

Simplified CQRS and DDD microservice

High level design

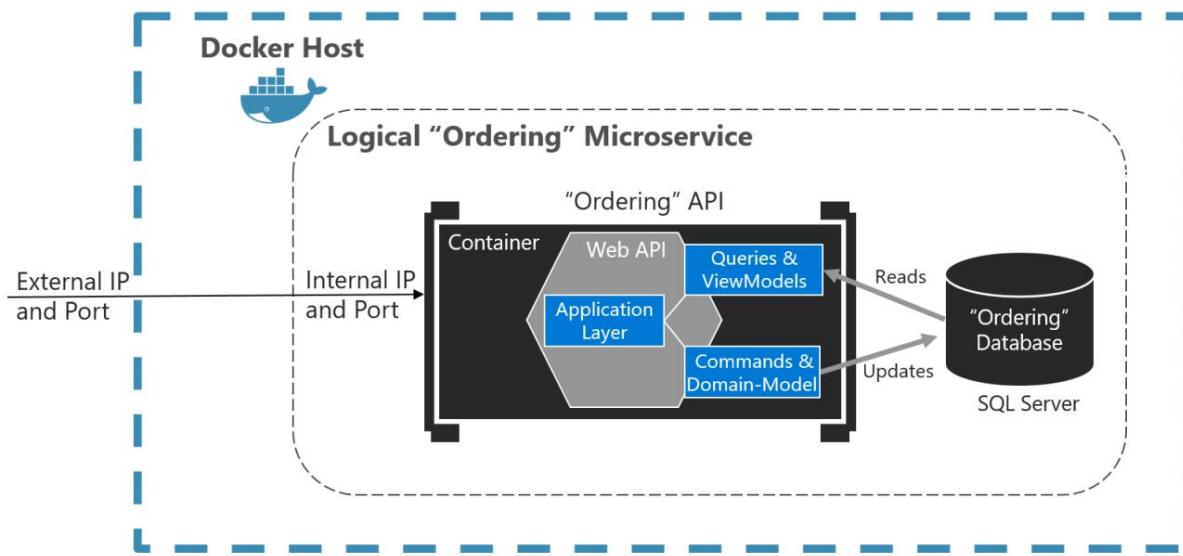


Figura 7-2. Microservicio simplificado basado en CQRS y DDD

El microservicio de "pedido" lógico incluye su base de datos de pedidos, que puede ser, pero no tiene que ser, el mismo host de Docker. Tener la base de datos en el mismo host de Docker es bueno para el desarrollo, pero no para la producción.

La capa de aplicación puede ser la propia API web. El aspecto de diseño importante aquí es que el microservicio ha separado las consultas y los ViewModels (modelos de datos especialmente creados para las aplicaciones cliente) de los comandos, el modelo de dominio y las transacciones siguiendo el patrón CQRS. Este enfoque mantiene las consultas independientes de las restricciones y restricciones provenientes de patrones DDD que solo tienen sentido para transacciones y actualizaciones, como se explica en secciones posteriores.

Recursos adicionales

- Greg jóven. Control de versiones en un sistema basado en eventos (libro electrónico gratuito para leer en línea) <https://leanpub.com/esversioning/read>

Aplicar enfoques 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 consiste simplemente en separar las consultas de los comandos y utilizar 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. En otras palabras, las consultas no tienen efectos secundarios.

Por lo tanto, podría usar un modelo de datos de "lectura" diferente al modelo de dominio de "escritura" de lógica transaccional, aunque los microservicios de pedido estén usando la misma base de datos. Por lo tanto, este es un enfoque CQRS simplificado.

Por otro lado, los comandos, que desencadenan transacciones y actualizaciones de datos, cambian de estado en el sistema. Con los comandos, debe tener cuidado al tratar con la complejidad y las reglas comerciales en constante cambio. Aquí es donde desea aplicar técnicas 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 brindan beneficios como una mayor calidad con el tiempo, especialmente en los comandos y otros códigos que modifican el estado del sistema. Sin embargo, esas restricciones agregan complejidad con menos beneficios para leer y consultar datos.

Uno de esos patrones es el patrón Agregado, que examinaremos con más detalle en secciones posteriores. Brevemente, en el patrón Agregado, trata muchos objetos de dominio como una sola unidad como resultado de su relación en el dominio. Es posible que no siempre obtenga ventajas de este patrón en las consultas; puede aumentar la complejidad de la lógica de consulta. Para las consultas de solo lectura, no obtiene las ventajas de tratar varios objetos como un único agregado. Solo obtienes la complejidad.

Como se muestra en la Figura 7-2 en la sección anterior, esta guía sugiere usar patrones DDD solo en el área transaccional/de actualizaciones de su microservicio (es decir, según lo activen 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 consultas", puede elegir entre muchos enfoques, desde su ORM completo como EF Core, proyecciones de AutoMapper, procedimientos almacenados, vistas, vistas materializadas o un micro ORM.

En esta guía y en eShopOnContainers (específicamente el microservicio de pedidos) elegimos implementar consultas directas utilizando un micro ORM como [Dapper](#). Esta guía le permite implementar cualquier consulta basada en declaraciones SQL para obtener el mejor rendimiento, gracias a un marco ligero con poca sobrecarga.

Cuando usa este enfoque, cualquier actualización de su modelo que afecte la forma en que las entidades se conservan en una base de datos SQL también necesita actualizaciones separadas para las consultas SQL utilizadas por Dapper o cualquier otro enfoque separado (no EF) para realizar consultas.

Los patrones CQRS y DDD no son arquitecturas de nivel superior

Es importante comprender 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 solo patrones arquitectónicos. Los microservicios, SOA y la arquitectura basada en eventos (EDA) son ejemplos de estilos arquitectónicos. Describen un sistema de muchos componentes, como muchos microservicios. Los patrones CQRS y DDD describen algo dentro de un solo sistema o componente; en este caso, algo dentro de un microservicio.

Diferentes contextos delimitados (BC) emplearán diferentes patrones. Tienen diferentes responsabilidades, y eso conduce a diferentes soluciones. Vale la pena enfatizar que forzar el mismo patrón en todas partes conduce al fracaso. No utilice patrones CQRS y DDD en todas partes. Muchos subsistemas, BC o microservicios son más simples y se pueden implementar más fácilmente mediante servicios CRUD simples u otro enfoque.

Solo hay una arquitectura de aplicación: la arquitectura del sistema o la aplicación de extremo a extremo 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 propias compensaciones y decisiones de diseño interno a nivel de patrones de arquitectura. No intente aplicar los mismos patrones arquitectónicos que CQRS o DDD en todas partes.

Recursos adicionales

- Martín Fowler. CQRS <https://martinfowler.com/bliki/CQRS.html>
- Greg jóven. Documentos CQRS https://cqrsrcs.files.wordpress.com/2010/11/cqrs_documents.pdf
- Udi Dahan. CQRS aclarado <https://udidahan.com/2009/12/09/clarified-cqrs/>

Implementar lecturas/consultas en un microservicio CQRS

Para lecturas/consultas, el microservicio de pedidos de la aplicación de referencia eShopOnContainers implementa las consultas independientemente del modelo DDD y el área transaccional. Esta implementación se realizó principalmente porque las demandas de consultas y transacciones son drásticamente diferentes. Las escrituras ejecutan transacciones que deben cumplir con la lógica del dominio.

Las consultas, por otro lado, son idempotentes y se pueden segregar de las reglas del dominio.

El enfoque es simple, como se muestra en la Figura 7-3. La interfaz de la API se implementa mediante los controladores de la API web que utilizan cualquier infraestructura, como un mapeador relacional de microobjetos (ORM) como Dapper, y devuelve modelos de vista dinámicos según las necesidades de las aplicaciones de la interfaz de usuario.

High level “Queries-side” in a simplified CQRS

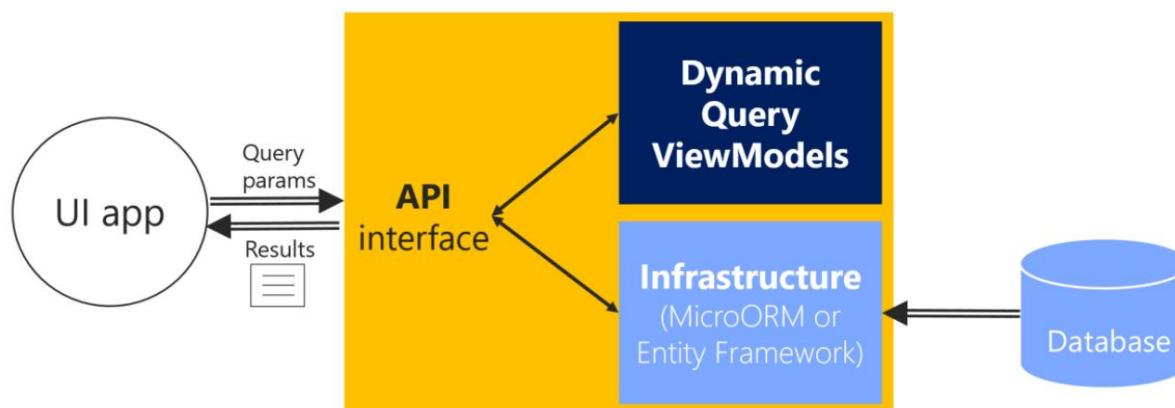


Figura 7-3. El enfoque más simple para consultas en un microservicio CQRS

El enfoque más simple para el lado de las consultas en un enfoque CQRS simplificado se puede implementar consultando la base de datos con un Micro-ORM como Dapper, que devuelve ViewModels dinámicos. La consulta

Las definiciones consultan la base de datos y devuelven un ViewModel dinámico creado sobre la marcha para cada consulta. Dado que las consultas son idempotentes, no cambiarán los datos sin importar cuántas veces ejecute una consulta. Por lo tanto, no necesita estar restringido por ningún patrón DDD utilizado en el lado transaccional, como agregados y otros patrones, y es por eso que las consultas están separadas del área transaccional. Consulta la base de datos para obtener los datos que necesita la interfaz de usuario y devuelve un modelo de vista dinámico que no necesita definirse estáticamente en ningún lugar (no hay clases para los modelos de vista), excepto en las propias declaraciones de SQL.

Dado que este enfoque es simple, el código requerido para 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 este enfoque. Las consultas se definen en el proyecto de microservicio Ordering.API dentro de la solución eShopOnContainers.

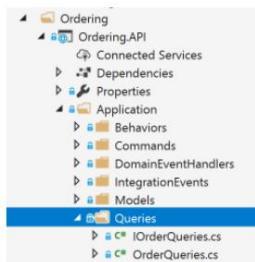


Figura 7-4. Consultas en el microservicio Pedidos en eShopOnContainers

[Use ViewModels creados específicamente para aplicaciones cliente, independientemente 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 realizar específicamente para los clientes, en función de los datos devueltos por las consultas. Estos modelos, u objetos de transferencia de datos (DTO), se denominan ViewModels.

Los datos devueltos (ViewModel) pueden ser el resultado de unir datos de varias entidades o tablas en la base de datos, o incluso de varios agregados definidos en el modelo de dominio para el área transaccional.

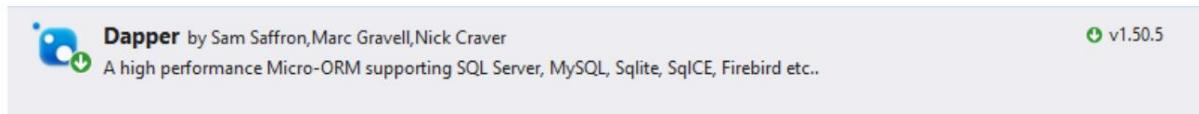
En este caso, debido a que está creando consultas independientes del modelo de dominio, los límites y las restricciones de los agregados se ignoran y puede consultar cualquier tabla y columna que pueda necesitar. Este enfoque proporciona una gran flexibilidad y productividad para los desarrolladores que crean o actualizan las consultas.

Los ViewModels pueden ser tipos estáticos definidos en clases (como se implementa en el microservicio de pedidos). O pueden crearse dinámicamente en función de las consultas realizadas, lo que es ágil para los desarrolladores.

[Use Dapper como un micro ORM para realizar consultas](#)

Puede usar cualquier micro ORM, Entity Framework Core o incluso ADO.NET simple para realizar consultas. En la aplicación de muestra, se seleccionó a Dapper para el microservicio de pedidos en eShopOnContainers como un buen ejemplo de un micro ORM popular. Puede ejecutar consultas SQL simples con un gran rendimiento, porque es un marco ligero. Con Dapper, puede escribir una consulta SQL que puede acceder y unirse a varias tablas.

Dapper es un proyecto de código abierto (originalmente creado por Sam Saffron) y es parte de los componentes básicos utilizados en [Stack Overflow](#). Para usar Dapper, solo necesita instalarlo a través del paquete [Dapper NuGet](#), como se muestra en la siguiente figura:



También debe agregar una directiva de uso para que su código tenga acceso a los métodos de extensión de Dapper.

Cuando usa Dapper en su código, usa directamente la clase [SqlConnection](#) disponible en el espacio de nombres [System.Data.SqlClient](#). Mediante el método `QueryAsync` y otros métodos de extensión que amplían la clase [SqlConnection](#), puede ejecutar consultas de forma sencilla y eficaz.

ViewModels dinámicos versus estáticos

Al devolver ViewModels del lado del servidor a las aplicaciones cliente, puede pensar en esos ViewModels como DTO (Objetos de transferencia de datos) que pueden ser diferentes a las entidades de dominio internas de su modelo de entidad porque ViewModels almacena los datos de la manera que la aplicación cliente necesita. Por lo tanto, en muchos casos, puede agregar datos provenientes de múltiples entidades de dominio y componer los ViewModels precisamente de acuerdo con cómo la aplicación cliente necesita esos datos.

Esos ViewModels o DTO se pueden definir explícitamente (como clases de soporte de datos), como la clase `OrderSummary` que se muestra en un fragmento de código posterior. O bien, podría simplemente devolver ViewModels dinámicos o DTO dinámicos en función de los atributos devueltos por sus consultas como un tipo dinámico.

ViewModel como tipo dinámico

Como se muestra en el siguiente código, las consultas pueden devolver directamente un modelo de vista simplemente devolviendo un tipo dinámico que internamente se basa en los atributos devueltos por una consulta. Eso significa que el subconjunto de atributos que se devolverá se basa en la consulta misma. Por lo tanto, si agrega una nueva columna a la consulta o combinación, esos datos se agregan dinámicamente al ViewModel devuelto.

```
usando apuesto;
utilizando Microsoft.Extensions.Configuration; utilizando
System.Data.SqlClient; utilizando System.Threading.Tasks;
utilizando System.Dynamic; usando
System.Collections.Generic;

clase pública OrderQueries : IOrderQueries {

    Tarea asíncrona pública <IEnumerable<dinámica>> GetOrdersAsync() {

        usando (var conexión = nueva SqlConnection(_connectionString)) {

            conexión.Abrir(); return
            await connection.QueryAsync<dynamic>(@"SELECT o.[Id]
                as ordernumber, o.[OrderDate] as [date], o.[Name] as
                [status], SUM(oi.units * oi.unitprice) como total DESDE [pedido].
                [Pedidos] o LEFT JOIN[pedido].[artículos del pedido] 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 ensambla dinámicamente como ViewModel.

Pros: este enfoque reduce la necesidad de modificar las clases estáticas de ViewModel cada vez que actualiza la oración SQL de una consulta, lo que hace que este enfoque de diseño sea ágil al codificar, directo y rápido para evolucionar con respecto a cambios futuros.

Contras: a largo plazo, los tipos dinámicos pueden afectar negativamente la claridad y la compatibilidad de un servicio con las aplicaciones de los clientes. Además, el software de middleware como Swashbuckle no puede proporcionar el mismo nivel de documentación sobre los tipos devueltos si se utilizan tipos dinámicos.

ViewModel como clases DTO predefinidas

Pros: tener clases ViewModel estáticas y predefinidas, como "contratos" basados en clases DTO explícitas, definitivamente es mejor para las API públicas, pero también para los microservicios a largo plazo, incluso si solo los usa la misma aplicación.

Si desea especificar tipos de respuesta para Swagger, debe usar clases DTO explícitas como tipo de retorno. Por lo tanto, las clases DTO predefinidas le permiten ofrecer información más rica de Swagger. Eso mejora la documentación y la compatibilidad de la API cuando se consume una API.

Contras: como se mencionó anteriormente, al actualizar 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 Pedidos en eShopOnContainers, comenzamos a desarrollar utilizando ViewModels dinámicos, ya que era sencillo y ágil en las primeras etapas de desarrollo. Pero, una vez que se estabilizó el desarrollo, optamos por refactorizar las API y usar DTO estáticos o predefinidos para los ViewModels, porque es más claro para los consumidores del microservicio conocer tipos de DTO explícitos, utilizados como "contratos".

En el siguiente ejemplo, puede ver cómo la consulta devuelve datos mediante el uso de una clase DTO de ViewModel explícita: la clase OrderSummary.

```

usando apueste;
utilizando Microsoft.Extensions.Configuration;
utilizando System.Data.SqlClient; utilizando
System.Threading.Tasks; utilizando System.Dynamic;
usando System.Collections.Generic;

clase pública OrderQueries : IOrderQueries {

    Tarea asíncrona pública <IEnumerable<OrderSummary>> GetOrdersAsync()
    {
        usando (var conexión = nueva SqlConnection(_connectionString)) {

            conexión.Abrir();
            regresar en espera de
            conexión.QueryAsync<OrderSummary>(@"SELECT o.[Id] as ordernumber,

```

```

        o.[FechaPedido] como [fecha],os.[Nombre] como [estado],
        SUM(oi.unidades*oi.precio unitario) como total DESDE [pedido].
        [Pedidos] o LEFT JOIN[pedido].[artículos] oi ON o.Id = oi.orderid
        LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId = os.Id GROUP BY o.[Id],
        o.[OrderDate], os.[Name]

        ORDEN POR o.[Id]"");
    }
}

```

Describir los tipos de respuesta de las API web

Los desarrolladores que consumen API 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). Los tipos de respuesta se manejan en los comentarios XML y las anotaciones de datos.

Si la documentación adecuada en la interfaz de usuario de Swagger, el consumidor no sabe qué tipos se devuelven o qué códigos HTTP se pueden devolver. Ese problema se soluciona agregando

[Microsoft.AspNetCore.Mvc.ProducesResponseTypeAttribute](#), de modo que Swashbuckle pueda generar información más detallada sobre el modelo de retorno y los valores de la API, como se muestra en el siguiente código:

```
espacio de nombres Microsoft.eShopOnContainers.Services.Ordering.API.Controllers {  
  
    [Ruta("api/v1/[controlador]")]  
    [Autorizar]  
    clase pública OrdersController : Controller {  
  
        //Código adicional...  
        [Ruta("")]  
        [HttpGet]  
        [ProducesResponseType(typeof(IEnumerable<OrderSummary>),  
            (int) HttpStatusCode.OK)] public async Task<IActionResult> GetOrders() {  
  
            var idusuario = _identityService.GetUserIdentity(); var pedidos =  
            esperar _orderQueries .GetOrdersFromUserAsync(Guid.Parse(userid));  
            volver Ok(pedidos);  
  
        }  
    }  
}
```

Sin embargo, el `atributo ProducesResponseType` no puede usar dinámico como tipo, sino que requiere el uso de tipos explícitos, como `OrderSummary` `ViewModel` `DTO`, que se muestra en el siguiente ejemplo:

```
resumen de pedido de clase pública
{
    public int numero de pedido { get; colocar; } Fecha
    fecha y hora pública { obtener; colocar; } estado de
    la cadena pública { get; colocar; } público doble total
    { obtener; colocar; }
}
```

Esta es otra razón por la que los tipos devueltos explícitos son mejores que los tipos dinámicos, a largo plazo.

Al usar 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 de `ResponseType`.

The screenshot shows the Swagger UI for the Ordering HTTP API. At the top, there's a navigation bar with the Swagger logo, a dropdown for 'Select a spec' set to 'Ordering API V1', and an 'Authorize' button with a lock icon.

The main content area is titled 'Orders'. It lists four operations:

- PUT** /api/v1/orders/cancel
- PUT** /api/v1/orders/ship
- GET** /api/v1/orders/{orderId}
- GET** /api/v1/orders

For the **GET /api/v1/orders** operation, the details are expanded:

- Parameters**: None
- Responses**:
 - Code**: 200 **Description**: Success
 - Example Value**: Model


