 9-1. *Autenticación centralizada con un API Gateway*

Si se puede acceder directamente a los servicios, puede utilizar un servicio de autenticación como Azure Active Directory o un microservicio de autenticación dedicado, que actúe como un servicio de token de seguridad (STS) para autenticar a los usuarios. Las decisiones de confianza se comparten entre servicios con tokens de seguridad o cookies. (Estas pueden ser compartidas entre aplicaciones, si es necesario, en ASP. NET Core con los servicios de protección de datos. Este patrón se ilustra en la Figura 9-2.

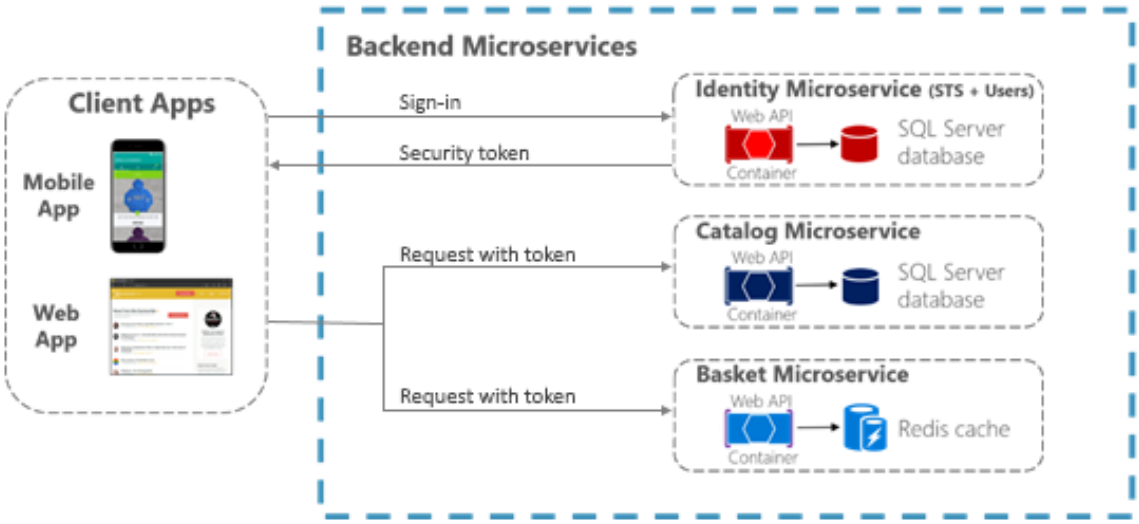


***Figura 9-2**. Autenticación por microservicio de identidad, se verifica la confiaza usando un token de autorización*

Autenticando con ASP.NET Core Identity

El mecanismo principal en ASP. NET Core para identificar a los usuarios de una aplicación es el sistema de membresía ASP. NET Core Identity. ASP. NET Core Identity almacena la información del usuario (incluyendo información de inicio de sesión, roles y afirmaciones - *claims*) en un almacenamiento de datos configurado por el desarrollador. Típicamente, el ASP. NET Core Identity data store es un almacenamiento provisto por Entity Framework en el paquete **`Microsoft.AspNetCore.Identity.EntityFrameworkCore`**. Sin embargo, también se pueden usar almacenamientos personalizados o paquetes de terceros, para almacenar información de identidad en Azure Table Storage, DocumentDB u otras ubicaciones.

El siguiente código se toma de la plantilla del proyecto ASP. NET Core Web Application con la autenticación de cuenta de usuario individual seleccionada. Muestra cómo configurar ASP.NET Core Identity usando Entity Framework Core en el método **`Startup.ConfigureServices`**.

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

Una vez que se configura ASP.NET Core Identity, se habilita llamando a **`app.UseIdentity`** en el método **`Startup.Configure`** del servicio.

El uso de ASP. NET Core Identity permite varios escenarios:

- Crear nueva información de usuario usando el tipo **`UserManager`** (**`userManager.CreateAsync`**).
- Autenticar usuarios usando el tipo **`SignInManager`**. Puede utilizar **`SignInInManager.signInAsync`** para iniciar sesión directamente o **`SignInManager.PasswordSignInAsync`** para confirmar que la contraseña del usuario es correcta y, a continuación, iniciar sesión.

- Identificar a un usuario basándose en la información almacenada en una cookie (que es leída por el middleware de ASP. NET Core Identity) para que las peticiones posteriores de un navegador incluyan la identidad y las *claims* (afirmaciones) del usuario registrado.

ASP. NET Core Identity también soporta autenticación de dos factores.

Para los escenarios de autenticación que usan almacenamiento local de datos de usuario y que persisten la identidad entre las peticiones usando cookies (como es típico para las aplicaciones web de MVC), ASP.NET Core Identity es una solución recomendada.

Autenticando con proveedores externos

El ASP.NET Core también soporta el uso de proveedores externos de autenticación para permitir que los usuarios inicien sesión a través de los flujos de OAuth 2.0. Esto significa que los usuarios pueden iniciar sesión usando los procesos de autenticación existentes de proveedores como Microsoft, Google, Facebook o Twitter y asociar esas identidades con una identidad ASP. NET Core en su aplicación.