```
[{"ordernumber": 0, "date": "2018-08-10T23:21:58.423Z", "status": "string", "total": 0}]
```
- Code**: 401 **Description**: Unauthorized
- Code**: 403 **Description**: Forbidden

Figura 7-5. Interfaz de usuario de Swagger que muestra tipos de respuesta y posibles códigos de estado HTTP de una API web

La imagen muestra algunos valores de ejemplo basados en los tipos de ViewModel y los posibles códigos de estado HTTP que se pueden devolver.

Recursos adicionales

- Elegante <https://github.com/StackExchange/dapper-dot-net>
- Julio Lerman. Puntos de datos: Dapper, Entity Framework y aplicaciones híbridas (artículo de la revista MSDN) <https://docs.microsoft.com/archive/msdn-magazine/2016/may/data-points-dapper-entity-framework-and-hybrid-apps>

- Páginas de ayuda de ASP.NET Core Web API usando Swagger
[https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger?
tabs=visual-studio](https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio)

Diseñe un microservicio orientado a DDD

El diseño basado en dominio (DDD) aboga por el modelado basado en la realidad del negocio como relevante para sus casos de uso. En el contexto de la creación de aplicaciones, DDD habla de problemas como dominios. Describe áreas de problemas independientes como contextos acotados (cada contexto acotado se correlaciona con un microservicio) y enfatiza un lenguaje común para hablar sobre estos problemas. También sugiere muchos conceptos y patrones técnicos, como entidades de dominio con modelos ricos (sin [modelo de dominio anémico](#)), [objetos de valor](#), [agregados](#) y [reglas](#) de raíz agregada (o entidad raíz) para respaldar la implementación interna. Esta sección presenta el diseño y la implementación de esos patrones internos.

A veces, estas reglas y patrones técnicos de DDD se perciben como obstáculos que tienen una curva de aprendizaje pronunciada para implementar los enfoques de DDD. Pero lo importante no son los patrones en sí mismos, sino organizar el código para que esté alineado con los problemas del negocio y usar los mismos términos comerciales (lenguaje ubicuo). Además, los enfoques DDD deben aplicarse solo si está implementando microservicios complejos con reglas comerciales significativas. Las responsabilidades más simples, como un servicio CRUD, se pueden administrar con enfoques más simples.

Dónde trazar los límites es la tarea clave al diseñar y definir un microservicio. Los patrones DDD lo ayudan a comprender la complejidad del dominio. Para el modelo de dominio de cada contexto acotado, usted identifica y define las entidades, los objetos de valor y los agregados que modelan su dominio.

Usted construye y refina un modelo de dominio que está contenido dentro de un límite que define su contexto. Y eso es explícito en forma de microservicio. Los componentes dentro de esos límites terminan siendo sus microservicios, aunque en algunos casos un BC o microservicios comerciales pueden estar compuestos por varios servicios físicos. DDD se trata de límites y también lo son los microservicios.

Mantenga los límites del contexto del microservicio relativamente pequeños

Determinar dónde colocar los límites entre los contextos delimitados equilibra dos objetivos contrapuestos. Primero, desea crear inicialmente los microservicios más pequeños posibles, aunque ese no debería ser el impulsor principal; debe crear un límite alrededor de las cosas que necesitan cohesión. En segundo lugar, desea evitar las comunicaciones comunicativas entre microservicios. Estos objetivos pueden contradecirse entre sí. Debe equilibrarlos descomponiendo el sistema en tantos microservicios pequeños como pueda hasta que vea que los límites de comunicación crecen rápidamente con cada intento adicional de separar un nuevo contexto delimitado. La cohesión es clave dentro de un solo contexto acotado.

Es similar al [olor del código de intimidad inapropiada](#) cuando se implementan clases. Si dos microservicios necesitan colaborar mucho entre sí, probablemente deberían ser el mismo microservicio.

Otra forma de ver este aspecto es la autonomía. Si un microservicio debe depender de otro servicio para atender directamente una solicitud, no es verdaderamente autónomo.

Capas en microservicios DDD

La mayoría de las aplicaciones empresariales con una complejidad comercial y técnica significativa están definidas por múltiples capas. Las capas son un artefacto lógico y no están relacionadas con la implementación del servicio. Existen para ayudar a los desarrolladores a gestionar la complejidad del código. Diferentes capas (como la capa del modelo de dominio frente a la capa de presentación, etc.) pueden tener diferentes tipos, lo que exige traducciones entre esos tipos.

Por ejemplo, se podría cargar una entidad desde la base de datos. Luego, parte de esa información, o una agregación de información que incluye datos adicionales de otras entidades, se puede enviar a la interfaz de usuario del cliente a través de una API web REST. El punto aquí es que la entidad de dominio está contenida dentro de la capa del modelo de dominio y no debe propagarse a otras áreas a las que no pertenece, como la capa de presentación.

Además, debe tener entidades siempre válidas (consulte la sección [Diseño de validaciones en la capa del modelo de dominio](#)) controladas por raíces agregadas (entidades raíz). Por lo tanto, las entidades no deben estar vinculadas a las vistas del cliente, ya que en el nivel de la interfaz de usuario es posible que aún no se validen algunos datos. Esta razón es para lo que sirve ViewModel. ViewModel es un modelo de datos exclusivamente para las necesidades de la capa de presentación. Las entidades de dominio no pertenecen directamente al 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 agregadas que garantice que todas las invariantes y reglas relacionadas con ese grupo de entidades (agregado) se realicen a través de un único punto de entrada o puerta, la raíz agregada.

La Figura 7-5 muestra cómo se implementa un diseño en capas en la aplicación eShopOnContainers.

Layers in a Domain-Driven Design Microservice

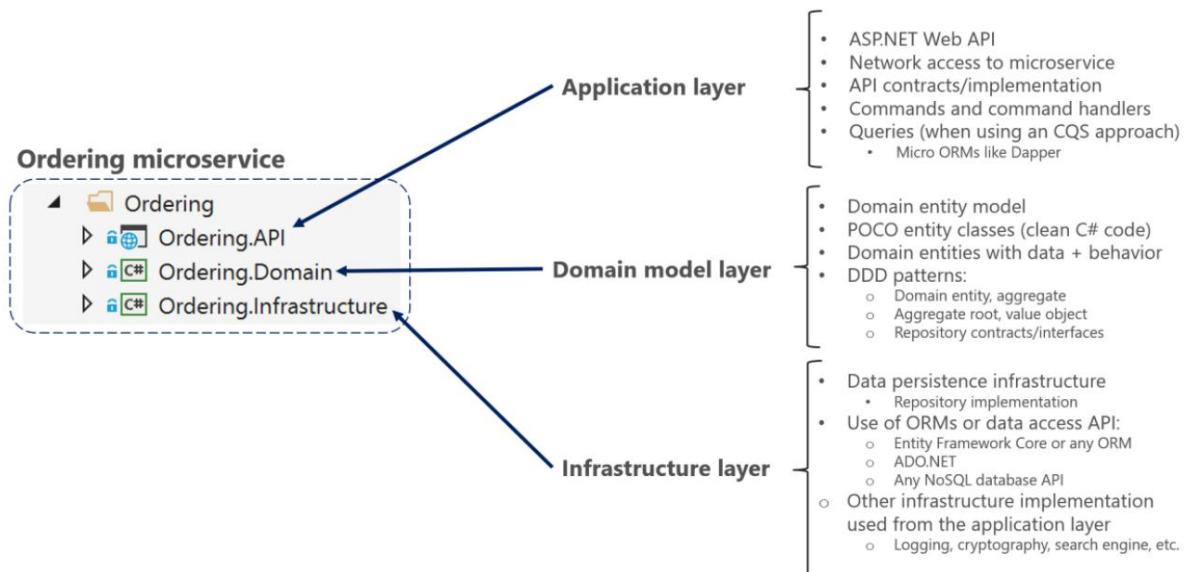


Figura 7-5. Capas DDD en el microservicio de pedidos en eShopOnContainers

Las tres capas en un microservicio DDD como Ordering. Cada capa es un proyecto VS: la capa de aplicación es Ordering.API, la capa de dominio es Ordering.Domain y la capa de infraestructura es Ordering.Infrastructure. Desea diseñar el sistema para que cada capa se comunique solo con ciertas otras capas. Ese enfoque puede ser más fácil de aplicar si las capas se implementan como bibliotecas de clases diferentes, porque puede identificar claramente qué dependencias se establecen entre las bibliotecas. Por ejemplo, la capa del modelo de dominio no debe depender de ninguna otra capa (las clases del modelo de dominio deben ser clases de objetos de clase antigua simple o **POCO**). Como se muestra en la Figura 7-6, la biblioteca de capas Ordering.Domain tiene dependencias solo en las bibliotecas .NET o paquetes NuGet, pero no en ninguna otra biblioteca personalizada, como la biblioteca de datos o la biblioteca de persistencia.

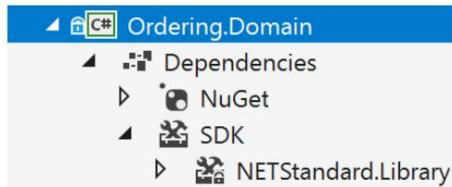


Figura 7-6. Las capas implementadas como bibliotecas permiten un mejor control de las dependencias entre capas

La capa del modelo de dominio

El excelente libro [Domain Driven Design](#) de Eric Evans dice lo siguiente sobre la capa de modelo de dominio y la capa de aplicación.

Capa de Modelo de Dominio: Responsable de representar conceptos del negocio, información sobre la situación del negocio y reglas de negocio. Aquí se controla y utiliza el estado que refleja la situación empresarial, aunque los detalles técnicos de almacenamiento se deleguen a la infraestructura. Esta capa es el corazón del software empresarial.

La capa del modelo de dominio es donde se expresa el negocio. Cuando implementa una capa de modelo de dominio de microservicio en .NET, esa capa se codifica como una biblioteca de clases con las entidades de dominio que capturan datos y comportamiento (métodos con lógica).

Siguiendo los [principios de Ignorancia de Persistencia](#) e [Ignorancia de 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 tener dependencias directas en la infraestructura, lo que significa que una regla importante es que las clases de entidad de su modelo de dominio deben ser POCO.

Las entidades de dominio no deben tener ninguna dependencia directa (como derivarse de una clase base) en ningún marco de infraestructura de acceso a datos como Entity Framework o NHibernate. Idealmente, las entidades de su dominio no deben derivar ni implementar ningún tipo definido en ningún marco de infraestructura.

La mayoría de los marcos ORM modernos, como Entity Framework Core, permiten este enfoque, por lo que las clases de su modelo de dominio no están acopladas a la infraestructura. Sin embargo, tener entidades POCO no siempre es posible cuando se usan ciertas bases de datos y marcos NoSQL, como Actors y Reliable Collections en Azure Service Fabric.

Incluso cuando es importante seguir el principio de Ignorancia de Persistencia para su modelo de Dominio, no debe ignorar las preocupaciones de persistencia. Todavía es importante comprender el modelo de datos físicos y cómo se asigna a su modelo de objeto de entidad. De lo contrario puedes crear diseños imposibles.

Además, este aspecto no significa que pueda tomar un modelo diseñado para una base de datos relacional y moverlo directamente a una base de datos NoSQL u orientada a documentos. En algunos modelos de entidades, el modelo puede encajar, pero normalmente no lo hace. Todavía existen restricciones que su modelo de entidad debe cumplir, basadas tanto en la tecnología de almacenamiento como en la tecnología ORM.

La capa de aplicación

Pasando a la capa de aplicación, podemos citar nuevamente el libro [Domain Driven Design de Eric Evans](#):

Capa de aplicación: define los trabajos que se supone que debe realizar el software y dirige los objetos del dominio expresivo para resolver los problemas. Las tareas de las que es responsable esta capa son significativas para el negocio o necesarias para la interacción con las capas de aplicación de otros sistemas. Esta capa se mantiene delgada. No contiene reglas comerciales ni conocimientos, sino que solo coordina tareas y delega trabajo a colaboraciones de objetos de dominio en la siguiente capa. No tiene un estado que refleje la situación comercial, pero puede tener un estado que refleje el progreso de una tarea para el usuario o el programa.

La capa de aplicación de un microservicio en .NET se suele codificar como un proyecto de API web de ASP.NET Core. 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 si se utiliza un enfoque CQRS, comandos aceptados por el microservicio e incluso la comunicación basada en eventos entre microservicios (eventos de integración). La API web de ASP.NET Core que representa la capa de la aplicación no debe contener reglas comerciales o conocimiento del dominio (especialmente reglas de dominio para transacciones o actualizaciones); estos deben ser propiedad de la biblioteca de clases del modelo de dominio. La capa de aplicación solo debe coordinar tareas y no debe contener ni definir ningún estado de dominio (modelo de dominio). Delega la ejecución de las reglas comerciales a las propias clases del modelo de dominio (raíces agregadas y entidades de dominio), que en última instancia actualizarán los datos dentro de esas entidades de dominio.

Básicamente, la lógica de la aplicación es donde implementa todos los casos de uso que dependen de una interfaz determinada. Por ejemplo, la implementación relacionada con un servicio Web API.

El objetivo es que la lógica de dominio en la capa del modelo de dominio, sus invariantes, el modelo de datos y las reglas comerciales relacionadas deben ser completamente independientes de las capas de presentación y aplicación. Sobre todo, la capa del modelo de dominio no debe depender directamente de ningún marco de infraestructura.

La capa de infraestructura

La capa de infraestructura es cómo los datos que se mantienen inicialmente en entidades de dominio (en la memoria) se conservan en bases de datos u otro almacén persistente. Un ejemplo es usar el código de Entity Framework Core para implementar las clases de patrón de Repository que usan un DBContext para conservar los datos en una base de datos relacional.

De acuerdo con los principios de [Ignorancia de Persistencia](#) e [Ignorancia de Infraestructura](#) mencionados anteriormente, la capa de infraestructura no debe “contaminar” la capa del modelo de dominio. Debe mantener las clases de entidad del modelo de dominio independientes de la infraestructura que usa para conservar los datos (EF o cualquier otro marco) al no depender mucho de los marcos. La biblioteca de clases de capas de su modelo de dominio debe tener solo su código de dominio, solo clases de entidades POCO que implementen el corazón de su software y estén completamente desvinculadas de las tecnologías de infraestructura.

Por lo tanto, sus capas o bibliotecas de clases y proyectos deberían depender en última instancia de su capa de modelo de dominio (biblioteca), no al revés, como se muestra en la Figura 7-7.

Dependencies between Layers in a Domain-Driven Design service

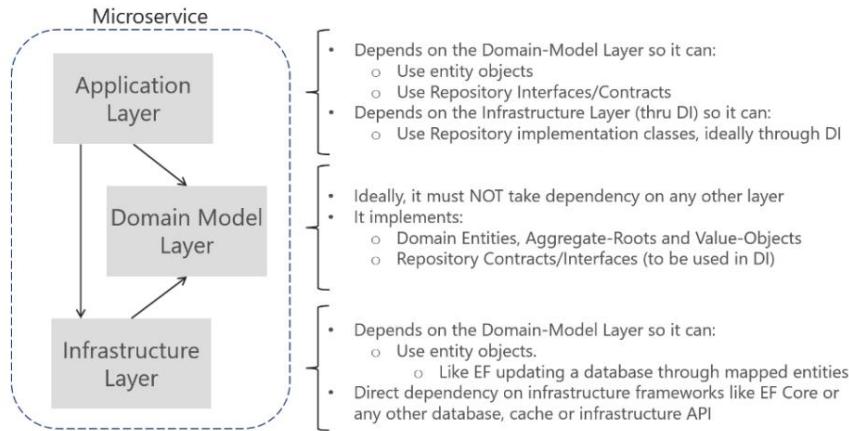


Figura 7-7. Dependencias entre capas en DDD

Dependencias en un Servicio DDD, la capa de Aplicación depende del Dominio y la Infraestructura, y la Infraestructura depende del Dominio, pero el Dominio no depende de ninguna capa. Este diseño de capa debe ser independiente para cada microservicio. Como se señaló anteriormente, puede implementar los microservicios más complejos siguiendo patrones DDD, mientras implementa microservicios basados en datos más simples (CRUD simple en una sola capa) de una manera más simple.

Recursos adicionales

- DevIQ. Principio de ignorancia de persistencia <https://deviq.com/persistence-ignorance/>
- Oren Eini. Ignorancia de Infraestructura <https://ayende.com/blog/3137/infrastructure-ignorance>
- Ángel López. Arquitectura en capas en diseño basado en dominios <https://ajlopez.wordpress.com/2008/09/12/architecture-layered-in-domain-driven-design/>

Diseñar un modelo de dominio de microservicio

Defina un modelo de dominio enriquecido para cada microservicio comercial o contexto acotado.

Su objetivo es crear un único modelo de dominio cohesivo para cada microservicio comercial o contexto acotado (BC). Tenga en cuenta, sin embargo, que un BC o microservicio comercial a veces puede estar compuesto por varios servicios físicos que comparten un solo modelo de dominio. El modelo de dominio debe capturar las reglas, el comportamiento, el lenguaje comercial y las restricciones del único contexto delimitado o microservicio comercial que representa.

El patrón de entidad de dominio

Las entidades representan objetos de dominio y se definen principalmente por su identidad, continuidad y persistencia en el tiempo, y no 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 de un modelo. Por lo tanto, debe identificarlos y diseñarlos con cuidado.

La identidad de una entidad puede cruzar varios microservicios o contextos acotados.

La misma identidad (es decir, el mismo valor de Id , aunque quizás no la misma entidad de dominio) se puede modelar en varios contextos delimitados o microservicios. Sin embargo, eso no implica que la misma entidad, con los mismos atributos y lógica, se implementaría en múltiples contextos acotados.

En cambio, las entidades en cada contexto acotado limitan sus atributos y comportamientos a los requeridos en el dominio de ese contexto acotado.

Por ejemplo, la entidad compradora puede tener la mayoría de los atributos de una persona que están definidos en la entidad de usuario en el perfil o el microservicio de identidad, incluida la identidad. Pero la entidad compradora en el microservicio de pedidos puede tener menos atributos, porque solo ciertos datos del comprador están relacionados con el proceso de pedido. El contexto de cada microservicio o Bounded Context impacta 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 entidad de pedido, debe tener operaciones y lógica empresarial implementadas como métodos para tareas como agregar un artículo de pedido, validación de datos y cálculo total. Los métodos de la entidad se encargan de las invariantes y las reglas de la entidad en lugar de que esas reglas se extiendan por la capa de aplicación.

La figura 7-8 muestra una entidad de dominio que implementa no solo atributos de datos, sino también operaciones o métodos con lógica de dominio relacionada.

Domain Entity pattern

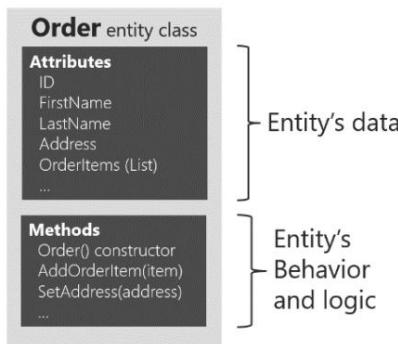


Figura 7-8. Ejemplo de un diseño de entidad de dominio que implementa datos más comportamiento

Una entidad modelo de dominio implementa comportamientos a través de métodos, es decir, no es un modelo "anémico". Por supuesto, a veces puede tener entidades que no implementen ninguna lógica como parte de la clase de entidad.

Esto puede ocurrir en entidades secundarias dentro de un agregado si la entidad secundaria no tiene ninguna lógica especial porque la mayor parte de la lógica se define en la raíz del agregado. Si tiene un microservicio complejo que

tiene lógica implementada en las clases de servicio en lugar de en las entidades de dominio, podría estar cayendo en el modelo de dominio anémico, explicado en la siguiente sección.

Modelo de dominio rico versus modelo de dominio anémico

En su publicación [AnemicDomainModel](#), Martin Fowler describe un modelo de dominio anémico de esta manera:

El síntoma básico de un modelo de dominio anémico es que, a primera vista, parece real. Hay objetos, muchos de los cuales llevan el nombre de los sustantivos en el espacio del dominio, y estos objetos están conectados con las ricas relaciones y estructura que tienen los verdaderos modelos de dominio. El truco viene cuando observas el comportamiento y te das cuenta de que casi no hay comportamiento en estos objetos, lo que los convierte en poco más que bolsas de getters y setters.

Por supuesto, cuando utiliza un modelo de dominio anémico, esos modelos de datos se utilizarán a partir de un conjunto de objetos de servicio (tradicionalmente denominado capa empresarial) que capturan todo el dominio o la lógica empresarial. La capa empresarial se encuentra sobre el modelo de datos y utiliza el modelo de datos como datos.

El modelo de dominio anémico es solo un diseño de estilo procedimental. Los objetos de entidad anémicos no son objetos reales porque carecen de comportamiento (métodos). Solo contienen propiedades de datos y, por lo tanto, no es un diseño orientado a objetos. Al colocar todo el comportamiento en los objetos de servicio (la capa empresarial), esencialmente termina con un [código de espagueti](#) o [scripts de transacción](#) y, por lo tanto, pierde las ventajas que proporciona un modelo de dominio.

Independientemente, si su microservicio o Bounded Context es muy simple (un servicio CRUD), el modelo de dominio anémico en forma de objetos de entidad con solo propiedades de datos podría ser lo suficientemente bueno, y podría no valer la pena implementar patrones DDD más complejos. En ese caso, será simplemente un modelo de persistencia, porque ha creado intencionalmente una entidad con solo datos para CRUD.

propósitos

Es por eso que las arquitecturas de microservicios son perfectas para un enfoque multiarquitectónico dependiendo 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 estés 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 antipatrón. Sin embargo, si necesita abordar la complejidad del dominio de un microservicio que tiene muchas reglas comerciales en constante cambio, el modelo de dominio anémico podría ser un antipatrón para ese microservicio o contexto delimitado. En ese caso, diseñarlo como un modelo enriquecido con entidades que contienen datos y comportamiento, además de implementar patrones DDD adicionales (agregados, objetos de valor, etc.) podría tener enormes beneficios para el éxito a largo plazo de dicho microservicio.

Recursos adicionales

- DevIQ. Entidad de dominio
<https://deviq.com/entity/>
- Martín Fowler. El modelo de dominio
<https://martinfowler.com/eaaCatalog/domainModel.html>

- Martín Fowler. El modelo de dominio anémico
<https://martinfowler.com/bliki/AnemicDomainModel.html>
-

El patrón de objeto de valor

Como ha señalado Eric Evans, “Muchos objetos no tienen identidad conceptual. Estos objetos describen ciertas características de una cosa”.

Una entidad requiere una identidad, pero hay muchos objetos en un sistema que no la requieren, como el patrón de objeto de valor. Un objeto de valor es un objeto sin identidad conceptual que describe un aspecto del dominio.

Estos son objetos que usted instancia para representar elementos de diseño que solo le conciernen temporalmente.

Te preocupas por lo que son, no por quiénes son. Los ejemplos incluyen números y cadenas, pero también pueden ser conceptos de nivel superior como grupos de atributos.

Algo que es una entidad en un microservicio podría no ser una entidad en otro microservicio porque, en el segundo caso, el contexto acotado podría tener un significado diferente. Por ejemplo, una dirección en una aplicación de comercio electrónico podría no tener ninguna identidad, ya que podría representar solo un grupo de atributos del perfil del cliente para una persona o empresa. En este caso, la dirección debe clasificarse como un objeto de valor. Sin embargo, en una solicitud para una empresa de servicios públicos de energía eléctrica, la dirección del cliente podría ser importante para el dominio comercial. Por lo tanto, la dirección debe tener una identidad para que el sistema de facturación pueda vincularse directamente a la dirección. En ese caso, una dirección debe clasificarse como una entidad de dominio.

Una persona con nombre y apellido suele ser una entidad porque una persona tiene identidad, incluso si el nombre y el apellido coinciden con otro conjunto de valores, como si esos nombres también se refieren a un diferente persona.

Los objetos de valor son difíciles de administrar en bases de datos relacionales y ORM como Entity Framework (EF), mientras que en bases de datos orientadas a documentos son más fáciles de implementar y usar.

EF Core 2.0 y versiones posteriores incluyen la característica de entidades propias que facilita el manejo del valor objetos, como veremos en detalle más adelante.

Recursos adicionales

- Martín Fowler. Patrón de objeto de valor <https://martinfowler.com/bliki/ValueObject.html>
 - Objeto de valor
<https://deviq.com/value-object/>
 - Objetos de valor en el desarrollo basado en pruebas <https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects>
 - Erick Evans. Diseño impulsado por dominio: abordar la complejidad en el corazón del software. (Libro; incluye una discusión sobre objetos de valor) <https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>
-

El patrón agregado

Un modelo de dominio contiene grupos de diferentes entidades de datos y procesos que pueden controlar un área importante de funcionalidad, como el cumplimiento de pedidos o el inventario. Una unidad DDD más detallada es el agregado, que describe un grupo o grupo de entidades y comportamientos que pueden tratarse como una unidad cohesiva.

Por lo general, define un agregado en función de las transacciones que necesita. Un ejemplo clásico es un pedido que también contiene una lista de artículos de pedido. Un artículo de pedido normalmente será una entidad. Pero será una entidad secundaria dentro del agregado de la orden, que también contendrá la entidad de la orden como su entidad raíz, normalmente denominada raíz agregada.

Identificar agregados puede ser difícil. Un agregado es un grupo de objetos que deben ser coherentes entre sí, pero no puede simplemente elegir 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 transaccionalmente consistentes son las que forman un agregado. Pensar en las operaciones de transacción es probablemente la mejor manera de identificar los agregados.

El patrón Agregado Raíz o Entidad Raíz

Un agregado está compuesto por al menos una entidad: la raíz del agregado, también llamada entidad raíz o entidad primaria. Además, puede tener múltiples entidades secundarias y objetos de valor, con todas las entidades y objetos trabajando juntos para implementar el comportamiento y las transacciones requeridas.

El propósito de una raíz agregada es asegurar 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 raíz del agregado.

Debe realizar cambios en las entidades dentro del agregado solo a través de la raíz del agregado. Es el guardián de la consistencia del agregado, considerando todas las invariantes y reglas de consistencia que podría necesitar cumplir en su agregado. Si cambia una entidad secundaria o un objeto de valor de forma independiente, la raíz agregada no puede garantizar que el agregado esté en un estado válido. Sería como una mesa con una pata suelta. Mantener la consistencia es el objetivo principal de la raíz agregada.

En la Figura 7-9, puede ver agregados de muestra como el agregado de comprador, que contiene una sola entidad (el comprador raíz agregado). El pedido agregado contiene varias entidades y un objeto de valor.

Aggregate pattern

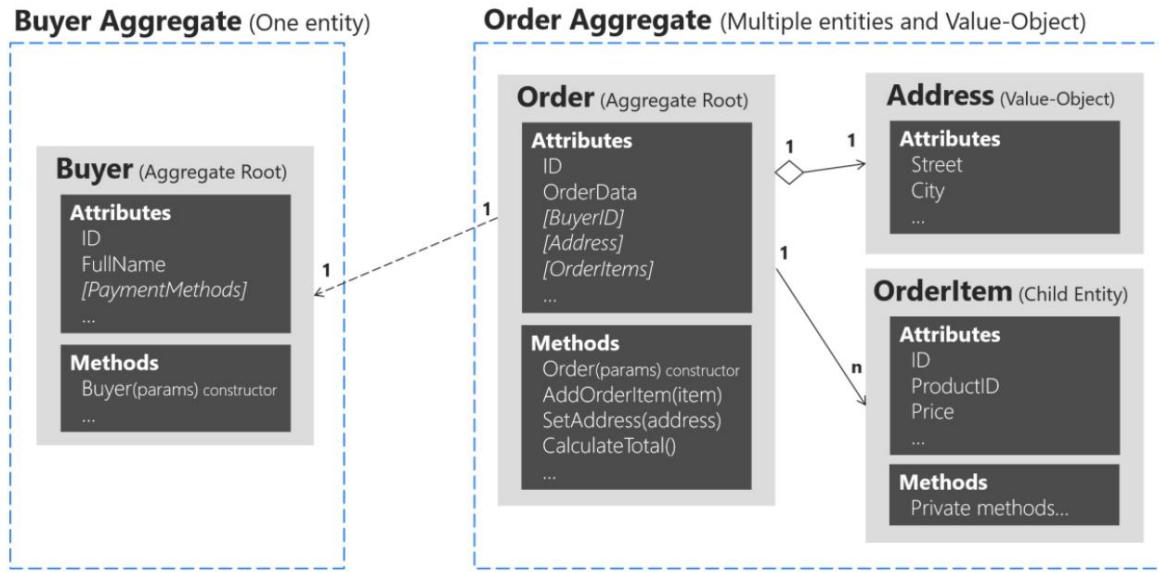


Figura 7-9. Ejemplo de agregados con entidades múltiples o únicas

Un modelo de dominio DDD se compone de agregados, un agregado puede tener solo una entidad o más, y también puede incluir objetos de valor. Tenga en cuenta que el agregado de compradores podría tener entidades secundarias adicionales, según su dominio, como lo hace en el microservicio de pedidos en la aplicación de referencia eShopOnContainers. La figura 7-9 solo ilustra un caso en el que el comprador tiene una sola entidad, como ejemplo de un agregado que contiene solo una raíz agregada.

Para mantener la separación de los agregados y mantener límites claros entre ellos, es una buena práctica en un modelo de dominio DDD no permitir la navegación directa entre los agregados y solo tener el campo de clave externa (FK), como se implementó en el [modelo de dominio de microservicio de pedido](#) en eShopEnContenedores. La entidad Pedido solo tiene un campo de clave externa para el comprador, pero no una propiedad de navegación de EF Core, como se muestra en el siguiente código:

```
Orden de clase pública : Entidad, IAggregateRoot {
    DateTime privado _orderDate;
    Dirección pública Dirección { get; conjunto privado; } int
    privado ? _IdComprador; // FK que apunta a una raíz agregada diferente public OrderStatus
    OrderStatus { get; conjunto privado; } Lista privada de solo lectura <Artículo de pedido>
    _artículos de pedido; public IReadOnlyCollection<OrderItem> OrderItems => _orderItems; // ...
    código adicional
}
```

Identificar y trabajar con agregados requiere investigación y experiencia. Para obtener más información, consulte la siguiente lista de recursos adicionales.

Recursos adicionales

- Vaughn Vernon. Diseño efectivo de agregados - Parte I: Modelado de un solo agregado (de <https://dddcommunity.org/>) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf
- Vaughn Vernon. Diseño efectivo de agregados - Parte II: Hacer que los agregados trabajen juntos (de <https://dddcommunity.org/>) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- Vaughn Vernon. Diseño Agregado Efectivo - Parte III: Obtener Perspectiva a Través del Descubrimiento (de <https://dddcommunity.org/>) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_3.pdf
- Serguéi Grybniak. Patrones de diseño táctico DDD <https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part>
- Chris Richardson. Desarrollo de microservicios transaccionales mediante agregados <https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson>
- DevIQ. El patrón Agregado <https://deviq.com/aggregate-pattern/>

Implementar un modelo de dominio de microservicio con .NET

En la sección anterior, se explicaron los principios y patrones de diseño fundamentales para diseñar un modelo de dominio. Ahora es el momento de explorar posibles formas de implementar el modelo de dominio mediante el uso de .NET (código C# sin formato) y EF Core. Su modelo de dominio estará compuesto simplemente por su código. Tendrá solo los requisitos del modelo EF Core, pero no dependencias reales de EF. No debe tener dependencias estrictas ni referencias a EF Core ni a ningún otro ORM en su modelo de dominio.

Estructura del modelo de dominio en una biblioteca estándar de .NET personalizada

La organización de carpetas utilizada para la aplicación de referencia eShopOnContainers demuestra el modelo DDD para la aplicación. Es posible que encuentre que una organización de carpetas diferente comunica más claramente las opciones 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 de pedidos y el agregado de compradores. Cada agregado es un grupo de entidades de dominio y objetos de valor, aunque también podría tener un agregado compuesto por una sola entidad de dominio (la raíz agregada o la entidad raíz).

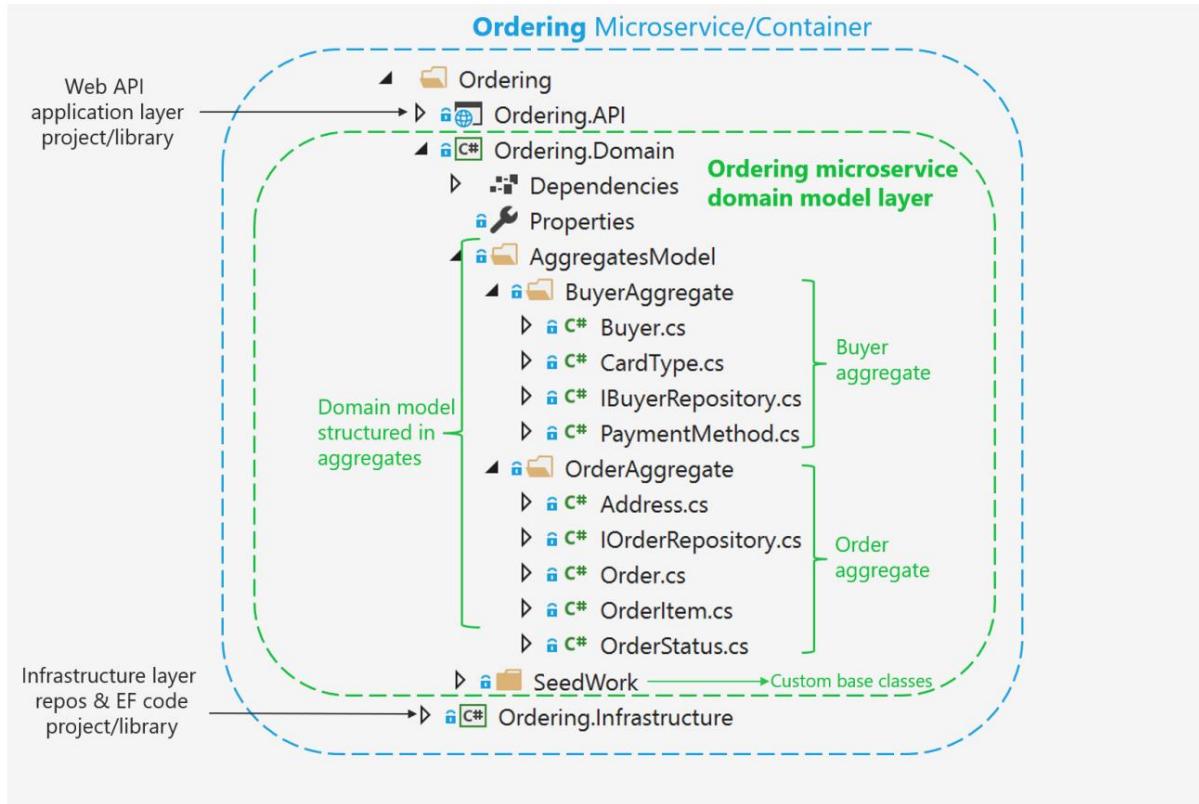


Figura 7-10. Estructura del modelo de dominio para el microservicio de pedidos en eShopOnContainers

Además, la capa del modelo de dominio incluye los contratos de repositorio (interfaces) que son los requisitos de infraestructura de su modelo de dominio. En otras palabras, estas interfaces expresan qué repositorios y los métodos que debe implementar la capa de infraestructura. Es fundamental que la implementación de los repositorios se coloque fuera de la capa del modelo de dominio, en la biblioteca de la capa de infraestructura, para que la capa del modelo de dominio no esté "contaminada" por API o clases de tecnologías de infraestructura, como Entity Framework.

También puede ver una [carpeta de SeedWork](#) que contiene clases base personalizadas que puede usar como base para sus entidades de dominio y objetos de valor, por lo que no tiene código redundante en la clase de objeto de cada dominio.

Estructurar agregados en una biblioteca estándar de .NET personalizada

Un agregado se refiere a un grupo de objetos de dominio agrupados para que coincidan con la coherencia transaccional. Esos objetos podrían ser instancias de entidades (una de las cuales es la raíz agregada o la entidad raíz) más cualquier objeto de valor adicional.

La consistencia transaccional significa que se garantiza que un agregado sea consistente y actualizado al final de una acción comercial. Por ejemplo, el agregado de pedidos del modelo de dominio de microservicios de pedidos de eShopOnContainers se compone como se muestra en la Figura 7-11.

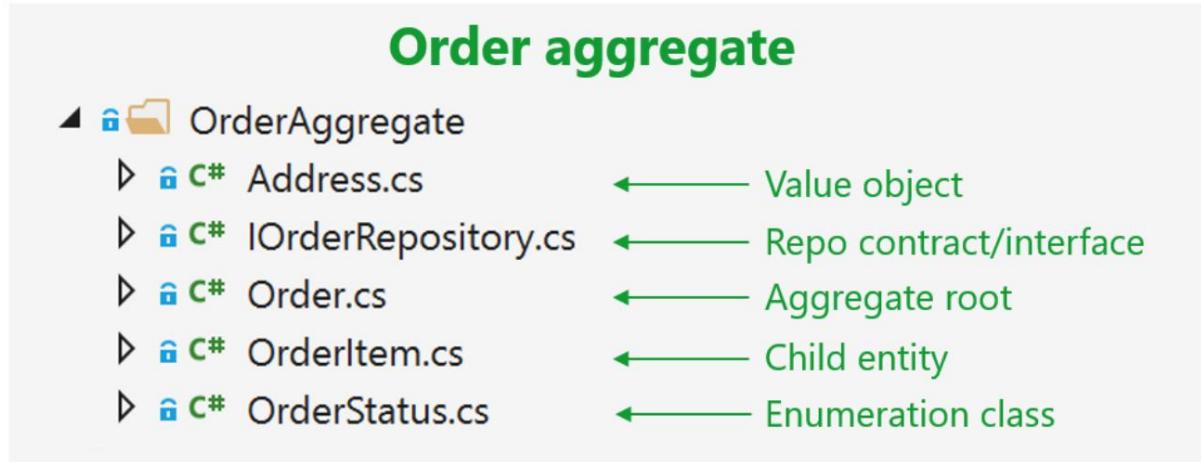


Figura 7-11. El pedido agregado en la solución de Visual Studio

Si abre cualquiera de los archivos en una carpeta agregada, puede ver cómo se marca como una clase o interfaz base personalizada, como una entidad o un objeto de valor, tal como se implementa en la carpeta [SeedWork](#).

Implementar entidades de dominio como clases POCO

Implementa un modelo de dominio en .NET mediante la creación de clases POCO que implementan sus entidades de dominio. En el siguiente ejemplo, la clase Order se define como una entidad y también como una raíz agregada. Debido a que la clase Order se deriva de la clase base Entity, puede reutilizar código común relacionado con entidades. Tenga en cuenta que estas clases base e interfaces las define usted en el proyecto del modelo de dominio, por lo que es su código, no el código de infraestructura de un ORM como EF.

```
// COMPATIBLE CON ENTITY FRAMEWORK CORE 5.0 // La
// entidad es una clase base personalizada con el ID de clase
public Orden: Entidad, IAggregateRoot {

    DateTime privado _orderDate; Dirección
    pública Dirección { get; conjunto privado; } int privado ?
    _IdComprador;

    public OrderStatus OrderStatus { get; conjunto privado; } privado int
    _orderStatusId;

    cadena privada _descripción; interno
    privado ? _IdMetodoDePago;

    lista privada de solo lectura <Artículo de pedido> _artículos de
    pedido; pública IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    pedido público (cadena de ID de usuario, dirección de dirección, int cardTypeId, string cardNumber, string cardSecurityNumber,
    cadena cardHolderName, DateTime cardExpiration, int? ID_comprador = nulo, int? IdMetodoPago = nulo) {

        _orderItems = new List<OrderItem>(); _IdComprador
        = IdComprador; _IdMetodoPago = IdMetodoPago;
        _orderStatusId = OrderStatus.Submitted.Id;
        _orderDate = DateTime.UtcNow;
    }
}
```

```

    Dirección = dirección;
    // ...Código adicional...
}

public void AddOrderItem(int productId, string productName, decimal unitPrice, decimal
    discount, string pictureUrl, int units = 1)

{
    ...
    // Reglas/lógica de dominio para agregar el artículo de pedido // ...

    var orderItem = new OrderItem(productId, productName, unitPrice, discount, pictureUrl, units);

    _orderItems.Add(orderItem);

} // ...
// Métodos adicionales con reglas/lógica de dominio relacionadas con el agregado de pedidos // ...

}

```

Es importante tener en cuenta que esta es una entidad de dominio implementada como una clase POCO. No tiene ninguna dependencia directa de Entity Framework Core ni de ningún otro marco de infraestructura. Esta implementación es como debería ser en DDD, solo código C# que implementa un modelo de dominio.

Además, la clase está decorada con una interfaz denominada `IAggregateRoot`. Esta interfaz es una interfaz vacía, a veces llamada interfaz de marcador, que se usa solo para indicar que esta clase de entidad también es una raíz agregada.

Una interfaz de marcador a veces se considera como un antipatrón; sin embargo, también es una forma limpia de marcar una clase, especialmente cuando esa interfaz podría estar evolucionando. Un atributo podría ser la otra opción para el marcador, pero es más rápido ver la clase base (`Entidad`) junto a la interfaz `IAggregate` en lugar de colocar un marcador de atributo `Aggregate` encima de la clase. Es una cuestión de preferencias, en cualquier caso.

Tener una raíz agregada significa que la mayor parte del código relacionado con la consistencia y las reglas comerciales de las entidades del agregado debe implementarse como métodos en la clase raíz agregada `Order` (por ejemplo, `AddOrderItem` al agregar un objeto `OrderItem` al agregado). No debe crear ni actualizar objetos `OrderItems` de forma independiente o directa; la clase `AggregateRoot` debe mantener el control y la coherencia de cualquier operación de actualización en sus entidades secundarias.

Encapsular datos en las Entidades de Dominio

Un problema común en los modelos de entidad es que exponen las propiedades de navegación de la colección como tipos de lista de acceso público. Esto permite que cualquier desarrollador colaborador manipule el contenido de estos tipos de colección, lo que puede pasar por alto reglas comerciales importantes relacionadas con la colección, lo que posiblemente deje el objeto en un estado no válido. La solución a esto es exponer el acceso de solo lectura a las colecciones relacionadas y proporcionar explícitamente métodos que definen las formas en que los clientes pueden manipularlos.

En el código anterior, tenga en cuenta que muchos atributos son de solo lectura o privados y solo los métodos de clase pueden actualizarlos, por lo que cualquier actualización considera las invariantes del dominio comercial y la lógica especificada dentro de los métodos de clase.

Por ejemplo, siguiendo los patrones DDD, no debe hacer lo siguiente desde ningún método de controlador de comandos o clase de capa de aplicación (en realidad, debería ser imposible para usted hacerlo):

```
// INCORRECTO SEGÚN LOS PATRONES DDD: CÓDIGO EN LA CAPA DE LA APLICACIÓN O //
CONTROLADORES DE COMANDOS // Código en los métodos del controlador de comandos o
controladores API web //... (INCORRECTO) Algunos códigos con lógica comercial fuera de las clases
de dominio...
OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName,
pictureUrl, unitPrice, descuento, unidades);

//... (INCORRECTO) Acceso a la colección OrderItems directamente desde la capa de aplicación // o controladores de comandos
myOrder.OrderItems.Add(myNewOrderItem); //...
```

En este caso, el método Add es puramente una operación para agregar datos, con acceso directo a la colección OrderItems. Por lo tanto, la mayor parte de la lógica, las reglas o las validaciones de dominio relacionadas con esa operación con las entidades secundarias se distribuirán en la capa de la aplicación (controladores de comandos y controladores de API web).

Si recorre la raíz agregada, la raíz agregada no puede garantizar sus invariantes, su validez o su consistencia. Eventualmente, tendrá un código de espagueti o un código de secuencia de comandos transaccional.

Para seguir los patrones DDD, las entidades no deben tener configuradores públicos en ninguna propiedad de la entidad. Los cambios en una entidad deben ser impulsados 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 artículos de pedido) deben ser propiedades de solo lectura (el método AsReadOnly se explica más adelante). Debería poder actualizarlo solo desde los métodos de la clase raíz agregada o los métodos de la entidad secundaria.

Como puede ver en el código para la raíz agregada de Order, todos los configuradores deben ser privados o al menos solo de lectura externa, de modo que cualquier operación contra los datos de la entidad o sus entidades secundarias debe realizarse a través de métodos en la clase de entidad. Esto mantiene la coherencia de forma controlada y orientada a objetos en lugar de implementar un código de secuencia de comandos transaccional.

El siguiente fragmento de código muestra la forma correcta de codificar la tarea de agregar un objeto OrderItem al agregado Order.

```
// CORRECTO SEGÚN DDD: CÓDIGO EN LA CAPA DE LA APLICACIÓN O EN LOS CONTROLADORES DE
COMANDOS // El código en los controladores de comandos o controladores WebAPI, relacionado solo con las cosas de la
aplicación // Aquí NO hay ningún código relacionado con la lógica empresarial del objeto OrderItem myOrder.AddOrderItem(productId ,
productName, pictureUrl, unitPrice, discount, units);

// El código relacionado con las validaciones de parámetros de OrderItem o reglas de dominio debe // estar
DENTRO del método AddOrderItem.

//...
```

En este fragmento, la mayoría de las validaciones o la lógica relacionada con la creación de un objeto OrderItem estarán bajo el control de la raíz del agregado Order (en el método AddOrderItem), especialmente las validaciones y la lógica relacionada con otros elementos del agregado. Por ejemplo, puede obtener el mismo artículo de producto como resultado de varias llamadas a AddOrderItem. En ese método, podría examinar los elementos del producto y consolidar los mismos elementos del producto en un solo objeto OrderItem con varias unidades.

Además, si hay diferentes montos de descuento pero el ID del producto es el mismo, es probable que aplique el descuento más alto. Este principio se aplica a cualquier otra lógica de dominio para el objeto OrderItem.

Además, la operación del nuevo artículo de pedido (parámetros) también será controlada y realizada por el método AddOrderItem desde la raíz del agregado de pedidos. Por lo tanto, la mayor parte de la lógica o validaciones relacionadas con esa operación (especialmente cualquier cosa que afecte la coherencia entre otras entidades secundarias) estará en un solo lugar dentro de la raíz agregada. Ese es el propósito final del patrón raíz agregado.

Cuando usa Entity Framework Core 1.1 o posterior, una entidad DDD se puede expresar mejor porque permite la [asignación a campos además de propiedades](#). Esto es útil cuando se protegen colecciones de entidades secundarias u objetos de valor. Con esta mejora, puede usar campos privados simples en lugar de propiedades y puede implementar cualquier actualización de la colección de campos en métodos públicos y proporcionar acceso de solo lectura a través del método AsReadOnly.

En DDD, desea actualizar la entidad solo 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 solo con un descriptor de acceso get. Las propiedades están respaldadas por campos privados. Solo se puede acceder a los miembros privados desde dentro de la clase. Sin embargo, hay una excepción: EF Core también necesita configurar estos campos (para que pueda devolver el objeto con los valores adecuados).

[Asigne propiedades con solo obtener accesores a los campos en la tabla de la base de datos](#)

La asignación de propiedades a las columnas de la tabla de la base de datos no es una responsabilidad del dominio, sino parte de la infraestructura y la capa de persistencia. Mencionamos esto aquí solo para que esté al tanto de las nuevas capacidades en EF Core 1.1 o versiones posteriores relacionadas con la forma en que puede modelar entidades. Detalles adicionales sobre este tema se explican en la sección de infraestructura y persistencia.

Cuando usa EF Core 1.0 o posterior, dentro de DbContext necesita asignar las propiedades que se definen solo con captadores a los campos reales en la tabla de la base de datos. Esto se hace con el método HasField de la clase PropertyBuilder.

[Asignar campos sin propiedades](#)

Con la característica de EF Core 1.1 o posterior para asignar columnas a campos, también es posible no usar propiedades.

En su lugar, solo puede asignar columnas de una tabla a campos. Un caso de uso común para esto son los campos privados para un estado interno al que no es necesario acceder desde fuera de la entidad.

Por ejemplo, en el ejemplo de código OrderAggregate anterior, hay varios campos privados, como el campo `_paymentMethodId`, que no tienen propiedades relacionadas ni para un setter ni para un getter. Ese campo también podría calcularse dentro de la lógica comercial del pedido y usarse a partir de los métodos del pedido, pero también debe conservarse en la base de datos. Entonces, en EF Core (desde v1.1) hay una forma de asignar 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. Modelado de agregados con DDD y Entity Framework. Tenga en cuenta que esto no es Entity Framework Core. <https://kalele.io/blog-posts/modeling-aggregates-with-ddd-and-entity-framework/>
- Julio Lerman. Puntos de datos: codificación para el diseño basado en dominios: consejos para desarrolladores centrados en datos [-para-desarrolladores-centrados-en-datos](#)
- Udi Dahan. Cómo crear modelos de dominio totalmente encapsulados <https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>

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

La carpeta de la solución contiene una carpeta de SeedWork . Esta carpeta contiene clases base personalizadas que puede usar como base para sus entidades de dominio y objetos de valor. Utilice estas clases base para no tener código redundante en la clase de objeto de cada dominio. La carpeta para este tipo de clases se llama SeedWork y no algo como Framework. Se llama SeedWork porque la carpeta contiene solo un pequeño subconjunto de clases reutilizables que en realidad no pueden considerarse un marco. Seedwork es un término introducido por [Michael Feathers](#) y popularizado por [Martin Fowler](#) , pero también podría llamar a esa carpeta [Common](#), [SharedKernel](#) o similar.

La Figura 7-12 muestra las clases que forman el trabajo inicial del modelo de dominio en el microservicio de pedidos. Tiene algunas clases base personalizadas como Entity, ValueObject y Enumeration, además de algunas interfaces. Estas interfaces (IRepository y IUnitOfWork) informan a la capa de infraestructura sobre lo que debe implementarse. Esas interfaces también se utilizan a través de la inyección de dependencias desde la capa de aplicación.

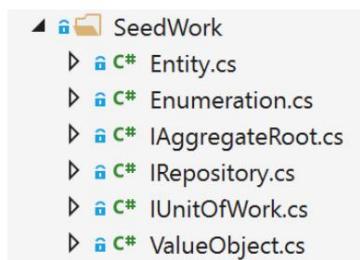


Figura 7-12. Un conjunto de muestras de interfaces y clases base de "trabajo inicial" del modelo de dominio

Este es el tipo de reutilización de copiar y pegar que muchos desarrolladores comparten entre proyectos, no un marco formal. Puede tener seedworks en cualquier capa o biblioteca. Sin embargo, si el conjunto de clases e interfaces es lo suficientemente grande, es posible que desee crear una biblioteca de una sola clase.

La clase base de Entidad personalizada

El siguiente código es un ejemplo de una clase base de Entidad en la que puede colocar código que cualquier entidad de dominio puede usar de la misma manera, como el ID de entidad, [los operadores de igualdad](#), una lista de eventos de dominio por entidad, etc.

```
// COMPATIBLE CON ENTITY FRAMEWORK CORE (1.1 y posteriores)
public abstract class Entity {

    ¿En t? _requestedHashCode; int
    _Id; lista privada <INotificación>
    _domainEvents; ID virtual virtual pública {

        conseguir
        {
            devolver _Id;

        } conjunto protegido
        {
            _Id = valor;
        }
    }

    public List<INotification> DomainEvents => _domainEvents; public void
    AddDomainEvent(INotification eventItem) {

        _dominioEventos = _dominioEventos ?? nueva Lista<INotificación>();
        _domainEvents.Add(elemento de evento);

    } public void RemoveDomainEvent(INotification eventItem) {

        if (_domainEvents es nulo) return;
        _domainEvents.Remove(elemento de evento);
    }

    public bool etransitorio() {

        devuelve esto. Id == predeterminado (Int32);
    }

    public override bool Equals(object obj) {

        if (obj == null || !(obj is Entity)) return false; if
            (Object.ReferenceEquals(this, obj)) devuelve
        verdadero; if (this.GetType() != obj.GetType())
        devuelve falso; Elemento de entidad =
        (Entidad)obj; if (item.IsTransient() || this.IsTransient())
        devuelve false; else devuelve item.Id == this.Id;

    }

    invalidación pública int GetHashCode() {

        si (!EsTransitorio())
    }
}
```

```

    {
        if (!_requestedHashCode.HasValue)
            _requestedHashCode = this.Id.GetHashCode() ^ 31; // XOR para
        distribución aleatoria. Consulte: // https://docs.microsoft.com/archive/
        blogs/ericlippert/guidelines-and-rules-for-gethashcode return _requestedHashCode.Value;

    } else
        return base.GetHashCode();

} Operador bool estático público == (Entidad a la izquierda, Entidad a la derecha)
{
    if (Object.Equals(left, null)) return
        (Object.Equals(right, null)); de lo contrario, regrese
        a la izquierda. Igual a (derecha);

} Operador bool estático público !=(Entidad a la izquierda, Entidad a la derecha)
{
    volver !(izquierda == derecha);
}
}

```

El código anterior que utiliza una lista de eventos de dominio por entidad se explicará en las siguientes secciones cuando se centre en los eventos de 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 propios repositorios, con código EF Core o cualquier otra dependencia de infraestructura y código (Linq, SQL, etc.), no deben implementarse dentro del modelo de dominio; los repositorios solo deben implementar las interfaces que defina en el modelo de dominio.

Un patrón relacionado con esta práctica (colocar las interfaces del repositorio en la capa del modelo de dominio) es el patrón de interfaz separada. Como [explicó Martin Fowler](#), “Use la interfaz separada para definir una interfaz en un paquete pero impleméntela en otro. De esta manera, un cliente que necesita la dependencia de la interfaz puede ignorar por completo la implementación”.

Seguir el patrón de interfaz separada permite que la capa de la aplicación (en este caso, el proyecto de API web para el microservicio) tenga una dependencia de los requisitos definidos en el modelo de dominio, pero no una dependencia directa de la capa de infraestructura/persistencia. Además, puede usar Inyección de dependencia para aislar la implementación, que se implementa en la capa de infraestructura/persistencia mediante repositorios.

Por ejemplo, el siguiente ejemplo con la interfaz `IOrderRepository` define qué operaciones necesitará implementar la clase `OrderRepository` en la capa de infraestructura. En la implementación actual de la aplicación, el código solo necesita agregar o actualizar pedidos a la base de datos, ya que las consultas se dividen siguiendo el enfoque CQRS simplificado.

```
// Definido en IOrderRepository.cs interfaz
pública IOrderRepository: IRepository<Order> {
```

```
Agregar orden (pedir orden);  
actualización nula (orden de pedido);  
Tarea<Pedido> GetAsync(int orderId);  
}  
  
// Definido en IRepository.cs (parte del dominio Seedwork) interfaz pública  
IRepository<T> donde T : IAggregateRoot {  
  
    IUnidadDeTrabajo UnidadDeTrabajo { get; }  
}
```

Recursos adicionales

- Martín Fowler. Interfaz separada. <https://www.martinfowler.com/eaaCatalog/separatedInterface.html>

Implementar objetos de valor

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 una identidad y un seguimiento de identidad, como los objetos de valor.

Un objeto de valor puede hacer referencia a otras entidades. Por ejemplo, en una aplicación que genera una ruta que describe cómo 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 entidades como Ciudad, Carretera, etc.

La Figura 7-13 muestra el objeto de valor Dirección dentro del agregado Orden.

Value Object within Aggregate

Order Aggregate (Multiple entities and Value-Object)

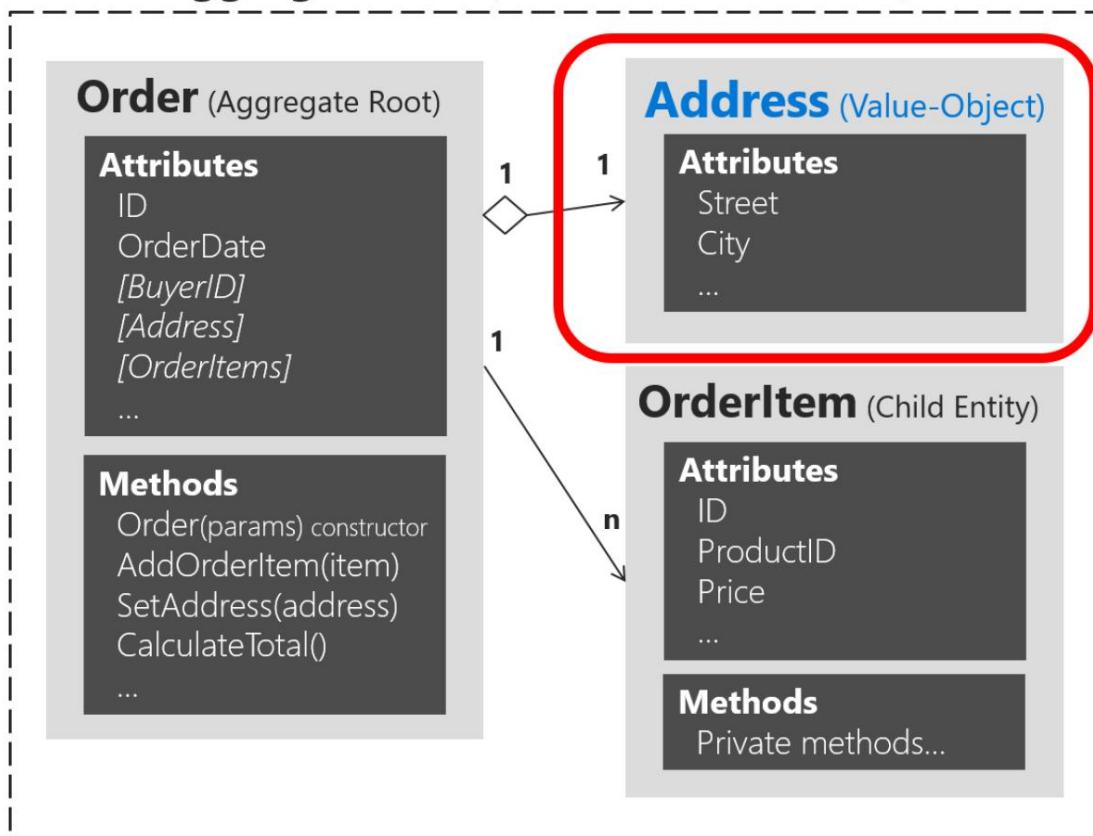


Figura 7-13. Objeto de valor de dirección dentro del agregado de pedido

Como se muestra en la Figura 7-13, una entidad generalmente se compone de múltiples atributos. Por ejemplo, la entidad Pedido se puede modelar como una entidad con una identidad y compuesta internamente por un conjunto de atributos como ID de pedido, Fecha de pedido, Artículos de pedido, etc. Pero la dirección, que es simplemente un valor complejo compuesto por país/región, calle, ciudad, etc., y no tiene identidad en este dominio, debe ser modelado y tratado como un objeto de valor.

Características importantes de los objetos de valor

Hay dos características principales para los objetos de valor:

- No tienen identidad.
- Son inmutables.

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 el objeto es

construido, debe proporcionar los valores requeridos, pero no debe permitir que cambien durante la vida útil del objeto.

Los objetos de valor le permiten realizar ciertos trucos para el rendimiento, gracias a su naturaleza inmutable. Esto es especialmente cierto en sistemas donde puede haber miles de instancias de objetos de valor, 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 marcar la 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 la implementación. contexto.

Implementación de objetos de valor en C#

En términos de implementación, puede tener una clase base de objeto de valor que tenga métodos de utilidad básicos como la igualdad 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.

```
clase abstracta pública ValueObject {
    bool estático protegido EqualOperator (ValueObject izquierda, ValueObject derecha) {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null)) {
            falso retorno;
        } devuelve ReferenceEquals (izquierda, nulo) || izquierda.Equals(derecha);
    }

    bool estático protegido NotEqualOperator (ValueObject izquierda, ValueObject derecha) {
        volver ! (EqualOperator (izquierda, derecha));
    }

    IEnumable<objeto> abstracto protegido GetEqualityComponents();
}

public override bool Equals(object obj) {
    if (obj == nulo || obj.GetType() != GetType()) {
        falso retorno;
    }

    var otro = (ValueObject)obj;
    devuelva this.GetEqualityComponents().SequenceEqual(other.GetEqualityComponents());
}

invalidación pública int GetHashCode() {
    devolver ObtenerComponentesEquidad()
        .Select(x => x != null ? x.GetHashCode() : 0)
        .Agregado((x, y) => x ^ y);
}
// Otros métodos de utilidad
}
```

ValueObject es un tipo de clase abstracto , pero en este ejemplo, no sobrecarga los operadores == y != . Puede optar por hacerlo, haciendo que las comparaciones se deleguen en la anulación de Igualdad. Por ejemplo, considere las siguientes sobrecargas de operadores en el tipo ValueObject :

```
operador booleano estático público ==( ValueObject uno, ValueObject dos) {
    devuelve EqualOperator(uno, dos);
}

operador booleano estático público !=( ValueObject uno, ValueObject dos) {
    devuelve NoEqualOperator(uno, dos);
}
```

Puede usar esta clase al implementar su objeto de valor real, como con el objeto de valor de dirección que se muestra en el siguiente ejemplo:

```
Dirección de clase pública : ValueObject {

    Public String Street { obtener; conjunto privado; } public
    String Ciudad { get; conjunto privado; } public String
    Estado { obtener; conjunto privado; } Cadena pública
    País { obtener; conjunto privado; } public String ZipCode
    { get; conjunto privado; }

    dirección pública () {}

    Dirección pública (cadena de calle, cadena de ciudad, cadena de estado, cadena de país, cadena de
    código postal) {

        calle = calle;
        Ciudad = ciudad;
        Estado = estado;
        País = país;
        Código Postal = código postal;
    }

    anulación protegida IEnumerable<objeto> GetEqualityComponents() {

        // Uso de una instrucción yield return para devolver cada elemento de uno en uno yield return
        Street; rendimiento retorno Ciudad; Estado de retorno de rendimiento ; retorno de rendimiento
        País; código postal de retorno de rendimiento ;

    }
}
```

Esta implementación de objeto de valor de Dirección no tiene identidad y, por lo tanto, no se define ningún campo de ID para ella, ni en la definición de clase de Dirección ni en la definición de clase de Objeto de valor .

No era posible no tener un campo de ID en una clase que usaría Entity Framework (EF) hasta EF Core 2.0, lo que ayuda enormemente a implementar objetos de mejor valor sin ID. Esa es precisamente la explicación de la siguiente sección.

Se podría argumentar que los objetos de valor, al ser inmutables, deberían ser de solo lectura (es decir, tener propiedades de solo obtener), y eso es cierto. Sin embargo, los objetos de valor generalmente se serializan y deserializan para pasar por las colas de mensajes, y ser de solo lectura evita que el deserializador asigne valores, por lo que simplemente los deja como un conjunto privado, que es lo suficientemente de solo lectura para ser práctico.

Semántica de comparación de objetos de valor

Se pueden comparar dos instancias del tipo Dirección utilizando todos los métodos siguientes:

```
var uno = nueva dirección ("1 Microsoft Way", "Redmond", "WA", "US", "98052"); var dos = nueva
dirección ("1 Microsoft Way", "Redmond", "WA", "US", "98052");

Console.WriteLine(EqualityComparer<Dirección>.Default.Equals(uno, dos)); // True
Console.WriteLine(objeto.Equals(uno, dos)); // True Console.WriteLine(one.Equals(two)); // True
Console.WriteLine(uno == dos); // Verdadero
```

Cuando todos los valores son iguales, las comparaciones se evalúan correctamente como verdaderas. Si no eligió sobrecargar los operadores == y !=, entonces la última comparación de uno == dos se evaluaría como falsa. Para obtener más información, consulte [Sobrecargar operadores de igualdad de ValueObject](#).

Cómo conservar objetos de valor en la base de datos con EF Core 2.0 y versiones posteriores

Acaba de ver cómo definir un objeto de valor en su modelo de dominio. Pero, ¿cómo puede persistirlo en la base de datos utilizando Entity Framework Core, ya que generalmente se dirige a entidades con identidad?

Antecedentes y enfoques anteriores con EF Core 1.1

Como antecedente, una limitación al usar EF Core 1.0 y 1.1 era que no podía usar tipos complejos como se define en EF 6.x en el .NET Framework tradicional. Por lo tanto, si usa EF Core 1.0 o 1.1, debe almacenar su objeto de valor como una entidad EF con un campo de ID. Luego, para que pareciera más un objeto de valor sin identidad, podría ocultar su ID para dejar en claro que la identidad de un objeto de valor no es importante en el modelo de dominio. Puede ocultar esa ID usando la ID como una [propiedad oculta](#).

Dado que esa configuración para ocultar la identificación en el modelo está configurada 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), el ID oculto que necesita la infraestructura de EF Core se implementó de la siguiente manera en el nivel de DbContext, utilizando Fluent API en el proyecto de infraestructura. Por lo tanto, la ID estaba oculta desde el punto de vista del modelo de dominio, pero aún estaba presente en la infraestructura.

```
// Enfoque antiguo con EF Core 1.1 // API
fluida dentro de OrderingContext:DbContext en el proyecto de infraestructura void
ConfigureAddress(EntityTypeBuilder<Address> addressConfiguration) {

    addressConfiguration.ToTable("dirección", DEFAULT_SCHEMA);

    addressConfiguration.Property<int>("Id") // Id es una propiedad oculta .IsRequired();

    direcciónConfiguración.HasKey("Id"); // Id es una propiedad oculta
}
```

Sin embargo, la persistencia de ese objeto de valor en la base de datos se realizó como una entidad regular en una tabla diferente.

Con EF Core 2.0 y versiones posteriores, hay formas nuevas y mejores de conservar los objetos de valor.

[Conservar objetos de valor como tipos de entidad de propiedad en EF Core 2.0 y versiones posteriores](#)

Incluso con algunas brechas entre el patrón de objeto de valor canónico en DDD y el tipo de entidad de propiedad en EF Core, actualmente es la mejor manera de conservar los objetos de valor con EF Core 2.0 y versiones posteriores. Puede ver las limitaciones al final de esta sección.

La función de tipo de entidad de propiedad se agregó a EF Core desde la versión 2.0.

Un tipo de entidad de propiedad le permite asignar tipos que no tienen su propia identidad definida explícitamente en el modelo de dominio y se utilizan como propiedades, como un objeto de valor, dentro de cualquiera de sus entidades. Un tipo de entidad de propiedad comparte el mismo tipo de CLR con otro tipo de entidad (es decir, es solo una clase normal).

La entidad que contiene la navegación definitoria es la entidad propietaria. Al consultar al propietario, los tipos de propiedad se incluyen de forma predeterminada.

Con solo mirar el modelo de dominio, parece que un tipo propio no tiene ninguna identidad. Sin embargo, bajo las cubiertas, los tipos de propiedad tienen la identidad, pero la propiedad de navegación del propietario es parte de esta identidad.

La identidad de instancias de tipos propios no es completamente propia. Consta de tres componentes:

- La identidad del propietario
- La propiedad de navegación que apunta a ellos
- En el caso de colecciones de tipos propios, un componente independiente (compatible con EF Core 2.2 y versiones posteriores).

Por ejemplo, en el modelo de dominio de Pedidos en eShopOnContainers, como parte de la entidad Pedido, el objeto de valor Dirección se implementa como un tipo de entidad de propiedad dentro de la entidad propietaria, que es la entidad Pedido. La dirección es un tipo sin propiedad de identidad definida en el modelo de dominio. Se utiliza como una propiedad del tipo Pedido para especificar la dirección de envío de un pedido en particular.

Por convención, se crea una clave principal oculta para el tipo propio y se asignará a la misma tabla que el propietario mediante la división de tablas. Esto permite usar tipos propios de manera similar a cómo se usan los tipos complejos en EF6 en el .NET Framework tradicional.

Es importante tener en cuenta que los tipos propios nunca se detectan por convención en EF Core, por lo que debe declararlos explícitamente.

En eShopOnContainers, en el archivo OrderingContext.cs, dentro del método `OnModelCreating()`, se aplican múltiples configuraciones de infraestructura. Uno de ellos está relacionado con la entidad Orden.