Para utilizar la autenticación externa, incluya el middleware de autenticación apropiado en el *pipeline* de peticiones HTTP de su aplicación. Este middleware es responsable de gestionar las peticiones de rutas URI de retorno del proveedor de autenticación, capturar la información de identidad y ponerla a disposición a través del método **`SignInManager.GetExternalLoginInfo`**.

En la siguiente tabla se muestran los proveedores de autenticación externa más populares y sus paquetes NuGet asociados.

Provider	**Package**
Microsoft	`Microsoft.AspNetCore.Authentication.MicrosoftAccount`
Google	`Microsoft.AspNetCore.Authentication.Google`
Facebook	`Microsoft.AspNetCore.Authentication.Facebook`
Twitter	`Microsoft.AspNetCore.Authentication.Twitter`

En todos los casos, el middleware se registra con una llamada a un método de registro similar a **`app.use{ExternalProvider}Authentication`** en **`Startup.Configure`**. Estos métodos de registro toman un objeto de opciones que contiene un ID de aplicación e información secreta (una contraseña, por ejemplo), según lo necesite el proveedor. Los proveedores de autenticación externa requieren que la aplicación esté registrada (como se explica en la documentación de ASP.NET Core) para que puedan informar al usuario qué aplicación está solicitando acceso a su identidad.

Una vez que el middleware está registrado en **`Startup.Configure`**, puede solicitar a los usuarios que inicien sesión desde cualquier acción del controlador. Para ello, cree un objeto **`AuthenticationProperties`** que incluya el nombre del proveedor de autenticación y una URL de redireccionamiento. A continuación, devuelva una respuesta **`Challenge`** que pasa el objeto **`AuthenticationProperties`**. El siguiente código muestra un ejemplo de ello.

```
var properties = _signInManager.ConfigureExternalAuthenticationProperties(provider,
                                                                           redirectUrl);
return Challenge(properties, provider);
```

El parámetro **`redirectUrl`** incluye la URL a la que el proveedor externo debería redirigir una vez que el usuario se haya autenticado. La URL debe representar una acción que inicie sesión del usuario basándose en información de identidad externa, como en el siguiente ejemplo simplificado:

```
// Sign in the user with this external login provider if the user
// already has a login.
var result = await _signInManager.ExternalLoginSignInAsync(info.LoginProvider,
info.ProviderKey, isPersistent: false);
if (result.Succeeded)
{
    return RedirectToLocal(returnUrl);
}
else
{
    ApplicationUser newUser = new ApplicationUser
    {
        // The user object can be constructed with claims from the
        // external authentication provider, combined with information
        // supplied by the user after they have authenticated with
        // the external provider.
        UserName = info.Principal.FindFirstValue(ClaimTypes.Name),
        Email = info.Principal.FindFirstValue(ClaimTypes.Email)
    };

    var identityResult = await _userManager.CreateAsync(newUser);
    if (identityResult.Succeeded)
    {
        identityResult = await _userManager.AddLoginAsync(newUser, info);
        if (identityResult.Succeeded)
        {
            await _signInManager.SignInAsync(newUser, isPersistent: false);
        }

        return RedirectToLocal(returnUrl);
    }
}
```

Si usted elige la opción de autenticación **Individual User Account** cuando crea el proyecto de aplicación web ASP.NET Core en Visual Studio, se incluye en el proyecto todo el código necesario para iniciar sesión con un proveedor externo, como se muestra en la Figura 9-3.