```
// Parte de la clase OrderingContext.cs en el proyecto Ordering.Infrastructure // invalidación protegida void
OnModelCreating(ModelBuilder modelBuilder) {
```

```
modelBuilder.ApplyConfiguration(nueva ClientRequestEntityTypeConfiguration());
modelBuilder.ApplyConfiguration(nuevo PaymentMethodEntityTypeConfiguration());
```

```

    modelBuilder.ApplyConfiguration(nueva OrderEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(nueva OrderItemEntityTypeConfiguration()); //...Configuraciones
    de tipos adicionales
}

```

En el siguiente código, la infraestructura de persistencia se define para la entidad Pedido:

```

// Parte de la clase OrderEntityTypeConfiguration.cs // public void
Configure(EntityTypeBuilder<Order> orderConfiguration) {

    orderConfiguration.ToTable("pedidos", OrderingContext.DEFAULT_SCHEMA);
    orderConfiguration.HasKey(o => o.Id); orderConfiguration.Ignore(b => b.DomainEvents);
    orderConfiguration.Property(o => o.Id)

        .ForSqlServerUseSequenceHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

    //El objeto de valor de dirección persiste como entidad de propiedad en EF Core 2.0
    orderConfiguration.OwnsOne(o => o.Address);

    orderConfiguration.Property<DateTime>("OrderDate").IsRequired();

    //...Validaciones adicionales, restricciones y código... //...

}

```

En el código anterior, el método `orderConfiguration.OwnsOne(o => o.Address)` especifica que la propiedad Address es una entidad de propiedad del tipo Order .

De forma predeterminada, las convenciones de EF Core nombran las columnas de la base de datos para las propiedades del tipo de entidad de propiedad como EntityProperty_OwnerEntityProperty. Por lo tanto, las propiedades internas de Dirección aparecerán en la tabla Pedidos con los nombres Dirección_Calle, Dirección_Ciudad (y así sucesivamente para Estado, País y Código Postal).

Puede agregar el método fluido `Property().HasColumnName()` para cambiar el nombre de esas columnas. En el caso de que Dirección sea una propiedad pública, las asignaciones serían como las siguientes:

```

orderConfiguration.OwnsOne(p => p.Dirección)
    .Property(p=>p.Street).HasColumnName("ShippingStreet");

orderConfiguration.OwnsOne(p => p.Dirección)
    .Property(p=>p.City).HasColumnName("ShippingCity");

```

Es posible encadenar el método `OwnsOne` en un mapeo fluido. En el siguiente ejemplo hipotético, OrderDetails posee BillingAddress y ShippingAddress, que son tipos de dirección . Entonces OrderDetails es propiedad del tipo de pedido .

```

orderConfiguration.OwnsOne(p => p.OrderDetails, cb =>
{
    cb.OwnsOne(c => c.Dirección de facturación);
    cb.OwnsOne(c => c.EnvíoDirección);

}); //... //...
orden de
clase pública {

    Id int público { obtener; colocar; }
    Detalles del pedido público Detalles del pedido { get; colocar; }
}

```

```

}

detalles de pedido de clase pública
{
    Dirección pública Dirección de facturación { get; colocar; }
    Dirección pública Dirección de envío { get; colocar; }
}

Dirección de clase pública {

    public string Calle { get; colocar; } public string
    Ciudad { get; colocar; }
}

```

Detalles adicionales sobre los tipos de entidades propias

- Los tipos propios se definen cuando configura una propiedad de navegación para un tipo particular usando la API fluida de OwnsOne.
- La definición de un tipo de propiedad en nuestro modelo de metadatos es una combinación de: el tipo de propietario, la propiedad de navegación y el tipo CLR del tipo de propiedad.
- La identidad (clave) de una instancia de tipo propio en nuestra pila es un compuesto de la identidad del tipo de propietario y la definición del tipo de propiedad.

Capacidades de entidades propias

- Los tipos propios pueden hacer referencia a otras entidades, ya sean propias (tipos propios anidados) o no propias (propiedades de navegación de referencia normal a otras entidades).
- Puede mapear el mismo tipo de CLR como diferentes tipos de propiedad en la misma entidad propietaria a través de propiedades de navegación separadas.
- La división de tablas se configura por convención, pero puede optar por no hacerlo asignando el tipo propio a una tabla diferente usando ToTable.
- La carga ansiosa se realiza automáticamente en los tipos propios, es decir, no es necesario llamar a .Include() en la consulta.
- Se puede configurar con el atributo [Owned], usando EF Core 2.1 y versiones posteriores.
- Puede manejar colecciones de tipos propios (usando la versión 2.2 y posteriores).

Limitaciones de entidades propias

- No puede crear un DbSet<T> de un tipo propio (por diseño).
- No puede llamar a ModelBuilder.Entity<T>() en tipos propios (actualmente por diseño).
- No hay soporte para tipos de propiedad opcionales (es decir, anulables) que se asignan con el propietario en la misma tabla (es decir, mediante la división de tablas). Esto se debe a que el mapeo se realiza para cada propiedad, no hay un centinela separado para el valor complejo nulo como un todo.

- No hay soporte de mapeo de herencia para tipos de propiedad, pero debería poder mapear dos tipos de hoja de las mismas jerarquías de herencia como diferentes tipos de propiedad. EF Core no razonará sobre el hecho de que forman parte de la misma jerarquía.

Principales diferencias con los tipos complejos de EF6

- La división de tablas es opcional, es decir, opcionalmente se pueden asignar a una tabla separada y seguir siendo tipos de propiedad.
- Pueden hacer referencia a otras entidades (es decir, pueden actuar como el lado dependiente en las relaciones con otros tipos sin propiedad).

Recursos adicionales

- Martín Fowler. Patrón ValueObject <https://martinfowler.com/bliki/ValueObject.html>
- Erick Evans. Diseño impulsado por dominio: abordar la complejidad en el corazón del software. (Libro; incluye una discusión sobre objetos de valor) [https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity Software/dp/0321125215/](https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity_Software/dp/0321125215/)

- Vaughn Vernon. Implementación del Diseño Dirigido por Dominio. (Libro; incluye una discusión sobre el valor objetos) [https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn Vernon/dp/0321834577/](https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn_Vernon/dp/0321834577/)
- Tipos de entidades propias <https://docs.microsoft.com/ef/core/modeling/owned-entities>
- Propiedades de sombra <https://docs.microsoft.com/ef/core/modeling/shadow-properties>
- Tipos complejos y/u objetos de valor. Discusión en el repositorio de EF Core GitHub (pestaña Problemas) <https://github.com/dotnet/efcore/issues/246>
- ValorObjeto.cs. Clase de objeto de valor base en eShopOnContainers. <https://github.com/dotnet/architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs>

- ValorObjeto.cs. Clase de objeto de valor base en CSharpFunctionalExtensions. <https://github.com/vkhorikov/CSharpFunctionalExtensions/blob/master/CSharpFunctionalExtensions/ValueObject/ValueObject.cs>

- Clase de dirección. Ejemplo de clase de objeto de valor en eShopOnContainers. <https://github.com/dotnet/architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/Aggregates Model/OrderAggregate/Address.cs>

Use clases de enumeración en lugar de tipos de enumeración

Las [enumeraciones](#) (o tipos de enumeración para abreviar) son un envoltorio de lenguaje ligero alrededor de un tipo integral.

Es posible que desee limitar su uso cuando almacene un valor de un conjunto cerrado de valores.

La clasificación basada en tamaños (pequeño, mediano, grande) es un buen ejemplo. El uso de enumeraciones para controlar el flujo o abstracciones más robustas puede ser un [olor a código](#). Este tipo de uso conduce a un código frágil con muchas declaraciones de flujo de control que verifican los valores de la enumeración.

En su lugar, puede crear clases de enumeración que habiliten todas las funciones enriquecidas de un lenguaje orientado a objetos.

Sin embargo, este no es un tema crítico y, en muchos casos, para simplificar, aún puede usar [tipos de enumeración regulares si esa](#) es su preferencia. El uso de clases de enumeración está más relacionado con conceptos relacionados con el negocio.

Implementar una clase base de enumeración

El microservicio de pedidos en eShopOnContainers proporciona una implementación de clase base de enumeración de muestra, como se muestra en el siguiente ejemplo:

```
Enumeración de clase abstracta pública : IComparable {
    cadena pública Nombre { obtener; conjunto privado; }

    Id int público { obtener; conjunto privado; }

    Enumeración protegida (int id, nombre de cadena ) => (Id, nombre) = (id, nombre);

    public override string ToString() => Nombre;

    public static IEnumerable<T> GetAll<T>() donde T : Enumeración =>
        typeof(T).GetFields(BindingFlags.Public |
            BindingFlags.Static |
            BindingFlags.DeclaredOnly)
        .Select(f => f.GetValue(null))
        .Cast<T>();

    public override bool Equals(object obj) {

        if (obj no es otro valor de enumeración) {

            falso retorno;
        }

        var typeMatches = GetType().Equals(obj.GetType()); var
        valueMatches = Id.Equals(otherValue.Id);

        devuelve las coincidencias de tipo && las coincidencias de valor;
    }

    public int CompareTo(otro) => Id.CompareTo(((Enumeración)otro).Id);

    // Otros métodos de utilidad...
}
```

Puede usar esta clase como un tipo en cualquier entidad u objeto de valor, como para el siguiente CardType :

Clase de enumeración :

```
tipo de tarjeta de clase pública
    : Enumeración
{
    public static CardType Amex = new(1, nameof(Amex)); Tipo de tarjeta
    Visa estática pública = nuevo (2, nombre de (Visa)); public static CardType
    MasterCard = new(3, nameof(MasterCard));

    public CardType(int id, cadena nombre) : base(id,
        nombre)

}
```

Recursos adicionales

- Jimmy Bogard. Clases de enumeración

<https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/>
- Steve Smith. Alternativas de enumeración en C#

<https://ardalis.com/enum-alternativas-en-c/>
- Enumeración.cs. Clase de enumeración base en eShopOnContainers

<https://github.com/dotnet/architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/E numeration.cs>

- CardType.cs. Ejemplo de clase de enumeración en eShopOnContainers.

<https://github.com/dotnet/architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/Aggregates Model/BuyerAggregate/CardType.cs>

- SmartEnum. Ardalís: clases para ayudar a producir enumeraciones más inteligentes fuertemente tipadas en .NET.

<https://www.nuget.org/packages/Ardalis.SmartEnum/>

Validaciones de diseño en la capa del modelo de dominio

En DDD, las reglas de validación pueden pensarse como invariantes. La principal responsabilidad de un agregado es aplicar invariantes a través de 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 número entero positivo, más un nombre de artículo y un precio. Por lo tanto, la aplicación de invariantes es responsabilidad de las entidades de dominio (especialmente de la raíz agregada) y un objeto de entidad no debería poder existir sin ser válido. Las reglas invariantes se expresan simplemente como contratos, y se generan excepciones o notificaciones cuando se infringen.

El razonamiento detrás de esto es que muchos errores ocurren porque los objetos están en un estado en el que nunca deberían haber estado.

Supongamos que ahora tenemos un `SendUserCreationEmailService` que toma un `UserProfile`... ¿cómo podemos racionalizar en ese servicio que `Name` no es nulo? ¿Lo comprobamos de nuevo? O más probablemente... simplemente no se molesta en verificar y "esperar lo mejor": espera que alguien se moleste en validar 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 empezamos a escribir este tipo de pruebas una y otra vez nos damos cuenta... "¿y si nunca permitiéramos que el nombre se volviera nulo? ¡No tendríamos todas estas pruebas!".

Implementar 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. Hay varias formas de implementar validaciones, como verificar datos y generar excepciones si falla la validación. También hay patrones más avanzados, como el uso del patrón de especificación para validaciones y el patrón de notificación para devolver una colección de errores en lugar de devolver una excepción para cada validación a medida que ocurre.

Validar condiciones y lanzar excepciones

El siguiente ejemplo de código muestra el enfoque más simple para la validación en una entidad de dominio generando 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(Dirección dirección) {
    _shippingAddress = dirección ?? lanzar una nueva ArgumentNullException(nombrededirección));
}
```

Un mejor ejemplo demostraría la necesidad de garantizar que el estado interno no cambie o que ocurran todas las mutaciones de un método. Por ejemplo, la siguiente implementación dejaría el objeto en un estado no válido:

```
public void SetAddress(string line1, string line2, string city, string
state, int zip)
{
    _shippingAddress.line1 = linea1 ?? lanzar nuevo ...
    _shippingAddress.line2 = line2; _shippingAddress.city =
ciudad ?? lanzar nuevo ... _shippingAddress.state =
(isValid(estado) ? estado: lanzar nuevo ...);
}
```

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 no sea válida.

Se puede usar un enfoque similar en el constructor de la entidad, generando una excepción para asegurarse de que la entidad sea válida una vez creada.

Utilice atributos de validación en el modelo en función de las anotaciones de datos

Las anotaciones de datos, como los atributos `Obligatorio` o `MaxLength`, se pueden usar para configurar las propiedades de campo de la base de datos de EF Core, como se explica en detalle en la sección [Asignación de tablas](#), pero ya no funcionan .

para la validación de entidades en EF Core (tampoco lo hace el método `IValidableObject.Validate`), como lo han hecho desde EF 4.x en .NET Framework.

Las anotaciones de datos y la interfaz `IValidableObject` todavía se pueden usar para la validación del modelo durante el enlace del modelo, antes de la invocación de acciones del controlador como de costumbre, pero ese modelo está destinado a ser un ViewModel o DTO y eso es una preocupación de MVC o API, no una preocupación de modelo de dominio.

Habiendo aclarado la diferencia conceptual, aún puede usar anotaciones de datos y `IValidableObject` en la clase de entidad para la validación, si sus acciones reciben un parámetro de objeto de clase de entidad, lo cual no se recomienda. En ese caso, la validación ocurrirá en el enlace del modelo, justo antes de invocar la acción y puede verificar la propiedad `ModelState.IsValid` del controlador para verificar el resultado, pero nuevamente, sucede en el controlador, no antes de persistir el objeto de entidad en el `DbContext`, como lo había hecho desde EF 4.x.

Todavía puede implementar la validación personalizada en la clase de entidad mediante anotaciones de datos y el método `IValidableObject.Validate`, anulando el método `SaveChanges` de `DbContext`.

Puede ver una implementación de muestra para validar entidades `IValidableObject` en [este comentario en GitHub](#). Esa muestra no realiza validaciones basadas en atributos, pero deberían ser fáciles de implementar mediante la reflexión en la misma anulación.

Sin embargo, desde el punto de vista de DDD, el modelo de dominio se mantiene simplificado con el uso de excepciones en los métodos de comportamiento de su entidad o mediante la implementación de patrones de especificación y notificación para hacer cumplir las reglas de validación.

Puede tener sentido usar anotaciones de datos en la capa de la aplicación en las clases de ViewModel (en lugar de las entidades de dominio) que aceptarán la entrada, para permitir la validación del modelo dentro de la capa de la interfaz de usuario. Sin embargo, esto no debe hacerse excluyendo la validación dentro del modelo de dominio.

Valide entidades implementando el patrón de especificación y el patrón de 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 solo uno de esos patrones, por ejemplo, validar manualmente con declaraciones de control, pero usar el patrón de notificación para apilar y devolver una lista de errores de validación.

Usar validación diferida en el dominio

Existen varios enfoques para tratar las validaciones diferidas en el dominio. En su libro [Implementing Domain-Driven Design](#), Vaughn Vernon los analiza en la sección sobre validación.

Validación en dos pasos

Considere también la validación en dos pasos. Utilice la validación a nivel de campo en sus objetos de transferencia de datos (DTO) de comando y la validación a nivel de dominio dentro de sus entidades. Puede hacer esto devolviendo un objeto de resultado en lugar de excepciones para facilitar el manejo de los errores de validación.

Al utilizar la validación de campos con anotaciones de datos, por ejemplo, no duplica la definición de validación. Sin embargo, la ejecución puede ser tanto del lado del servidor como del lado del cliente en el caso de los DTO (comandos y ViewModels, por ejemplo).

Recursos adicionales

- Raquel Appel. Introducción a la validación de modelos en ASP.NET Core MVC <https://docs.microsoft.com/aspnet/core/mvc/models/validation>
- Rick Anderson. Agregar validación <https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation>
- Martín Fowler. Reemplazo de excepciones de lanzamiento con notificación en validaciones <https://martinfowler.com/articles/replaceThrowWithNotification.html>
- Patrones de especificación y notificación <https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns>
- Lev Gorodinski. Validación en diseño controlado por dominio (DDD) <http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/>
- Colín Jack. Validación del modelo de dominio <https://colinjack.blogspot.com/2008/03/domain-model-validation.html>
- Jimmy Bogard. Validación en un mundo DDD <https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/>

Validación del lado del cliente (validación en las capas de presentación)

Incluso cuando la fuente de la verdad es el modelo de dominio y, en última instancia, debe tener una validación en el nivel del modelo de dominio, la validación aún se puede manejar tanto en el nivel del modelo de dominio (lado del servidor) como en la interfaz de usuario (lado del cliente).

La validación del lado del cliente es una gran comodidad para los usuarios. Ahorra tiempo que, de otro modo, pasarían esperando un viaje de ida y vuelta al servidor que podría devolver errores de validación. En términos comerciales, incluso unas pocas fracciones de segundo multiplicadas cientos de veces cada día suman mucho tiempo, gastos y frustración. La validación sencilla e inmediata permite a los usuarios trabajar de manera más eficiente y producir entradas y salidas de mejor calidad.

Así como el modelo de vista y el modelo de dominio son diferentes, la validación del modelo de vista y la validación del modelo de dominio pueden ser similares pero tienen un propósito diferente. Si le preocupa DRY (el principio de No repetirse), considere que en este caso la reutilización de código también podría significar acoplamiento, y en las aplicaciones empresariales es más importante no acoplar el lado del servidor al lado del cliente que seguir el Principio SECO.

Incluso cuando utilice la validación del lado del cliente, siempre debe validar sus comandos o ingresar DTO en el código del servidor, ya que las API del servidor son un posible vector de ataque. Por lo general, hacer ambas cosas es tu mejor apuesta.

porque si tiene una aplicación cliente, desde una perspectiva de UX, es mejor ser proactivo y no permitir que el usuario ingrese información no válida.

Por lo tanto, en el código del lado del cliente, normalmente valida los ViewModels. También puede validar los DTO o los comandos de salida del cliente antes de enviarlos a los servicios.

La implementación de la validación del lado del cliente depende del tipo de aplicación cliente que esté creando.

Será 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 codificada con Xamarin y C#.

Recursos adicionales

Validación en aplicaciones móviles de Xamarin

- Valide la entrada de texto y muestre los errores https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/
- Devolución de llamada de validación <https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/>

Validación en aplicaciones ASP.NET Core

- Rick Anderson. Agregar validación <https://docs.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation>

Validación en aplicaciones web SPA (Angular 2, TypeScript, JavaScript)

- Validación de formularios <https://angular.io/guide/form-validation>
- Validación. Documentación de la brisa. <https://breeze.github.io/doc-js/validation.html>

En resumen, estos son los conceptos más importantes en lo que respecta a la validación:

- Las entidades y los agregados deben imponer su propia coherencia y ser "siempre válidos". Las raíces agregadas son responsables de la consistencia de múltiples entidades dentro del mismo agregado.
- Si cree que una entidad necesita ingresar a un estado no válido, considere usar un modelo de objeto diferente, por ejemplo, usar un DTO temporal hasta que cree la entidad de dominio final.
- Si necesita crear varios objetos relacionados, como un agregado, y solo son válidos una vez que se han creado todos, considere usar el patrón Factory.
- En la mayoría de los casos, es bueno tener una validación redundante en el lado del cliente, porque la aplicación puede ser proactiva.

Eventos de 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 utilizando la terminología de DDD, use eventos de dominio para implementar explícitamente efectos secundarios en múltiples agregados.

Opcionalmente, para una mejor escalabilidad y un menor impacto en los bloqueos de bases de datos, utilice la coherencia final entre agregados dentro del mismo dominio.

¿Qué es un evento de dominio?

Un evento es algo que ha sucedido en el pasado. Un evento de dominio es algo que sucedió en el dominio y que desea que otras partes del mismo dominio (en proceso) conozcan. Las partes notificadas suelen reaccionar de alguna manera a los eventos.

Un beneficio importante de los eventos de dominio es que los efectos secundarios se pueden expresar explícitamente.

Por ejemplo, si solo usa Entity Framework y tiene que haber una reacción a algún evento, probablemente codificará lo que necesita cerca de lo que desencadena el evento. Entonces, la regla se acopla, implícitamente, al código, y usted tiene que examinar el código para, con suerte, darse cuenta de que la regla está implementada allí.

Por otro lado, el uso de eventos de dominio hace que el concepto sea explícito, porque hay un DomainEvent y al menos un DomainEventHandler involucrados.

Por ejemplo, en la aplicación eShopOnContainers, cuando se crea un pedido, el usuario se convierte en comprador, por lo que se genera un OrderStartedDomainEvent y se maneja en ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler, por lo que el concepto subyacente es evidente.

En resumen, los eventos de dominio lo ayudan a expresar, explícitamente, las reglas del dominio, basadas en el lenguaje ubicuo proporcionado por los expertos del dominio. Los eventos de dominio también permiten una mejor separación de preocupaciones entre clases dentro del mismo dominio.

Es importante asegurarse de que, al igual que una transacción de base de datos, todas las operaciones relacionadas con un evento de dominio finalicen correctamente o ninguna de ellas finalice.

Los eventos de dominio son similares a los eventos de estilo de mensajería, con una diferencia importante. Con mensajería real, colas de mensajes, intermediarios de mensajes o un bus de servicio que usa AMQP, un mensaje siempre se envía de forma asíncrona y se comunica a través de procesos y máquinas. Esto es útil para integrar múltiples Bounded Contexts, microservicios o incluso diferentes aplicaciones. Sin embargo, con los eventos de dominio, desea generar un evento desde la operación de dominio que está ejecutando actualmente, pero desea que ocurran efectos secundarios dentro del mismo dominio.

Los eventos del dominio y sus efectos secundarios (las acciones desencadenadas posteriormente que son administradas por los controladores de eventos) deben ocurrir casi de inmediato, generalmente durante el proceso y dentro del mismo dominio. Por lo tanto, los eventos de dominio pueden ser síncronos o asíncronos. Sin embargo, los eventos de integración siempre deben ser asíncronos.

Eventos de dominio frente a eventos de integración

Semánticamente, los eventos de dominio e integración son lo mismo: notificaciones sobre algo que acaba de suceder. Sin embargo, su implementación debe ser diferente. Los eventos de dominio son solo 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 comprometidas y actualizaciones a subsistemas adicionales, ya sean otros microservicios, Bounded Contexts o incluso aplicaciones externas. Por lo tanto, deberían ocurrir solo si la entidad persiste con éxito; de lo contrario, es como si la operación completa nunca hubiera ocurrido.

Como se mencionó anteriormente, los eventos de integración deben basarse en la comunicación asíncrona entre múltiples microservicios (otros Bounded Contexts) o incluso sistemas/aplicaciones externos.

Por lo tanto, la interfaz del bus de eventos necesita alguna infraestructura que permita la comunicación entre procesos y distribuida entre 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 basado en push.

Eventos de dominio como una forma preferida de desencadenar efectos secundarios en múltiples agregados dentro del mismo dominio

Si ejecutar un comando relacionado con una instancia agregada requiere que se ejecuten reglas de dominio adicionales en uno o más agregados adicionales, debe diseñar e implementar esos efectos secundarios para que los activen los eventos del 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.

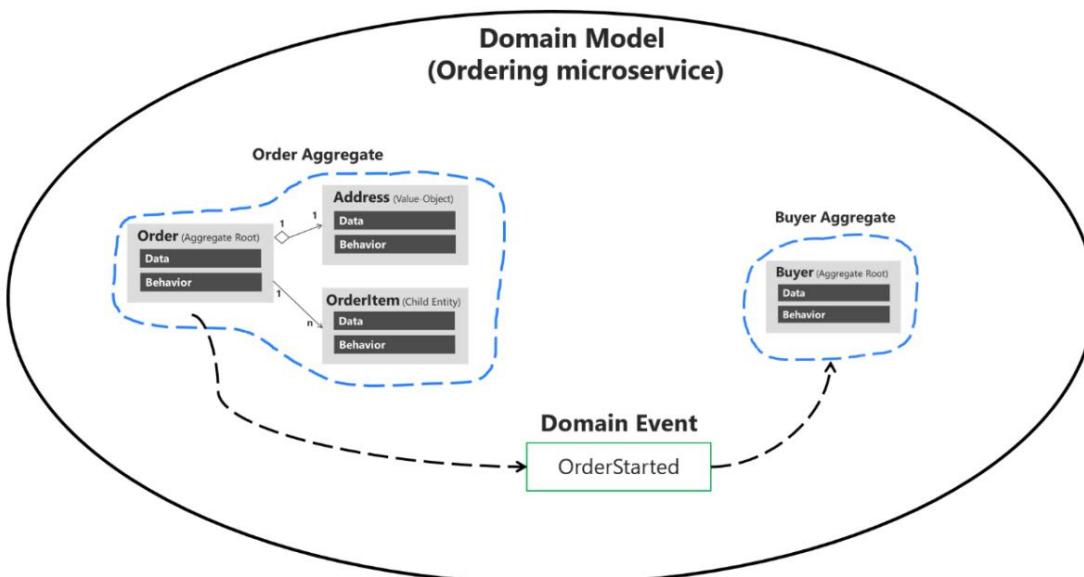


Figura 7-14. Eventos de dominio para hacer cumplir la coherencia entre múltiples agregados dentro del mismo dominio

La figura 7-14 muestra cómo los eventos de dominio logran la consistencia entre los agregados. Cuando el usuario inicia un pedido, Order Aggregate envía un **evento de dominio OrderStarted**. Buyer Aggregate gestiona el evento de dominio OrderStarted para crear un objeto Buyer en el microservicio de pedidos, en función de la información de usuario original del microservicio de identidad (con información proporcionada en el comando CreateOrder).

Como alternativa, puede suscribir la raíz agregada para eventos generados por miembros de sus agregados (entidades secundarias). Por ejemplo, cada entidad secundaria OrderItem puede generar un evento cuando el precio del artículo es mayor que una cantidad específica, o cuando la cantidad del artículo del producto es demasiado alta. La raíz agregada puede recibir esos eventos y realizar un cálculo o agregación global.

Es importante comprender que esta comunicación basada en eventos no se implementa directamente dentro de los agregados; necesita implementar controladores de eventos de dominio.

El manejo de los eventos del dominio es una preocupación de la aplicación. La capa del modelo de dominio solo debe centrarse en la lógica del dominio, cosas que un experto en el dominio entendería, no en la infraestructura de la aplicación como los controladores y las acciones de persistencia de efectos secundarios que usan repositorios. Por lo tanto, el nivel de la capa de aplicación es donde debe tener controladores de eventos de dominio que activen acciones cuando se genera un evento de dominio.

Los eventos de dominio también se pueden usar para desencadenar cualquier número de acciones de la aplicación y, lo que es más importante, deben estar abiertos para aumentar ese número en el futuro de forma 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 generar acciones de aplicación como notificaciones.

El punto clave es el número abierto de acciones que se ejecutarán cuando ocurra un evento de dominio. Eventualmente, las acciones y reglas en el dominio y la aplicación crecerán. La complejidad o la cantidad de acciones de efectos secundarios cuando algo suceda aumentará, pero si su código estuviera acoplado con "pegamiento" (es decir, creando objetos específicos con nuevos), entonces cada vez que necesitará agregar una nueva acción, también tendrá que cambiar el código de trabajo y probado.

Este cambio podría generar nuevos errores y este enfoque también va en contra del [principio Abierto/Cerrado de SOLID](#). No solo eso, la clase [original que](#) orquestaba las operaciones 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 detallada y desacoplada segregando responsabilidades usando este enfoque:

1. Envíe un comando (por ejemplo, CreateOrder).
2. Recibir el comando en un controlador de comandos.
 - Ejecutar una sola transacción de agregado.
 - (Opcional) Genere eventos de dominio para efectos secundarios (por ejemplo, OrderStartedDomainEvent).
3. Manejar eventos de dominio (dentro del proceso actual) que ejecutarán un número abierto de eventos secundarios. efectos en múltiples agregados o acciones de aplicación. Por ejemplo:
 - Verificar o crear comprador y forma de pago.
 - Cree y envíe 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 aplicaciones adicionales que necesita realizar en microservicios que se conectan con eventos de integración y el bus de eventos.

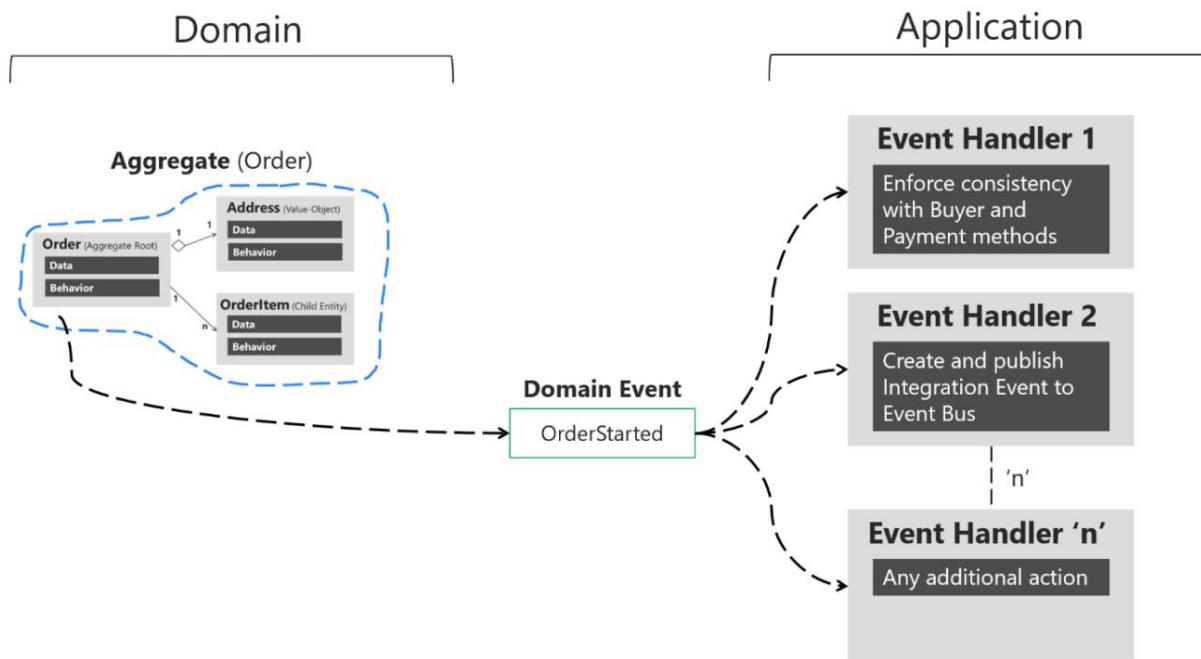


Figura 7-15. Manejo de múltiples acciones por dominio

Puede haber varios controladores para el mismo evento de dominio en la capa de aplicación, un controlador puede resolver la coherencia entre los agregados y otro controlador puede publicar un evento de integración, por lo que otros microservicios pueden hacer algo con él. Los controladores de eventos suelen estar en la capa de la aplicación, porque usará objetos de infraestructura como repositorios o una API de aplicación para el comportamiento del microservicio. En ese sentido, los controladores de eventos son similares a los controladores de comandos, por lo que ambos forman parte de la capa de aplicación. La diferencia importante es que un comando debe procesarse solo una vez. Un evento de dominio podría procesarse cero o n veces, porque varios receptores o controladores de eventos pueden recibirla con un propósito diferente para cada controlador.

Tener un número abierto de controladores por evento de dominio le permite agregar tantas reglas de dominio como sea necesario, sin afectar el código actual. Por ejemplo, implementar la siguiente regla comercial podría ser tan fácil como agregar algunos controladores de eventos (o incluso solo uno):

Cuando el monto total comprado por un cliente en la tienda, en cualquier número de pedidos, supere los \$6,000, aplique un descuento del 10 % a cada pedido nuevo y notifique al cliente con un correo electrónico sobre ese descuento para pedidos futuros.

Implementar 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 suceder en el dominio, como se muestra en el siguiente ejemplo:

```
Clase pública OrderStartedDomainEvent: INotification {
```

```

cadena pública ID de usuario { obtener; }
public string nombre de usuario { get; }
public int CardTypeid { obtener; } public
string Número de tarjeta { get; } public string
CardSecurityNumber { get; } public string
NombreTitularDeLaTarjeta { get; } public DateTime
CardExpiration { get; } Orden pública Orden { get; }

public OrderStartedDomainEvent(orden de pedido, string ID de usuario, string nombre de usuario , int
cardTypeid, string cardNumber, string cardSecurityNumber,
string cardHolderName,
Expiración de la tarjeta de fecha y hora)
{
    orden = orden;
    ID de usuario = ID de usuario;
    nombre de usuario = nombre de usuario;
    IdTipoTarjeta = IdTipoTarjeta;
    NúmeroTarjeta = NúmeroTarjeta;
    NúmeroSeguridadTarjeta = NúmeroSeguridadTarjeta;
    Nombre del Titular de la Tarjeta = Nombre del Titular de la Tarjeta;
    ExpiraciónTarjeta = ExpiraciónTarjeta;
}
}

```

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 la clase del evento debe representarse como un verbo en tiempo pasado, como OrderStartedDomainEvent o OrderShippedDomainEvent. Así es como se implementa el evento de dominio en el microservicio de pedidos en eShopOnContainers.

Como se señaló anteriormente, una característica importante de los eventos es que, dado que un evento es algo que sucedió en el pasado, no debe cambiar. Por lo tanto, debe ser una clase inmutable. Puede ver en el código anterior que las propiedades son de solo lectura. No hay forma de actualizar el objeto, solo puede establecer valores cuando lo crea.

Es importante resaltar aquí que si los eventos del dominio fueran a manejarse de forma asíncrona, usando una cola que requería serializar y deserializar los objetos del evento, las propiedades tendrían que ser "conjunto privado" en lugar de solo lectura, por lo que el deserializador podría asignar los valores al quitar la cola. Esto no es un problema en el microservicio de pedidos, ya que la publicación/suscripción del evento de dominio se implementa sincrónicamente mediante MediatR.

Levantar eventos de dominio

La siguiente pregunta es cómo generar un evento de dominio para que llegue a sus controladores de eventos relacionados. Puede utilizar múltiples enfoques.

Udi Dahan propuso originalmente (por ejemplo, en varias publicaciones relacionadas, como [Domain Events - Take 2](#)) usar una clase estática para administrar y generar eventos. Esto podría incluir una clase estática llamada DomainEvents que generaría eventos de dominio inmediatamente cuando se llama, usando una sintaxis como DomainEvents.Raise(Event myEvent). Jimmy Bogard escribió una publicación de blog ([Fortalecimiento de su dominio: Eventos de dominio](#)) que recomienda un enfoque similar.

Sin embargo, cuando la clase de eventos de dominio es estática, también se envía a los controladores de inmediato. Esto hace que las pruebas y la depuración sean más difíciles, porque los controladores de eventos con lógica de efectos secundarios se ejecutan inmediatamente después de que se genera el evento. Cuando está probando y depurando, solo desea concentrarse en lo que sucede en las clases agregadas actuales; no desea ser redirigido repentinamente a otros controladores de eventos por efectos secundarios relacionados con otros agregados o la lógica de la aplicación.

Esta es la razón por la que han evolucionado otros enfoques, como se explica en la siguiente sección.

El enfoque diferido para generar y despachar eventos

En lugar de enviar a un controlador de eventos de dominio inmediatamente, un mejor enfoque es agregar los eventos de dominio a una colección y luego enviar esos eventos de dominio justo antes o justo después de confirmar la transacción (como con `SaveChanges` en EF). (Este enfoque fue descrito por Jimmy Bogard en esta publicación [Un mejor patrón de eventos de dominio](#)).

Es importante decidir si envía los eventos de dominio justo antes o justo después de confirmar 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 lidiar con la eventual coherencia entre múltiples agregados. Este tema se trata en la siguiente sección.

El enfoque diferido es el que usa `eShopOnContainers`. Primero, agrega los eventos que suceden en sus entidades a una colección o lista de eventos por entidad. Esa lista debe ser parte del objeto de entidad, o incluso mejor, parte de su clase de entidad base, como se muestra en el siguiente ejemplo de la clase base `Entity`:

```
Entidad de clase abstracta pública {

    //...
    lista privada <INotificación> _domainEvents; public
    List<INotification> DomainEvents => _domainEvents;

    public void AddDomainEvent(INotification eventItem) {

        _dominioEventos = _dominioEventos ?? nueva Lista<INotificación>();
        _domainEvents.Add(elemento de evento);
    }

    public void RemoveDomainEvent(INotification eventItem) {

        _domainEvents?.Remove(eventItem);
    }
    //... Código adicional
}
```

Cuando desee generar un evento, simplemente agréguelo a la colección de eventos desde el código en cualquier método de la entidad raíz agregada.

El siguiente código, parte de la [raíz agregada del pedido en eShopOnContainers](#), muestra un ejemplo:

```
var orderStartedDomainEvent = new OrderStartedDomainEvent(this, //Order object cardTypeId, cardNumber,
                                                       cardSecurityNumber,
                                                       cardHolderName, cardExpiration);

this.AddDomainEvent(orderStartedDomainEvent);
```

Tenga en cuenta que lo único que hace el método AddDomainEvent es agregar un evento a la lista.

Aún no se ha enviado ningún evento y todavía no se ha invocado ningún controlador de eventos.

En realidad, desea enviar los eventos más adelante, cuando confirme la transacción en la base de datos. Si está utilizando Entity Framework Core, eso significa en el método SaveChanges de su EF DbContext, como en el siguiente código:

```
// EF Core DbContext clase
pública OrderingContext: DbContext, IUnitOfWork {

    // ...
    public async Task<bool> SaveEntitiesAsync(CancellationToken cancelacionToken =
predeterminado (token de cancelación))
    {
        // Envío de la colección de eventos de dominio.
        // Opciones: //
        A) Justo ANTES de confirmar datos (EF SaveChanges) en la base de datos. Esto hace // una sola transacción
que incluye los efectos secundarios del evento de dominio // controladores que usan el mismo DbContext con
la duración del alcance // B) Justo DESPUÉS de enviar datos (EF SaveChanges) a la base de datos. Esto hace
que // múltiples transacciones. Necesitará manejar la consistencia eventual y // las acciones compensatorias en
caso de fallas. esperar _mediator.DispatchDomainEventsAsync(estos);

        // Despues de que se ejecute esta linea, todos los cambios (desde el controlador de comandos y los controladores
de // eventos de dominio) realizados a través de DbContext se confirmarán var result = await
base.SaveChangesAsync();
    }
}
```

Con este código, envía los eventos de la entidad a sus respectivos controladores de eventos.

El resultado general es que ha desacoplado la generación de un evento de dominio (un simple agregado a una lista en la memoria) del envío a un controlador de eventos. Además, según el tipo de despachador que utilice, puede despachar los eventos de forma síncrona o asíncrona.

Tenga en cuenta que los límites transaccionales entran en juego aquí. Si su unidad de trabajo y transacción puede abarcar más de un agregado (como cuando usa EF Core y una base de datos relacional), esto puede funcionar bien. Pero si la transacción no puede abarcar agregados, debe implementar pasos adicionales para lograr la coherencia. Esta es otra razón por la que la ignorancia persistente no es universal; Depende del sistema de almacenamiento que utilice.

Transacción única entre agregados versus consistencia final entre agregados

La cuestión de si realizar una sola transacción entre agregados o confiar en la consistencia final entre esos 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, abogan por la eventual consistencia entre los agregados. Por ejemplo, en su libro Domain-Driven Design, Eric Evans dice esto:

No se espera que cualquier regla que abarque Agregados esté actualizada en todo momento. Mediante el procesamiento de eventos, el procesamiento por lotes u otros mecanismos de actualización, se pueden resolver otras dependencias dentro de un tiempo específico. (página 128)

Vaughn Vernon dice lo siguiente en [Diseño agregado eficaz. Parte II: Hacer que los agregados trabajen juntos](#):

Por lo tanto, si la ejecución de un comando en una instancia agregada requiere que se ejecuten reglas comerciales adicionales en uno o más agregados, use la consistencia eventual [...] Existe una forma práctica de respaldar la consistencia eventual en un modelo DDD. Un método agregado publica un evento de dominio que se entrega a tiempo a uno o más suscriptores asíncronos.

Este fundamento se basa en aceptar transacciones detalladas en lugar de transacciones que abarquen muchos agregados o entidades. La idea es que, en el segundo caso, la cantidad de bloqueos de la base de datos sea sustancial en aplicaciones a gran escala con necesidades de alta escalabilidad. Aceptar el hecho de que las aplicaciones altamente escalables 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 el negocio y, en cualquier caso, es responsabilidad de los expertos del dominio decir si las operaciones particulares necesitan transacciones atómicas o no. Si una operación siempre necesita una transacción atómica entre múltiples agregados, puede preguntar si su agregado debe ser más grande o si no se diseñó correctamente.

Sin embargo, otros desarrolladores y arquitectos como Jimmy Bogard están de acuerdo con abarcar una sola transacción en varios agregados, pero solo 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:

Por lo general, quiero que los efectos secundarios de un evento de dominio ocurran dentro de la misma transacción lógica, pero no necesariamente en el mismo alcance de generar el evento de dominio [...] Justo antes de confirmar nuestra transacción, enviamos nuestros eventos a sus respectivos controladores.

Si envía los eventos del dominio justo antes de confirmar la transacción original, es porque desea que los efectos secundarios de esos eventos se incluyan en la misma transacción. Por ejemplo, si falla el método EF DbContext SaveChanges, la transacción revertirá todos los cambios, incluido el resultado de las operaciones de efectos secundarios implementadas por los controladores de eventos de dominio relacionados. Esto se debe a que el ámbito de vida de DbContext se define de forma predeterminada como "ámbito". Por lo tanto, el objeto DbContext se comparte entre varios objetos de repositorio que se instancian dentro del mismo ámbito o gráfico de objetos. Esto coincide con el alcance de HttpRequest al desarrollar aplicaciones Web API 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 le digan los expertos en dominios. También depende de cuán escalable necesite que sea el servicio (las transacciones más granulares tienen menos impacto con respecto a los bloqueos de la base de datos). Y depende de cuánta inversión esté dispuesto a hacer en su código, ya que la eventual consistencia requiere un código más complejo 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, si hay un problema y los controladores de eventos no pueden confirmar sus efectos secundarios, tendrá incoherencias entre los agregados.

Una forma de permitir acciones compensatorias sería almacenar los eventos del dominio en tablas de bases de datos adicionales para que puedan ser parte de la transacción original. Posteriormente, podría tener un proceso por lotes que detecte inconsistencias y ejecute acciones compensatorias comparando la lista de eventos con la

estado actual de los agregados. Las acciones compensatorias son parte de un tema complejo que requerirá un análisis profundo de su parte, que incluye discutirlo con el usuario comercial y los expertos del dominio.