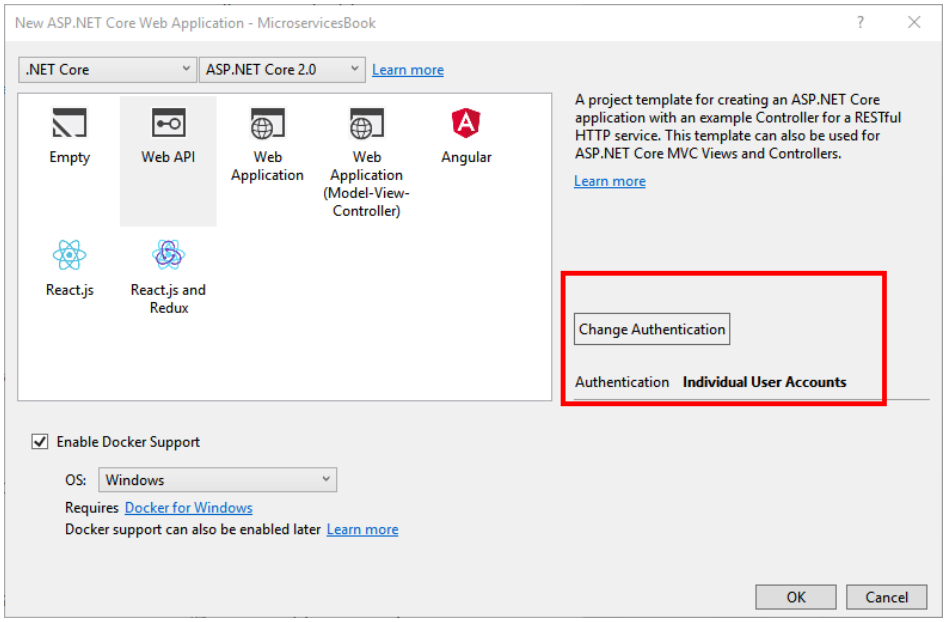


***Figura 9-3**. Seleccionando una opción para usar autenticación externa al crear un proyecto de aplicación web*

Además de los proveedores de autenticación externa listados anteriormente, hay disponibles paquetes de terceros que proporcionan middleware para utilizar muchos más proveedores de autenticación externa. Puede ver una lista en el repositorio AspNet.Security.OAuth.Providers en GitHub.

Por supuesto, también puede crear su propio middleware de autenticación externa.

Autenticando con tokens del portador (*bearer tokens*)

La autenticación con ASP.NET Core Identity (incluyendo también a los proveedores externos) funciona bien para muchos escenarios de aplicaciones web, en los que es adecuado almacenar la información del usuario en una *cookie*. En otros escenarios, sin embargo, las *cookies* no son un medio natural para persistir y transmitir datos.

Por ejemplo, en una aplicación API Web ASP.NET Core que expone *endpoints* RESTful a los que pueden acceder las aplicaciones de una sola página (SPA), clientes nativos o incluso otras APIs Web, normalmente desea utilizar la autenticación de *token* del portador en su lugar. Estos tipos de aplicaciones no funcionan con cookies, pero pueden recuperar fácilmente un *token* e incluirlo en la cabecera de autorización de peticiones posteriores. Para habilitar la autenticación por *token*, ASP.NET Core soporta varias opciones para usar OAuth 2.0 y OpenID Connect.

Autenticando con un proveedor de identidad OpenID Connect u OAuth 2.0

Si la información del usuario se almacena en Azure Active Directory u otra solución de identidad que soporte OpenID Connect u OAuth 2.0, puede utilizar el paquete **`Microsoft.AspNetCore.Authentication.OpenIdConnect`** para autenticarse usando el flujo de trabajo de OpenID Connect. Por ejemplo, una aplicación web ASP.NET Core puede usar middleware de ese paquete para autenticarse contra Azure Active Directory, como se muestra en el siguiente ejemplo:

```
// Configure the OWIN pipeline to use OpenID Connect auth
app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions
{
    ClientId = Configuration["AzureAD:ClientId"],
    Authority = String.Format(Configuration["AzureAd:AadInstance"],
                              Configuration["AzureAd:Tenant"]),
    ResponseType = OpenIdConnectResponseType.IdToken,
    PostLogoutRedirectUri = Configuration["AzureAd:PostLogoutRedirectUri"]
});
```

Los valores de **`Configuration["…`** son los que se crean al registrar su aplicación en Azure Active Directory. Un ID único de cliente puede ser compartido entre múltiples microservicios en una aplicación si todos ellos necesitan autenticar usuarios a través de Azure Active Directory.

Tenga en cuenta que cuando utiliza este flujo de trabajo, no se necesita el middleware de ASP.NET Core Identity, ya que toda la información de almacenamiento de usuario y autenticación es manejada por Azure Active Directory.

Emitiendo tokens de seguridad desde un servicio ASP.NET Core

Si prefiere emitir tokens de seguridad para usuarios locales de ASP.NET Core Identity, en lugar de usar un proveedor de identidad externo, puede aprovechar algunas librerías de terceros.

IdentityServer4 y OpenIddict son proveedores de OpenID Connect que se integran fácilmente con ASP.NET Core Identity para poder emitir tokens de seguridad desde un servicio ASP.NET Core. La documentación de IdentityServer4 contiene instrucciones detalladas para el uso de la librería. Sin embargo, los pasos básicos para usar IdentityServer4 para emitir tokens son los siguientes.

1. Llame a **`app.UseIdentityServer`** en el método **`Startup.Configure`** para agregar IdentityServer4 al pipeline de procesamiento de peticiones HTTP de la aplicación. Esto permite a la librería servir peticiones a los *endpoints* OpenID Connect y OAuth2 como /connect/token.
2. Configure IdentityServer4 en **`Startup.ConfigureServices`** haciendo una llamada a **`services.AddIdentityServer`**.
3. Configure Identity Server proporcionando los siguientes datos:
   - Las credenciales a utilizar para firmar.
   - Los recursos de Identidad y API a los que los usuarios pueden solicitar acceso:
     - Los recursos API representan datos protegidos o funcionalidad a la que un usuario puede acceder con un token de acceso. Un ejemplo de un recurso API sería una API web (o un conjunto de API) que requiere autorización.

- Los recursos de identidad representan información (afirmaciones) que se dan a un cliente para identificar a un usuario. Las afirmaciones pueden incluir el nombre de usuario, la dirección de correo electrónico, etc.
- Los clientes que se conectarán para solicitar tokens.
- El mecanismo de almacenamiento para la información del usuario, como ASP.NET Core Identity o una alternativa.