En cualquier caso, puedes elegir el enfoque que necesites. Pero el enfoque diferido inicial (generar los eventos antes de la confirmación, por lo que usa 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 de negocios. También es el enfoque utilizado en el microservicio de pedidos en eShopOnContainers.

Pero, ¿cómo envía realmente esos eventos a sus respectivos controladores de eventos? ¿Cuál es el objeto `_mediator` que ves en el ejemplo anterior? Tiene que ver con las técnicas y los artefactos que usa para mapear entre eventos y sus controladores de eventos.

El despachador de eventos de dominio: asignación de eventos a controladores de eventos

Una vez que pueda enviar o publicar los eventos, necesita algún tipo de artefacto que publique el evento, de modo que cada controlador 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 solo necesita procesar esos eventos dentro del mismo proceso (es decir, dentro del mismo dominio y capa de aplicación).

Otra forma de asignar eventos a varios controladores de eventos es mediante el registro de tipos en un contenedor de IoC para que pueda deducir dinámicamente dónde enviar los eventos. En otras palabras, necesita saber qué controladores de eventos necesitan para obtener un evento específico. La Figura 7-16 muestra un enfoque simplificado para este enfoque.

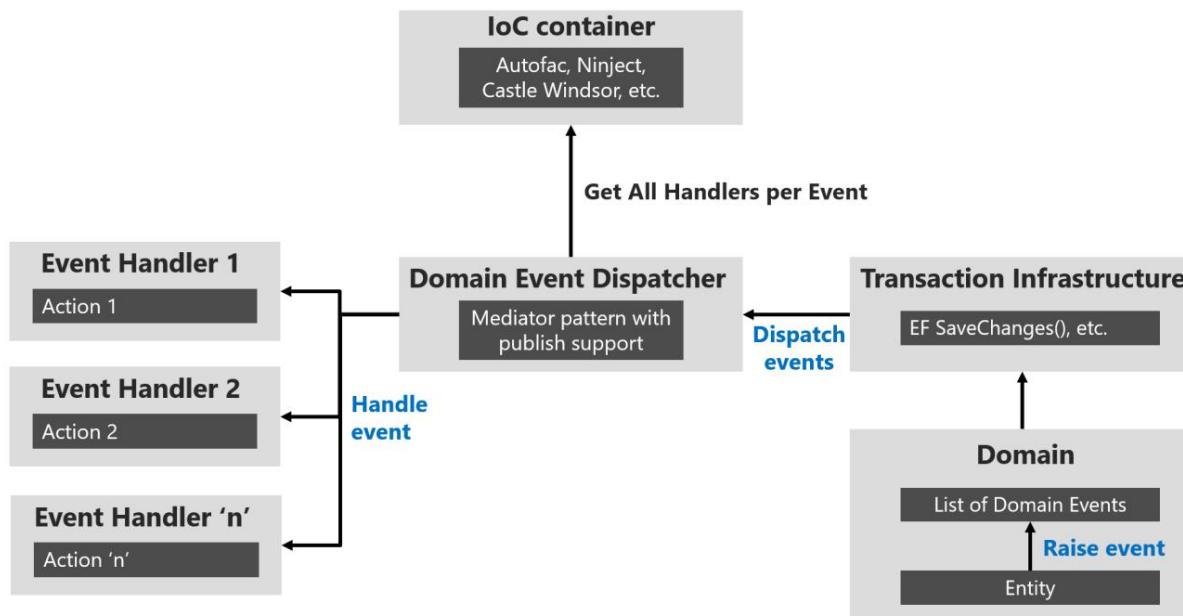


Figura 7-16. Despachador de eventos de dominio usando IoC

Puede construir todas las tuberías y artefactos para implementar ese enfoque usted mismo. Sin embargo, también puede usar bibliotecas disponibles como [MediatR](#) que usa su contenedor IoC debajo de las cubiertas. Por lo tanto, puede usar directamente las interfaces predefinidas y los métodos de publicación/despacho del objeto mediador.

En el código, primero debe registrar los tipos de controladores de eventos en su contenedor IoC, como se muestra en el siguiente ejemplo en [el microservicio eShopOnContainers Ordering](#):

```
Clase pública MediatorModule: Autofac.Module {
    anulación protegida carga vacía (ContainerBuilder builder) {
        // Otros registros...
        // Registrar las clases DomainEventHandler (implementan
IAsyncNotificationHandler<>) // en el ensamblado que contiene los eventos del dominio

        builder.RegisterAssemblyTypes(typeof(ValidateOrAddBuyerAggregateWhenOrderStartedDomainEvent Handler)
            .GetTypeInfo().Asamblea)

        .AsClosedTypesOf(typeof(IAsyncNotificationHandler<>)); // Otros
        registros...
    }
}
```

El código primero identifica el ensamblado que contiene los controladores de eventos del dominio al ubicar el ensamblado que contiene cualquiera de los controladores (usando typeof(ValidateOrAddBuyerAggregateWhenXxxx), pero podría haber elegido cualquier otro controlador de eventos para ubicar el ensamblado). Dado que todos los controladores de eventos implementan la interfaz IAsyncNotificationHandler, el código solo busca esos tipos y registra todos los controladores de eventos.

Cómo suscribirse a eventos de dominio

Cuando usa MediatR, cada controlador de eventos debe usar un tipo de evento que se proporciona en el parámetro genérico de la interfaz INotificationHandler, como puede ver en el siguiente código:

```
clase pública ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : IAsyncNotificationHandler<OrderStartedDomainEvent>
```

En función de la relación entre el evento y el controlador de eventos, que puede considerarse la suscripción, el artefacto MediatR puede descubrir todos los controladores de eventos para cada evento y activar cada uno de esos controladores de eventos.

Cómo manejar eventos de dominio

Finalmente, el controlador de eventos generalmente implementa código de capa de aplicación que usa repositorios de infraestructura para obtener los agregados adicionales requeridos y para ejecutar la lógica de dominio de efectos secundarios. El siguiente [código de controlador de eventos de dominio en eShopOnContainers](#) muestra un ejemplo de implementación.

```
clase pública ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : INotificationHandler<OrderStartedDomainEvent>
{
    privado de solo lectura ILoggerFactory _logger;
    privado de solo lectura IBuyerRepository<Comprador> _buyerRepository;
    privado de solo lectura IIdentityService _identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
        registrar ILoggerFactory,
        IBuyerRepository<Comprador> compradorRepositorio,
```

```

    IIdentityService servicio de identidad)
{
    // ...Validaciones de parámetros...
}

Manejador de tareas asíncronas públicas (OrderStartedDomainEvent orderStartedEvent) {

    var cardTypeId = (orderStartedEvent.CardTypeId != 0) ? orderStartedEvent.CardTypeId
    : 1;
    var userGuid = _identityService.GetUserIdentity(); var comprador =
    esperar _buyerRepository.FindAsync(userGuid); bool compradorOriginalmenteExiste
    = (comprador == nulo) ? falso : verdadero;

    if (!compradorExistióOriginalmente) {

        comprador = nuevo comprador (guidusuario);
    }

    comprador.VerificarOrAddPaymentMethod(cardTypeId,
        $"Método de pago en {DateTime.UtcNow}",
        orderStartedEvent.CardNumber,
        orderStartedEvent.CardSecurityNumber,
        orderStartedEvent.CardHolderName,
        orderStartedEvent.CardExpiration,
        orderStartedEvent.Order.Id);

    var compradorActualizado = compradorOriginalmenteExiste ? _buyerRepository.Update(comprador)
    :
    _buyerRepository.Add(comprador);

    esperar
        _buyerRepository.UnitOfWork .SaveEntitiesAsync();

    // Código de registro usando información actualizada del comprador, etc.
}
}

```

El código del controlador de eventos de dominio anterior se considera código de capa de aplicación porque utiliza repositorios de infraestructura, como se explica en la siguiente sección sobre la capa de persistencia de infraestructura.

Los controladores de eventos también podrían usar otros componentes de infraestructura.

Los eventos de dominio pueden generar eventos de integración que se publicarán fuera de los límites del microservicio

Por último, es importante mencionar que, en ocasiones, es posible que desee propagar eventos en varios microservicios. Esta propagación es un evento de integración y podría publicarse a través de un bus de eventos desde cualquier controlador de eventos de dominio específico.

Conclusiones sobre eventos de dominio

Como se indicó, use eventos de dominio para implementar explícitamente los efectos secundarios de los cambios dentro de su dominio. Para usar la terminología de DDD, use eventos de dominio para implementar explícitamente efectos secundarios en uno o varios agregados. Además, y para una mejor escalabilidad y un menor impacto en los bloqueos de bases de datos, utilice la coherencia final entre agregados dentro del mismo dominio.

La aplicación de referencia usa [MediatR](#) para propagar eventos de dominio sincrónicamente entre agregados, dentro de una sola transacción. Sin embargo, también podría usar alguna implementación de AMQP como [RabbitMQ](#) o [Azure Service Bus](#) para propagar eventos de dominio de forma asíncrona, usando coherencia eventual pero, como se mencionó anteriormente, debe considerar la necesidad de acciones compensatorias en caso de fallas.

Recursos adicionales

- Greg joven. ¿Qué es un evento de dominio?
https://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf#page=25
- Jan Stenberg. Eventos de dominio y consistencia eventual <https://www.infoq.com/news/2015/09/domain-events-consistency>
- Jimmy Bogard. Un mejor patrón de eventos de dominio
<https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/>
- Vaughn Vernon. Diseño efectivo de agregados Parte II: hacer que los agregados funcionen juntos https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- Jimmy Bogard. Fortalecimiento de su dominio: Eventos de dominio <https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/>
- Tony Truong. Ejemplo de patrón de eventos de dominio
<https://www.tonytruong.net/dominio-eventos-patrón-ejemplo/>
- Udi Dahan. Cómo crear modelos de dominio totalmente encapsulados
<https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>
- Udi Dahan. Eventos de dominio: toma 2
<https://udidahan.com/2008/08/25/dominio-eventos-toma-2/>
- Udi Dahan. Eventos de Dominio – Salvación
<https://udidahan.com/2009/06/14/domain-events-salvation/>
- Jan Kronquist. ¡No publique Eventos de Dominio, devuélvelos! <https://blog.jayway.com/2013/06/20/dont-publish-domain-events-return-them/>
- César de la Torre. Eventos de dominio frente a eventos de integración en DDD y arquitecturas de microservicios
<https://devblogs.microsoft.com/cesardelatorre/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/>

Diseñar la capa de persistencia de la infraestructura

Los componentes de persistencia de datos brindan acceso a los datos alojados dentro de los límites de un microservicio (es decir, la base de datos de un microservicio). Contienen la implementación real de componentes como repositorios y clases [de Unidad de trabajo](#), como Entity Framework (EF) personalizado [Objetos DbContext](#). EF DbContext implementa tanto el repositorio como los patrones de unidad de trabajo.

El patrón del repositorio

Los repositorios son clases o componentes que encapsulan la lógica necesaria para acceder a las fuentes de datos. Centralizan la funcionalidad de acceso a datos comunes, lo que proporciona una mejor capacidad de mantenimiento y desacopla la infraestructura o la tecnología utilizada para acceder a las bases de datos desde la capa del modelo de dominio. Si usa un Mapeador relacional de objetos (ORM) como Entity Framework, el código que debe implementarse se simplifica, gracias a LINQ y la tipificación fuerte. Esto le permite concentrarse en la lógica de persistencia de datos en lugar de en las tuberías de acceso a datos.

El patrón Repository es una forma bien documentada de trabajar con una fuente de datos. 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 del modelo de dominio y el mapeo de datos, actuando de manera similar a un conjunto de objetos de dominio en la memoria. Los objetos del cliente construyen consultas declarativamente y las envían a los repositorios para obtener respuestas. Conceptualmente, un repositorio encapsula un conjunto de objetos almacenados en la base de datos y las operaciones que se pueden realizar sobre ellos, proporcionando una forma más cercana a la capa de persistencia. Los repositorios, además, soportan el propósito de separar, de forma clara y unidireccional, la dependencia entre el dominio de trabajo y la asignación o mapeo de datos.

Definir un repositorio por agregado

Para cada agregado o raíz agregada, debe crear una clase de repositorio. En un microservicio basado en patrones de diseño controlado por dominio (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 del agregado, que controla las invariantes del agregado y la consistencia transaccional. Está bien consultar la base de datos a través de otros canales (como puede hacerlo siguiendo un enfoque CQRS), porque las consultas no cambian el estado de la base de datos. Sin embargo, el área transaccional (es decir, las actualizaciones) siempre debe estar controlada por los repositorios y las raíces agregadas.

Básicamente, un repositorio le permite llenar datos en la memoria que provienen de la base de datos en forma de entidades de dominio. Una vez que las entidades están en la memoria, se pueden cambiar y luego persistir en la base de datos a través de transacciones.

Como se señaló anteriormente, si usa el patrón arquitectónico CQS/CQRS, las consultas iniciales se realizan mediante consultas secundarias fuera del modelo de dominio, realizadas mediante declaraciones SQL simples que usan Dapper. Este enfoque es mucho más flexible que los repositorios porque puede consultar y unir cualquier tabla que necesite, y estas consultas no están restringidas por reglas de los agregados. Esos datos van a la capa de presentación o aplicación cliente.

Si el usuario realiza cambios, los datos que se actualizarán provienen de la aplicación del cliente o de la capa de presentación a la capa de la aplicación (como un servicio API web). Cuando recibe un comando en un controlador de comandos, utiliza repositorios para obtener los datos que desea actualizar de la base de datos. Lo actualiza en la memoria con los datos pasados con los comandos y luego agrega o actualiza los datos (entidades de dominio) en la base de datos a través de una transacción.

Es importante enfatizar nuevamente que solo debe definir un repositorio para cada raíz agregada, como se muestra en la Figura 7-17. Para lograr el objetivo de la raíz agregada de mantener la consistencia transaccional entre todos los objetos dentro del agregado, nunca debe crear un repositorio para cada tabla en la base de datos.

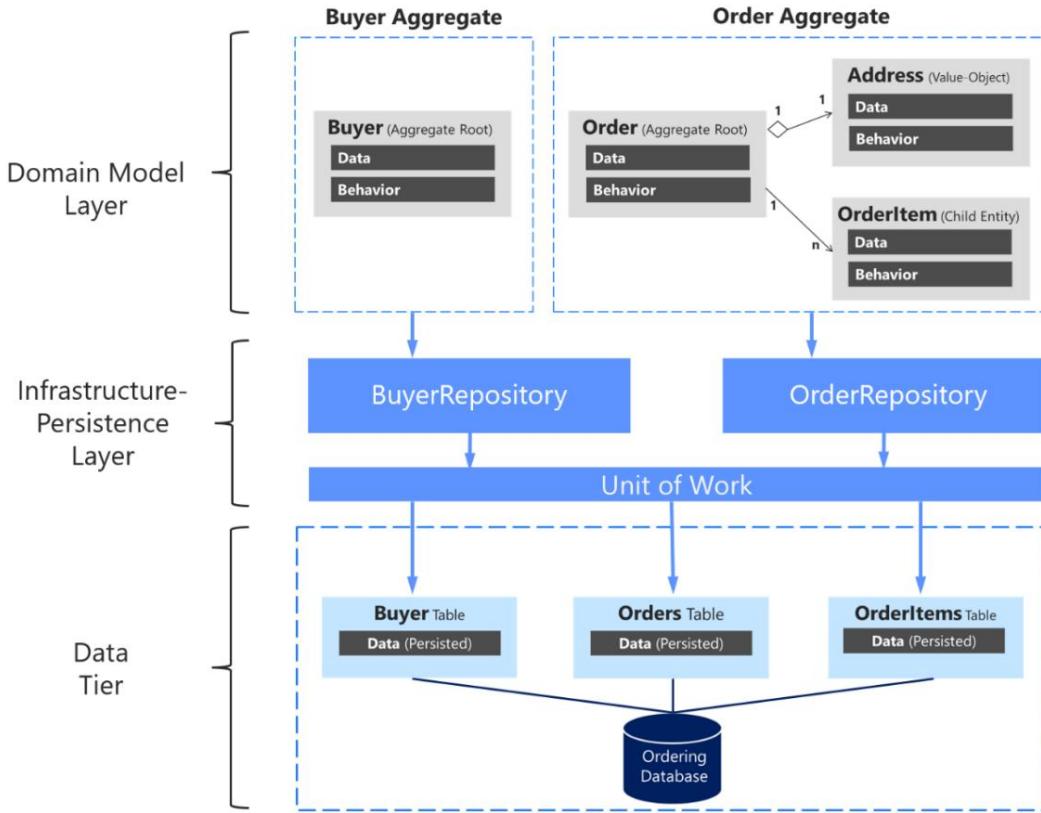


Figura 7-17. La relación entre repositorios, agregados y tablas de bases de datos

El diagrama anterior muestra las relaciones entre las capas de Dominio e Infraestructura: Buyer Aggregate depende de IBuyerRepository y Order Aggregate depende de las interfaces IOrderRepository, estas interfaces están implementadas en la capa de Infraestructura por los repositorios correspondientes que dependen de UnitOfWork, también implementado allí, que accede las tablas en el nivel de datos.

Hacer cumplir una raíz agregada por repositorio

Puede ser valioso implementar el diseño de su repositorio de tal manera que aplique la regla de que solo las raíces agregadas deben tener repositorios. Puede crear un tipo de repositorio base o genérico que restrinja el tipo de entidades con las que trabaja para garantizar que tengan la interfaz de marcador IAggregateRoot .

Por lo tanto, cada clase de repositorio implementada en la capa de infraestructura implementa su propio contrato o interfaz, como se muestra en el siguiente código:

```
espacio de nombres Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories {
    clase pública OrderRepository : IOrderRepository { // ...
    }
}
```

Cada interfaz de repositorio específica implementa la interfaz IRepository genérica:

```
interfaz pública IOrderRepository: IRepository<Pedido> {
    Agregar orden (pedir
    orden); // ...
}
```

Sin embargo, una mejor manera de hacer que el código haga cumplir la convención de que cada repositorio está relacionado con un solo agregado es implementar un tipo de repositorio genérico. De esa forma, es explícito que está utilizando un repositorio para apuntar a un agregado específico. Eso se puede hacer fácilmente implementando una interfaz base genérica de IRepository , como en el siguiente código:

```
interfaz pública IRepository<T> donde T : IAggregateRoot {
    //...
}
```

El patrón Repositorio facilita la prueba de la lógica de su aplicación

El patrón Repositorio le permite probar fácilmente su aplicación con pruebas unitarias. Recuerde que las pruebas unitarias solo prueban su código, no la infraestructura, por lo que las abstracciones del repositorio facilitan el logro de ese objetivo.

Como se indicó en una sección anterior, se recomienda que defina y coloque las interfaces del repositorio en la capa del modelo de dominio para que la capa de la aplicación, como su microservicio Web API, no dependa directamente de la capa de infraestructura donde implementó las clases de repositorio. Al hacer esto y usar la inyección de dependencia en los controladores de su API web, puede implementar repositorios simulados que devuelvan datos falsos en lugar de datos de la base de datos. Este enfoque desacoplado le permite crear y ejecutar pruebas unitarias que se centran en la lógica de su aplicación sin necesidad de conectividad a la base de datos.

Las conexiones a las bases de datos pueden fallar y, lo que es más importante, ejecutar cientos de pruebas contra una base de datos es malo por dos razones. Primero, puede llevar mucho tiempo debido a la gran cantidad de pruebas.

En segundo lugar, los registros de la base de datos pueden cambiar y afectar los resultados de sus pruebas, por lo que es posible que no sean consistentes. La prueba contra la base de datos no es una prueba unitaria sino una prueba de integración. Debería tener muchas pruebas unitarias ejecutándose rápidamente, pero menos pruebas de integración contra las bases de datos.

En términos de separación de preocupaciones para pruebas unitarias, su lógica opera en entidades de dominio en la memoria. Asume que la clase del repositorio los ha entregado. Una vez que su lógica modifica las entidades de dominio, asume que la clase de repositorio las almacenará correctamente. El punto importante aquí es crear pruebas unitarias contra su modelo de dominio y su lógica de dominio.

Las raíces agregadas son los principales límites de consistencia en DDD.

Los repositorios implementados en eShopOnContainers se basan en la implementación DbContext de EF Core de los patrones Repositorio y Unidad de trabajo mediante su rastreador de cambios, por lo que no duplican esta funcionalidad.

La diferencia entre el patrón de repositorio y la clase de acceso a datos heredada (clase DAL) patrón

Un objeto de acceso a datos realiza directamente operaciones de acceso y persistencia de datos contra el almacenamiento. Un repositorio marca los datos con las operaciones que se quieren realizar en la memoria de una unidad de trabajo

objeto (como en EF cuando se usa la clase [DbContext](#)), pero estas actualizaciones no se realizan inmediatamente en la base de datos.

Se hace referencia a una unidad de trabajo como una sola transacción que involucra múltiples operaciones de inserción, actualización o eliminación. En términos simples, significa que para una acción específica del usuario, como el registro en un sitio web, todas las operaciones de inserción, actualización y eliminación se manejan en una sola transacción. Esto es más eficiente que manejar múltiples transacciones de bases de datos de una manera más conversacional.

Estas operaciones de persistencia múltiple se realizan más tarde en una sola acción cuando su código de la capa de aplicación lo ordena. La decisión sobre la aplicación de los cambios en la memoria al almacenamiento real de la base de datos generalmente se basa en el [patrón de Unidad de trabajo](#). En EF, el patrón [Unidad de trabajo](#) se implementa como [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. También reduce el bloqueo de transacciones en las tablas de la base de datos, porque todas las operaciones previstas se confirman 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 puede optimizar la ejecución contra la base de datos agrupando varias acciones de actualización dentro de la misma transacción, a diferencia de muchas ejecuciones de transacciones pequeñas y separadas.

[Los repositorios no deberían ser obligatorios](#)

Los repositorios personalizados son útiles por las razones citadas anteriormente, y ese es el enfoque para el microservicio de pedidos en eShopOnContainers. Sin embargo, no es un patrón esencial para implementar en un diseño DDD o incluso en el desarrollo general de .NET.