Cuando especifique los clientes y recursos que utilizará IdentityServer4, puede pasar una colección **`IEnumerable<T>`** del tipo apropiado a los métodos que reciben clientes o almacenes de recursos en memoria. Para escenarios más complejos, puede proporcionar tipos de proveedores de clientes o de recursos mediante la inyección de dependencias.

Un ejemplo de configuración para que IdentityServer4 use recursos y clientes en memoria, proporcionados por un tipo de **`IClientStore`** personalizado, se podría parecer lo siguiente:

```
// Add IdentityServer services
services.AddSingleton<IClientStore, CustomClientStore>();

services.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<ApplicationUser>();
```

Consumiendo tokens de seguridad

Autenticar contra un *endpoint* OpenID Connect o la emisión de sus propios *tokens* de seguridad cubre algunos escenarios. Pero ¿qué pasa con un servicio que simplemente necesita limitar el acceso a aquellos usuarios que tienen *tokens* de seguridad válidos que fueron proporcionados por un servicio diferente?

Para ese escenario, el middleware de autenticación que maneja los *tokens* JWT está disponible en el paquete **`Microsoft.AspNetCore.Authentication.JwtBearer`**. JWT significa "JSON Web Token" y es un formato común de token de seguridad (definido por RFC 7519) para la comunicación de afirmaciones (*claims*) de seguridad. Un simple ejemplo de cómo usar middleware para consumir tales tokens podría ser algo como lo siguiente. Este código debe preceder las llamadas al middleware ASP.NET Core MVC (**`app.UseMvc`**).

```
app.UseJwtBearerAuthentication(new JwtBearerOptions()
{
    Audience = "http://localhost:5001/",
    Authority = "http://localhost:5000/",
    AutomaticAuthenticate = true
});
```

Los parámetros en esta forma de usarlo son:

- **`Audience`** representa el receptor del *token* entrante o el recurso al que el *token* da acceso. Si el valor especificado en este parámetro no coincide con el parámetro **`aud`** del *token*, éste será rechazado.

- **`Authority`** es la dirección del servidor de autenticación que emite el token. El middleware de autenticación del JWT utiliza este URI para obtener la clave pública que puede utilizarse para validar la firma del *token*. El middleware también confirma que el parámetro **`iss`** del token coincide con este URI.
- **`AutomaticAuthenticate`** es un valor booleano que indica si el usuario definido por el token debe iniciar sesión automáticamente.

En este ejemplo no se utiliza otro parámetro, **`RequireHttpsMetadata`**. Es útil para fines de prueba; este parámetro se configura como **`false`** para que pueda probar en entornos donde no tenga certificados. En los despliegues del mundo real, los *tokens* JWT ***siempre*** se deben pasar sólo por HTTPS.

Con este middleware instalado, los *tokens* JWT se extraen automáticamente de las cabeceras de autorización. A continuación, se deserializa, se valida (utilizando los valores de los parámetros de **`Audience`** y **`Authority`**) y se almacena como información de usuario, a la que posteriormente se puede hacer referencia mediante acciones MVC o filtros de autorización.

El middleware de autenticación por *token* JWT también puede soportar escenarios más avanzados, como el uso de un certificado local para validar un *token* si la autoridad no está disponible. Para este escenario, puede especificar un objeto **`TokenValidationParameters`** en el objeto **`JwtBearerOptions`**.

Recursos adicionales

- **Uso compartido de las cookies entre aplicaciones**
  https://docs.microsoft.com/aspnet/core/security/data-protection/compatibility/cookie-sharing#sharing-authentication-cookies-between-applications
- **Introducción a Identity en ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/security/authentication/identity
- **Rick Anderson. Autenticación de dos factores con SMS**
  https://docs.microsoft.com/aspnet/core/security/authentication/2fa
- **Habilitando la autenticación con Facebook, Google y otros proveedores externos**
  https://docs.microsoft.com/aspnet/core/security/authentication/social/
- **Michell Anicas. An Introduction to OAuth 2**
  https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2
- **AspNet.Security.OAuth.Providers** (Repositorio GitHub para proveedores de OAuth en ASP.NET.
  https://github.com/aspnet-contrib/AspNet.Security.OAuth.Providers/tree/dev/src
- **Danny Strockis. Integrating Azure AD into an ASP.NET Core web app**
  https://azure.microsoft.com/resources/samples/active-directory-dotnet-webapp-openidconnect-aspnetcore/
- **IdentityServer4. Official documentation**
  https://identityserver4.readthedocs.io/en/release/

Acerca de la autorización en aplicaciones web y microservicios .NET

Después de la autenticación, las APIs Web de ASP.NET Core necesitan autorizar el acceso. Este proceso permite que un servicio ponga APIs a disposición de algunos usuarios autenticados, pero no de todos. La autorización se puede hacer según las funciones de los usuarios o en base a una política personalizada, que puede incluir la inspección de las *claims* (afirmaciones) u otras técnicas.

Restringir el acceso a una ruta ASP.NET Core MVC es tan fácil como aplicar un atributo **`Authorize`** al método de la acción (o a la clase del controlador si todas las acciones requieren autorización), como se muestra en el siguiente ejemplo:

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
```

Por defecto, al usar el atributo **`Authorize`** sin parámetros, se limitará el acceso a los usuarios autenticados para ese controlador o acción. Para restringir aún más la disponibilidad de una API, sólo para usuarios específicos, el atributo puede ampliarse para especificar roles o políticas que los usuarios deben satisfacer.

Implementando autorización basada en roles

ASP.NET Core Identity incluye el concepto de roles. Además de los usuarios, ASP.NET Core Identity almacena información sobre los diferentes roles usados por la aplicación y los usuarios a quienes se les asignan. Estas asignaciones se pueden modificar programáticamente con el tipo **`RoleManager`** (que actualiza las funciones en el almacenamiento persistente) y el tipo de **`UserManager`** (que gestiona la asignación de roles a usuarios).

Si está autenticando con *tokens* JWT, el middleware JWT ASP.NET Core cargará los roles del usuario basándose en las *claims* de roles encontradas en el token. Para limitar el acceso a una acción o controlador MVC a usuarios con roles específicos, puede incluir un parámetro **Roles** en la anotación (atributo) **Authorize**, como se muestra en el ejemplo siguiente:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

En este ejemplo, sólo los usuarios con los roles **Administrator** o **PowerUser** pueden acceder a las API en el controlador **ControlPanel** (como ejecutar la acción **SetTime**). La API **ShutDown** se restringe aún más, para permitir el acceso sólo a los usuarios en el rol **Administrator**.

Para exigir que un usuario tenga múltiples roles, utilice varios atributos de **Authorize**, como se muestra en el ejemplo siguiente:

```
[Authorize(Roles = "Administrator, PowerUser")]
[Authorize(Roles = "RemoteEmployee ")]
[Authorize(Policy = "CustomPolicy")]
public ActionResult API1 ()
{
}
```

En este ejemplo, para llamar a API1, un usuario debe:

- Tener el rol de **Administrator** o **PowerUser** y
- Tener el rol de **RemoteEmployee** y
- Satisfacer la política personalizada de autorización **CustomPolicy**.

Implementando autorización basada en políticas

Las reglas de autorización personalizadas también se pueden escribir usando políticas de autorización. En esta sección le ofrecemos una visión general. Más detalles están disponibles en el Taller de Autorización ASP.NET en línea.

Las políticas de autorización personalizadas se registran en el método **`Startup.ConfigureServices`** usando el método **`services.AddAuthorization`**. Este método recibe un delegado que configura un parámetro **`AuthorizationOptions`**.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("AdministratorsOnly", policy =>
        policy.RequireRole("Administrator"));
    options.AddPolicy("EmployeesOnly", policy =>
        policy.RequireClaim("EmployeeNumber"));
    options.AddPolicy("Over21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});
```

Como se muestra en el ejemplo, las políticas pueden asociarse con diferentes tipos de necesidades. Después de que las políticas se registran, se pueden aplicar a una acción o controlador al pasar el nombre de la política como el parámetro **`Policy`** del atributo **`Authorize`** (por ejemplo, **`[Authorize(Policy="EmployeesOnly")]`**. Las políticas pueden tener múltiples requisitos, no sólo uno, como se muestra en estos ejemplos.

En el ejemplo anterior, la primera llamada de **`AddPolicy`** es sólo una forma alternativa de autorizar por rol. Si se aplica **`[Authorize(Policy="AdministratorsOnly")]`** se aplica a una API, sólo los usuarios con el rol **`Administrator`** podrán acceder a ella.

La segunda llamada **`AddPolicy`** demuestra una manera fácil de exigir que el usuario debe tener una *claim* particular. El método de **`RequireClaim`** también recibe, opcionalmente, los valores esperados para la *claim*. Si se especifican valores, el requisito sólo se cumple si el usuario tiene una *claim* del tipo correcto y uno de los valores especificados. Si está utilizando el middleware de autenticación por *token* JWT, todas las propiedades del JWT estarán disponibles como claims de usuario.

La política más interesante que se muestra aquí, es el tercer método de **`AddPolicy`**, ya que utiliza un requisito personalizado de autorización. Al usar requisitos personalizados de autorización, puede tener un gran control sobre cómo se realiza la autorización. Para que esto funcione, debe implementar estos tipos:

- Un tipo **`Requirements`** que deriva de **`IAuthorizationRequirement`** y que contiene campos que especifican los detalles del requisito. En el ejemplo, este es un campo de edad para el tipo **`MinimumAgeRequirement`** del ejemplo.
- Un manejador que implementa **`AuthorizationHandler<T>`**, donde T es el tipo de **`IAuthorizationRequirement`** que el manejador puede satisfacer. El manejador debe implementar el método **`HandleRequirementAsync`**, que comprueba si un contexto específico que contiene información sobre el usuario satisface el requisito.

Si el usuario cumple con los requisitos, una llamada a **`context.Succeed`** indica que el usuario está autorizado. Si hay varias maneras en que un usuario puede satisfacer un requisito de autorización, se pueden crear varios manejadores.

Además de registrar los requisitos de las políticas personalizadas con las llamadas a **`AddPolicy`**, también necesita registrar los manejadores de requisitos personalizados mediante Inyección de Dependencias (**`servicios.AddTransient<IAuthorizationHandler, MinimumAgeHandler>()`**).

Un ejemplo de un requisito personalizado de autorización y su manejador para verificar la edad del usuario (basado en una *claim* **`DateOfBirth`**) está disponible en la documentación de autorización de ASP.NET Core.

Recursos adicionales

- **Autenticación en ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/security/authentication/identity
- **Autorización en ASP.NET Core**
  https://docs.microsoft.com/aspnet/core/security/authorization/introduction
- **Autorización basada en roles**
  https://docs.microsoft.com/aspnet/core/security/authorization/roles
- **Autorización personalizada basada en políticas**
  https://docs.microsoft.com/aspnet/core/security/authorization/policies

Guardando de forma segura los secretos de la aplicación durante el desarrollo

Para conectarse con recursos protegidos y otros servicios, las aplicaciones ASP.NET Core típicamente necesitan usar cadenas de conexión, contraseñas u otras credenciales que contengan información confidencial. Estas partes de información sensible se llaman secretos. Es una buena práctica no incluir secretos en el código fuente y ciertamente no almacenar secretos en el sistema de control de versiones. En su lugar, debería utilizar el modelo de configuración de ASP.NET Core para leer los secretos desde ubicaciones más seguras.

Debe separar los secretos para acceder a los recursos de desarrollo y pre-producción (*staging*) de los que se usan para acceder a los recursos de producción, porque diferentes individuos necesitarán acceder a esos conjuntos diferentes de secretos. Para almacenar secretos usados durante el desarrollo, los enfoques comunes son almacenar secretos en variables de entorno o mediante la herramienta ASP.NET Core Secret Manager. Para un almacenamiento más seguro en entornos de producción, los microservicios pueden almacenar secretos en una Azure Key Vault.

Guardando secretos en variables de entorno

Una forma de mantener secretos fuera del código fuente es que los desarrolladores establezcan secretos basados en cadenas, como variables de entorno en sus máquinas de desarrollo. Cuando utilice variables de entorno para almacenar secretos con nombres jerárquicos (los anidados en secciones de configuración), debe crear un nombre para las variables de entorno que incluya la jerarquía completa del nombre del secreto, delimitada por dos puntos (:).

Por ejemplo, establecer una variable de entorno **`Logging:LogLevel:Default`** a **`Debug`** es equivalente al valor de configuración del siguiente fichero JSON:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

Para acceder a estos valores desde variables de entorno, la aplicación sólo necesita llamar a **`AddEnvironmentVariables`** en su **`ConfigurationBuilder`** al construir un objeto **`IConfigurationRoot`**.

Tenga en cuenta que las variables de entorno generalmente se almacenan como texto plano, por lo que, si la máquina o el proceso con las variables de entorno se ve comprometido, los valores de las variables de entorno serán visibles.

Guardando secretos con el ASP.NET Core Secret Manager

La herramienta ASP.NET Core Secret Manager proporciona otro método para mantener los secretos fuera del código fuente. Para usar el Secret Manager Tool, incluya una referencia de herramientas (**`DotNetCliToolReference`**) al paquete **`Microsoft.Extensions.SecretManager.Tools`** en su fichero de proyecto. Una vez que esa dependencia está presente y ha sido restaurada, se puede usar

el comando **`dotnet user-secrets`** para establecer el valor de los secretos desde la línea de comandos. Estos secretos se almacenarán en un fichero JSON en el directorio de perfiles del usuario (los detalles varían según el sistema operativo), lejos del código fuente.

Los secretos establecidos por la Secret Manager tool están organizados por la propiedad **`UserSecretsId`** del proyecto que está utilizando los secretos. Por lo tanto, debe asegurarse de establecer la propiedad **`UserSecretsId`** en su fichero de proyecto (como se muestra en el fragmento a continuación). La cadena real utilizada como ID no es importante mientras sea única en el proyecto.

```
<PropertyGroup>
  <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

El uso de secretos almacenados con la Secret Manager tool en una aplicación, se logra llamando a **`AddUserSecrets<T>`** en la instancia **`ConfigurationBuilder`** para incluir secretos para la aplicación en su configuración. El parámetro genérico T debe ser un tipo del ensamblado al que se aplicó el **`UserSecretId`**. Normalmente, el uso de **`AddUserSecrets<Startup>`** está bien.

Usando el Azure Key Vault para proteger los secretos en producción

Los secretos almacenados como variables de entorno o almacenados por la Secret Manager tool siguen almacenados localmente y sin encriptar en la máquina. Una opción más segura para guardar secretos es el Azure Key Vault, que proporciona una ubicación central y segura para guardar llaves y secretos.

El paquete `Microsoft.Extensions.Configuration.AzureKeyVault` permite que una aplicación ASP.NET Core pueda leer la información de configuración del Azure Key Vault. Para empezar a usar secretos de Azure Key Vault, siga estos pasos:

1. Registre su aplicación como una aplicación Azure AD. (El acceso a las bóvedas de claves es gestionado por Azure AD.) Esto se puede hacer a través del portal de gestión de Azure.

   Alternativamente, si desea que su aplicación se autentique utilizando un certificado en lugar de una contraseña o un secreto de cliente, puede utilizar el nuevo *cmdlet* PowerShell New-AzureRmADApplication. El certificado que registre con Azure Key Vault sólo necesita su clave pública. (Su aplicación utilizará la clave privada.)

2. Proporcione acceso a la bóveda de claves a la aplicación registrada creando un nuevo *service principal*. Puede hacerlo utilizando los siguientes comandos de PowerShell:

   ```
   $sp = New-AzureRmADServicePrincipal -ApplicationId "<Application ID guid>"
   ```

   ```
   Set-AzureRmKeyVaultAccessPolicy -VaultName "<VaultName>" -ServicePrincipalName
   $sp.ServicePrincipalNames[0] -PermissionsToSecrets all -ResourceGroupName
   "<KeyVault Resource Group>"
   ```

3. Incluya la bóveda de claves como fuente de configuración en su aplicación llamando al método **`IConfigurationBuilder.AddAzureKeyVault`** cuando cree una instancia de

IConfigurationRoot. Tenga en cuenta que al llamar a **AddAzureKeyVault** requerirá el ID de la aplicación que se registró y dio acceso a la bóveda de claves en los pasos anteriores.

Desde la versión 3.14.0 del paquete Microsoft.IdentityModel.Clients.ActiveDirectory está disponible el método AddAzureKeyVault para.NET Standard, .NET Core y .NET Framework. Las aplicaciones ASP.NET Core también pueden acceder a un Azure Key Vault con autenticación basada en certificados mediante la creación explícita de un objeto **KeyVaultClient**, como se muestra en el siguiente ejemplo:

```
// Configure Key Vault client
var kvClient = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(async
    (authority, resource, scope) =>
{
    var cert = // Get certificate from local store/file/key vault etc. as needed
    // From the Microsoft.IdentityModel.Clients.ActiveDirectory pacakge
    var authContext = new AuthenticationContext(authority,
                                                TokenCache.DefaultShared);
    var result = await authContext.AcquireTokenAsync(resource,
           // From the Microsoft.Rest.ClientRuntime.Azure.Authentication pacakge
           new ClientAssertionCertificate("{Application ID}", cert));
    return result.AccessToken;
}));

// Get configuration values from Key Vault
var builder = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    // Other configuration providers go here.
    .AddAzureKeyVault("{KeyValueUri}", kvClient,
                      new DefaultKeyVaultSecretManager());
```

En este ejemplo, la llamada a **AddAzureKeyVault** viene al final del registro del proveedor de configuración. Es una buena práctica registrar Azure Key Vault como el último proveedor de configuración para que tenga la oportunidad de invalidar los valores de configuración de proveedores anteriores y para que ningún valor de configuración de otras fuentes reemplace los de la bóveda de claves.

Recursos adicionales

- **Uso de Azure Key Vault para proteger los secretos de la aplicación**
  https://docs.microsoft.com/azure/guidance/guidance-multitenant-identity-keyvault
- **Almacenamiento Seguro de los secretos de la aplicación durante el desarrollo**
  https://docs.microsoft.com/aspnet/core/security/app-secrets
- **Configurando la protección de datos**
  https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/overview
- **Administración y duración de claves**
  https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/default-settings#data-protection-default-settings

Aprendizajes clave

Como resumen de los aprendizajes clave, a continuación, presentamos las conclusiones más importantes de esta guía.

Beneficios del uso de contenedores. Las soluciones basadas en contenedores proporcionan el beneficio importante del ahorro de costes, ya que los contenedores son una solución a los problemas de despliegue causados por la falta de dependencias en los entornos de producción. Los contenedores mejoran significativamente las operaciones de DevOps y producción.

Los contenedores serán ubicuos. Los contenedores basados en Docker se están convirtiendo en el estándar de facto en la industria de contenedores, con el soporte de los proveedores más importantes en los ecosistemas Windows y Linux. Esto incluye Microsoft, Amazon AWS, Google e IBM. En un futuro próximo, es probable que Docker sea ubicuo tanto en centros de datos en la nube como locales.

Contenedores como unidad de despliegue. Un contenedor Docker se está convirtiendo en la unidad estándar de despliegue para cualquier aplicación o servicio basado en servidor.

Microservicios. La arquitectura de microservicios se está convirtiendo en el enfoque preferido para aplicaciones grandes o complejas de misión crítica, distribuidas y basadas en múltiples subsistemas independientes en forma de servicios autónomos. En una arquitectura basada en microservicios, la aplicación se construye como una colección de servicios que se pueden desarrollar, probar, versionar, desplegar y escalar independientemente, además pueden incluir cualquier base de datos autónoma relacionada.

Diseño basado en el dominio y SOA. Los patrones de arquitectura de microservicios de derivan de la arquitectura orientada a servicios (SOA) y el diseño basado en el dominio (DDD). Cuando diseña y desarrolla microservicios para entornos con reglas cambiantes del negocio, que conforman un dominio en particular, es importante tener en cuenta los enfoques y patrones de DDD.

Desafíos de los microservicios. Los microservicios ofrecen muchas capacidades poderosas, como el despliegue independiente, fuertes límites de subsistema y diversidad tecnológica. Sin embargo, también plantean muchos desafíos nuevos, relacionados con el desarrollo distribuido de aplicaciones, como modelos de datos fragmentados e independientes, comunicación resiliente entre microservicios, consistencia eventual y complejidad operativa que resulta de agregar la información de registro y monitorización de múltiples microservicios. Estos aspectos introducen un grado mayor de complejidad que una aplicación monolítica tradicional. Como resultado, sólo escenarios específicos son adecuados para aplicaciones basadas en microservicios. Estas incluyen aplicaciones grandes y complejas con múltiples subsistemas en evolución. En estos casos, vale la pena invertir en una arquitectura de software más compleja, ya que proporcionará una mayor agilidad a largo plazo y un mejor mantenimiento de las aplicaciones.

Contenedores para cualquier aplicación. Los contenedores son convenientes para los microservicios, pero no son exclusivos para ellos. Los contenedores también se pueden utilizar con aplicaciones monolíticas, incluyendo aplicaciones *legacy* basadas en el .NET Framework tradicional y

modernizadas a través de Windows Containers. Los beneficios de utilizar Docker, como las facilidades para resolver muchos problemas de despliegue a producción y proporcionar entornos de desarrollo y pruebas de última generación, se aplican a muchos tipos de aplicaciones diferentes.

CLI versus IDE. Con las herramientas de Microsoft, puede desarrollar aplicaciones .NET contenerizadas usando el enfoque que prefiera. Puede desarrollar con un CLI y un entorno basado en un editor de texto, usando Docker CLI y Visual Studio Code. O bien, puede utilizar un enfoque centrado en el IDE con Visual Studio y sus características exclusivas para Docker, como ser capaz de depurar aplicaciones multi-contenedor.

Aplicaciones resilientes en la nube. En los sistemas basados en nube y en los sistemas distribuidos en general, siempre existe el riesgo de fallo parcial. Dado que los clientes y los servicios son procesos separados (contenedores), es posible que un servicio no pueda responder a tiempo a la solicitud de un cliente. Por ejemplo, un servicio podría estar inactivo debido a una falla parcial o por mantenimiento, el servicio podría estar sobrecargado y responder extremadamente lento a las solicitudes, o simplemente podría no estar accesible por un tiempo breve, debido a problemas de red. Por lo tanto, una aplicación basada en la nube debe aceptar estos fallos y contar con una estrategia para responder a ellos. Estas estrategias pueden incluir políticas de reintento (reenvío de mensajes o peticiones de reintento) y la implementación de patrones de interruptores automáticos para evitar la carga exponencial de peticiones repetidas. Básicamente, las aplicaciones basadas en la nube deben tener mecanismos resilientes, ya sea personalizados o basados en la infraestructura de la nube, como los *frameworks* de alto nivel de los orquestadores o los buses de servicio.

Seguridad. Nuestro mundo moderno de contenedores y microservicios puede exponer nuevas vulnerabilidades. La seguridad básica de las aplicaciones se basa en la autenticación y autorización; existen múltiples formas de implementarlas. Sin embargo, la seguridad de los contenedores incluye componentes clave adicionales que resultan en aplicaciones intrínsecamente más seguras. Un elemento crítico para crear aplicaciones más seguras es tener una forma segura de comunicarse con otras aplicaciones y sistemas, algo que a menudo requiere credenciales, *tokens*, contraseñas y otros tipos de información confidencial, lo que generalmente se conoce como secretos de aplicación. Cualquier solución segura debe seguir las mejores prácticas de seguridad, tales como encriptar secretos durante el transporte, encriptar secretos en reposo y evitar que se produzcan fugas involuntarias cuando se consumen en la aplicación final. Esos secretos necesitan ser guardados y mantenidos a salvo en algún lugar. Para ayudarle con la seguridad, puede aprovechar la infraestructura de su orquestador elegido, o la infraestructura de nube como Azure Key Vault y las maneras en se puede utilizar desde el código de aplicación.

Orquestadores. Los orquestadores basados en contenedores como los que se proporcionan en Azure Container Service (Kubernetes, Mesos DC/OS, and Docker Swarm) y Azure Service Fabric son indispensables para cualquier aplicación basada en microservicios y para cualquier aplicación multi-contenedor con una complejidad significativa, necesidades de escalabilidad y evolución constante. Esta guía ha presentado los orquestadores y su papel en las soluciones basadas en microservicios y contenedores. Si sus necesidades de aplicación le llevan hacia aplicaciones contenerizadas complejas, encontrará útil buscar en los recursos adicionales para aprender más sobre orquestadores.