Por ejemplo, Jimmy Bogard, al proporcionar comentarios directos para esta guía, dijo lo siguiente:

Este será probablemente mi mayor comentario. Realmente no soy un fanático de los repositorios, principalmente porque ocultan los detalles importantes del mecanismo de persistencia subyacente. Es por eso que también elijo MediatR para los comandos. Puedo usar todo el poder de la capa de persistencia e impulsar todo ese comportamiento de dominio en mis raíces agregadas. Por lo general, no quiero burlarme de mis repositorios; todavía necesito tener esa prueba de integración con la cosa real. Pasar a CQRS significó que ya no necesitábamos repositorios.

Los repositorios pueden ser útiles, pero no son críticos para el diseño de su DDD, como lo son el patrón Agregado y el modelo de dominio enriquecido. Por lo tanto, use el patrón Repository o no, según le parezca. De todos modos, usará el patrón de repositorio cada vez que use EF Core aunque, en este caso, el repositorio cubre todo el microservicio o el contexto limitado.

[Recursos adicionales](#)

[Patrón de repositorio](#)

- Edward Hieatt y Rob Mee. Patrón de repositorio. <https://martinfowler.com/eaaCatalog/repository.html>

- El patrón Repositorio [https://docs.microsoft.com/visiones-anteriores/msp-np/ff649690\(v=pandp.10\)](https://docs.microsoft.com/visiones-anteriores/msp-np/ff649690(v=pandp.10))

- Erick Evans. Diseño impulsado por dominio: abordar la complejidad en el corazón del software. (Libro; incluye una discusión sobre el patrón Repository) <https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>
-

Patrón de unidad de trabajo

- Martín Fowler. Patrón de unidad de trabajo.
<https://martinfowler.com/eaaCatalog/unitOfWork.html>
 - Implementación de patrones de repositorio y unidad de trabajo en una aplicación ASP.NET MVC <https://docs.microsoft.com/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementation-of-repository-and-unit-of-work-patterns-in-an-asp-net-mvc>
-

Implemente la capa de persistencia de la infraestructura con Núcleo del marco de la entidad

Cuando usa 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 es compatible con 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, también debe usar Entity Framework Core, que se ejecuta en Windows o Linux de la misma manera que .NET. EF Core es una reescritura completa de Entity Framework que se implementa con una huella mucho más pequeña e importantes mejoras en el rendimiento.

Introducción a Entity Framework Core

Entity Framework (EF) Core es una versión liviana, extensible y multiplataforma de la popular tecnología de acceso a datos de Entity Framework. Se introdujo con .NET Core a mediados de 2016.

Dado que una introducción a EF Core ya está disponible en la documentación de Microsoft, aquí simplemente proporcionamos enlaces a esa información.

Recursos adicionales

- Núcleo de Entity Framework
<https://docs.microsoft.com/ef/core/>
- Introducción a ASP.NET Core y Entity Framework Core con Visual Studio <https://docs.microsoft.com/aspnet/core/data/ef-mvc/>
- Clase DbContext
<https://docs.microsoft.com/dotnet/api/microsoft.entityframeworkcore.dbcontext>
- Compare EF Core y EF6.x <https://docs.microsoft.com/ef/efcore-and-ef6/index>

Infraestructura en Entity Framework Core desde una perspectiva DDD

Desde el punto de vista de DDD, una capacidad importante de EF es la capacidad de usar entidades de dominio POCO, también conocidas en la terminología de EF como entidades de código POCO primero. Si usa entidades de dominio POCO, sus clases de modelo de dominio ignoran la persistencia, siguiendo los principios de Ignorancia de persistencia e Ignorancia de infraestructura.

Según los patrones de DDD, debe encapsular el comportamiento y las reglas del dominio dentro de la propia clase de entidad, de modo que pueda controlar los invariantes, las validaciones y las 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 u objetos de valor. En su lugar, desea exponer métodos que controlen cómo y cuándo se pueden actualizar sus campos y colecciones de propiedades, y qué comportamiento y acciones deben ocurrir cuando eso sucede.

Desde EF Core 1.1, para satisfacer esos requisitos de DDD, puede tener campos sin formato en sus entidades en lugar de propiedades públicas. Si no desea que un campo de entidad sea accesible externamente, simplemente puede crear el atributo o campo en lugar de una propiedad. También puede utilizar setters de propiedad privada.

De manera similar, ahora puede tener acceso de solo lectura a las colecciones usando una propiedad pública escrita como `IReadOnlyCollection<T>`, que está respaldada por un miembro de campo privado para la colección (como `List<T>`) en su entidad que depende de EF para la persistencia. Las versiones anteriores de Entity Framework requerían propiedades de colección para admitir `ICollection<T>`, lo que significaba que cualquier desarrollador que usara la clase de entidad principal podía agregar o quitar elementos a través de sus colecciones de propiedades. Esa posibilidad estaría en contra de los patrones recomendados en DDD.

Puede usar una colección privada mientras expone un objeto `IReadOnlyCollection<T>` de solo lectura, como se muestra en el siguiente ejemplo de código:

```
Orden de clase pública : Entidad {

    // Usar campos privados, permitido desde EF Core 1.1 private
    DateTime _orderDate; // Otros campos ...

    lista privada de solo lectura <Artículo de pedido> _artículos
    de pedido; public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    Orden protegida () {}

    orden pública (int compradorId, int pagoMethodId, Dirección dirección) {

        // Inicializaciones...
    }

    public void AddOrderItem(int productId, string productName, decimal unitPrice,
                           decimal discount, string pictureUrl, int units = 1)

    {
        // Lógica de validación...

        var orderItem = new OrderItem(productId, productName, unitPrice,
                                      discount, pictureUrl, units);

        _orderItems.Add(orderItem);
    }
}
```

Solo se puede acceder a la propiedad OrderItems como de solo lectura mediante `IReadOnlyCollection < OrderItem >`. Este tipo es de solo lectura, por lo que está protegido contra actualizaciones externas periódicas.

EF Core proporciona una manera de asignar el modelo de dominio a la base de datos física sin "contaminar" el modelo de dominio. Es código .NET POCO puro, porque la acción de mapeo se implementa en la capa de persistencia. En esa acción de mapeo, debe configurar el mapeo de campos a base de datos. En el siguiente ejemplo del método `OnModelCreating` de `OrderingContext` y la clase `OrderEntityTypeConfiguration`, la llamada a `SetPropertyAccessMode` le indica a EF Core que acceda a la propiedad `OrderItems` a través de su campo.

```
// En OrderingContext.cs de eShopOnContainers protected
override void OnModelCreating(ModelBuilder modelBuilder) {

    // ...
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration()); // Configuración
    // de otras entidades...
}

// En OrderEntityTypeConfiguration.cs de eShopOnContainers clase
OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order> {

    public void Configure(EntityTypeBuilder<Order> orderConfiguration) {

        orderConfiguration.ToTable("pedidos", OrderingContext.DEFAULT_SCHEMA); // Otra
        // configuración

        var navegación =
            orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));

        // EF accede a la propiedad de la colección OrderItem a través de su campo de respaldo
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

        // Otra configuración
    }
}
```

Cuando usa campos en lugar de propiedades, la entidad `OrderItem` se mantiene como si tuviera una propiedad `List<OrderItem>`. Sin embargo, expone un acceso único, el método `AddOrderItem`, para agregar nuevos artículos al pedido. Como resultado, el comportamiento y los datos están vinculados y serán consistentes en cualquier código de aplicación que use el modelo de dominio.

Implementar repositorios personalizados con Entity Framework Core

A nivel de implementación, un repositorio es simplemente una clase con código de persistencia de datos coordinado por una unidad de trabajo (`DbContext` en EF Core) al realizar actualizaciones, como se muestra en la siguiente clase:

```
// usar directivas... espacio
de nombres Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories {

    clase pública BuyerRepository : IBuyerRepository {

        privado de solo lectura OrderingContext _context;
        public IUnidadDeTrabajo UnidadDeTrabajo {
```

```

        conseguir
    {
        volver _contexto;
    }
}

BuyerRepository público (contexto de OrderingContext) {

    _contexto = contexto ?? lanzar una nueva excepción ArgumentNullException (nombre de (contexto));
}

Public Buyer Add (Comprador comprador)
{
    return _context.Buyers.Add(comprador).Entidad;
}

tarea asíncrona pública <comprador> FindAsync (string compradorIdentityGuid) {

    var comprador = esperar
        _contexto.Compradores .Incluir (b => b.Pagos)
        .Dónde(b => b.Nombre completo == ID de identidad del comprador)
        .SingleOrDefaultAsync();

    comprador de devolución ;
}
}
}

```

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 EF `DbContext` llega a través del constructor a través de la inyección de dependencia. Se comparte entre varios repositorios dentro del mismo ámbito de solicitud HTTP, gracias a su duración predeterminada (`ServiceLifetime.Scoped`) en el contenedor IoC (que también se puede configurar explícitamente con `services.AddDbContext<>`).

Métodos a implementar en un repositorio (actualizaciones o transacciones versus consultas)

Dentro de cada clase de repositorio, debe colocar los métodos de persistencia que actualizan el estado de las entidades contenidas por su agregado relacionado. Recuerde que existe una relación de uno a uno entre un agregado y su repositorio relacionado. Considere que un objeto de entidad raíz agregada podría tener entidades secundarias incrustadas dentro de su gráfico EF. Por ejemplo, un comprador puede tener varios métodos de pago como entidades secundarias relacionadas.

Dado que 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 la libertad de crear las consultas y uniones que necesitan para la capa de presentación sin las restricciones impuestas por agregados, repositorios personalizados por agregado y DDD en general. La mayoría de los repositorios personalizados sugeridos por esta guía tienen varios métodos transaccionales o de actualización, pero solo 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 obtener datos para enviar a la capa de presentación o aplicaciones cliente se implementan, como se mencionó, en las consultas CQRS basadas en consultas flexibles usando Dapper.

Usar un repositorio personalizado versus usar EF DbContext directamente

La clase Entity Framework DbContext se basa en los patrones de unidad de trabajo y repositorio y se puede usar directamente desde su código, como desde un controlador ASP.NET Core MVC. Los patrones de Unidad de trabajo y Repositorio dan como resultado el código más simple, como en el microservicio de catálogo CRUD en eShopOnContainers. En los casos en los que desee el código más simple posible, es posible que desee utilizar directamente la clase DbContext, como hacen muchos desarrolladores.

Sin embargo, la implementación de repositorios personalizados brinda varios beneficios al implementar aplicaciones o microservicios más complejos. Los patrones de Unidad de trabajo y Repositorio están destinados a encapsular la capa de persistencia de la infraestructura para que se desacople de las capas de modelo de aplicación y dominio. La implementación de estos patrones puede facilitar el uso de repositorios simulados que simulan el acceso a la base de datos.

En la Figura 7-18, puede ver las diferencias entre no usar repositorios (usando directamente EF DbContext) y usar repositorios, lo que facilita la simulación de esos repositorios.

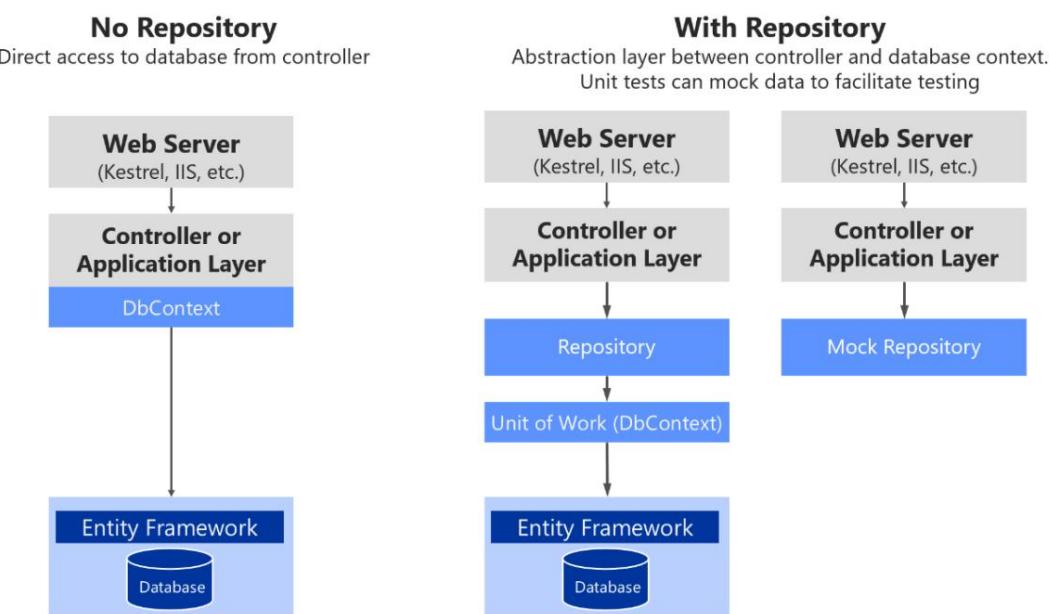


Figura 7-18. Uso de repositorios personalizados frente a un DbContext simple

La Figura 7-18 muestra que el uso de un repositorio personalizado agrega una capa de abstracción que se puede usar para facilitar las pruebas simulando el repositorio. Existen múltiples alternativas a la hora de burlarse. Puede simular solo repositorios o puede simular una unidad de trabajo completa. Por lo general, simular solo los repositorios es suficiente, y la complejidad de abstraer y simular una unidad de trabajo completa generalmente no es necesaria.

Más adelante, cuando nos centremos en la capa de aplicación, verá cómo funciona la inyección de dependencia en ASP.NET Core y cómo se implementa cuando se usan repositorios.

En resumen, los repositorios personalizados le permiten probar el código más fácilmente con pruebas unitarias que no se ven afectadas por el estado del nivel de datos. Si ejecuta pruebas que también acceden a la base de datos real a través de Entity Framework, no son pruebas unitarias sino pruebas de integración, que son mucho más lentas.

Si estuviera usando DbContext directamente, tendría que simularlo o ejecutar pruebas unitarias usando un SQL Server en memoria con datos predecibles para pruebas unitarias. Pero burlarse de DbContext o controlar datos falsos requiere más trabajo que burlarse a nivel de repositorio. Por supuesto, siempre puedes probar los controladores MVC.

Duración de la instancia de EF DbContext y IUnitOfWork en su contenedor IoC

El objeto DbContext (expuesto como un objeto IUnitOfWork) debe compartirse entre varios repositorios dentro del mismo ámbito de solicitud HTTP. Por ejemplo, esto es cierto cuando la operación que se ejecuta debe tratar con varios agregados, o simplemente porque está utilizando varias instancias de repositorio. También es importante mencionar que la interfaz IUnitOfWork es parte de su capa de dominio, no un tipo de EF Core.

Para hacer eso, la instancia del objeto DbContext debe tener la duración del servicio configurada en ServiceLifetime.Scoped. Esta es la vida útil predeterminada al registrar un DbContext con services.AddDbContext en su contenedor IoC desde el método ConfigureServices del archivo Startup.cs en su proyecto ASP.NET Core Web API. El siguiente código ilustra esto.

```
public IServiceProvider ConfigureServices(servicios IServiceCollection) {
    // Agregar servicios de marco.
    servicios.AddMvc(opciones => {

        opciones.Filters.Add(typeof(HttpGlobalExceptionFilter));
    }).AddControllersAsServices();

    servicios.AddEntityFrameworkSqlServer()
        .AddDbContext<ContextoPedido>(opciones => {

            opciones.UseSqlServer(Configuration["ConnectionString"], sqlOptions =>
                sqlOptions.MigrationsAssembly(typeof(Startup).GetTypeInfo().
                    Asamblea.GetName().Nombre));
        });

    servicios.AddScoped // 
        // Tenga en cuenta que Scoped es la opción predeterminada
        // en AddDbContext. Se muestra aquí sólo con fines //
        // pedagógicos.
    );
}
```

El modo de creación de instancias de DbContext no debe configurarse como ServiceLifetime.Transient o ServiceLifetime.Singleton.

La vida útil de la instancia del repositorio en su contenedor IoC

De manera similar, la vida útil del repositorio generalmente debe establecerse como alcance (InstancePerLifetimeScope en Autofac). También podría ser transitorio (InstancePerDependency en Autofac), pero su servicio será más eficiente en lo que respecta a la memoria cuando use la duración del ámbito.

```
// Registro de un repositorio en Autofac IoC container
builder.RegisterType<OrderRepository>()
    .As<IOrderRepository>()
    .InstancePerLifetimeScope();
```

El uso de la vida útil de singleton para el repositorio podría causarle serios problemas de simultaneidad cuando su DbContext se establece en la vida útil con ámbito (InstancePerLifetimeScope) (la vida útil predeterminada para un DbContext).

Recursos adicionales

- Implementación de patrones de repositorio y unidad de trabajo en una aplicación ASP.NET MVC <https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc/4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>
- Jonathan Allen. Estrategias de implementación para el patrón de repositorio con Entity Framework, Dapper y Chain <https://www.infoq.com/articles/repository-implementation-strategies>
- César de la Torre. Comparación de la vida útil del servicio de contenedor de ASP.NET Core IoC con los ámbitos de instancia de contenedor de Autofac IoC <https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-alcances/>

Mapeo de tablas

El mapeo de tablas identifica los datos de la tabla que se consultarán y guardarán en la base de datos. Anteriormente vio cómo las entidades de dominio (por ejemplo, un dominio de producto o pedido) se pueden usar 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 suelen basarse en nombres convencionales. Por ejemplo, es típico que la clave principal sea una propiedad que termine con Id.

Por convención, cada entidad se configurará para asignarse a una tabla con el mismo nombre que la propiedad DbSet< TEntity > que expone la entidad en el contexto derivado. Si no se proporciona ningún valor DbSet< TEntity > para la entidad dada, se usa el nombre de la clase.

Anotaciones de datos versus API fluida

Hay muchas convenciones adicionales de EF Core, y la mayoría de ellas se pueden cambiar mediante anotaciones de datos o API Fluent, implementadas dentro del método OnModelCreating.

Las anotaciones de datos deben usarse en las propias clases del modelo de entidad, que es una forma más intrusiva desde el punto de vista de 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, Fluent API es una forma conveniente de cambiar la mayoría de las convenciones y asignaciones 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.

API fluida y el método OnModelCreating

Como se mencionó, para cambiar las convenciones y las asignaciones, puede usar el método `OnModelCreating` en la clase `DbContext`.

El microservicio de pedidos en `eShopOnContainers` implementa el mapeo y la configuración explícitos, cuando es necesario, como se muestra en el siguiente código.

```
// En OrderingContext.cs de eShopOnContainers protected
override void OnModelCreating(ModelBuilder modelBuilder) {

    // ...
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration()); // Configuración de
    otras entidades...
}

// En OrderEntityTypeConfiguration.cs de eShopOnContainers clase
OrderEntityTypeConfiguration : IEntityEntityTypeConfiguration<Order> {

    public void Configure(EntityTypeBuilder<Order> orderConfiguration) {

        orderConfiguration.ToTable("pedidos", OrderingContext.DEFAULT_SCHEMA);

        orderConfiguration.HasKey(o => o.Id);

        orderConfiguration.Ignore(b => b.DomainEvents);

        orderConfiguration.Property(o => o.Id)
            .UseHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

        //El objeto de valor de dirección persiste como tipo de entidad de propiedad compatible desde EF Core 2.0
        orderConfiguration.OwnsOne(o => o.Address, a => {

            a.ConPropietario();
        });

        orderConfiguration.Property
            <int?>("_buyerId")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("CompradorId")
            .EsRequerido(falso);

        orderConfiguration.Property
            <DateTime?>("_orderDate")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("FechaPedido")
            .SeRequiere();

        orderConfiguration.Property
            <int?>("_orderId")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("OrderId")
            .SeRequiere();

        orderConfiguration.Property
            <int?>("_paymentMethodId")
            .UsePropertyAccessMode(PropertyAccessMode.Field)
            .HasColumnName("PaymentMethodId")
            .EsRequerido(falso);
    }
}
```

```

orderConfiguration.Property<cadena>("Descripción").IsRequired(false);

var navigation =
orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));

// Comentario de patrones DDD: //
Establecer como campo (Nuevo desde EF 1.1) para acceder a la propiedad de la colección OrderItem a
través de su campo de navegación SetPropertyAccessMode(PropertyAccessMode.Field);

orderConfiguration.HasOne<PaymentMethod>()
    .Con muchas()
    .HasForeignKey("_IdMetodoDePago")
    .EsRequerido(falso)
    .OnDelete(DeleteBehavior.Restrict);

orderConfiguration.HasOne<Comprador>()
    .Con muchas()
    .EsRequerido(falso)
    .HasForeignKey("_IdComprador");

orderConfiguration.HasOne(o => o.OrderStatus)
    .Con muchas()
    .HasForeignKey("_orderId");
}

}

```

Puede establecer todas las asignaciones de API de Fluent dentro del mismo método `OnModelCreating`, pero es recomendable particionar ese código y tener varias clases de configuración, una por entidad, como se muestra en el ejemplo. Especialmente para modelos grandes, es recomendable tener clases de configuración separadas para configurar diferentes tipos de entidades.

El código del ejemplo muestra algunas declaraciones y asignaciones explícitas. 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 del ejemplo anterior es que utiliza el [algoritmo Hi/Lo](#) como [estrategia de generación](#) de claves.

El algoritmo Hi/Lo es útil cuando necesita claves únicas antes de realizar cambios. En resumen, el algoritmo Hi-Lo asigna identificadores únicos a las filas de la tabla sin depender del almacenamiento inmediato de la fila en la base de datos. Esto le permite comenzar a usar los identificadores de inmediato, como sucede con los ID de bases de datos secuenciales regulares.

El algoritmo Hi/Lo describe un mecanismo para obtener un lote de ID únicos de una secuencia de base de datos relacionada. Estos ID son seguros de usar porque la base de datos garantiza la unicidad, por lo que no habrá colisiones entre los usuarios. Este algoritmo es interesante por estas razones:

- No rompe el patrón de Unidad de Trabajo.
- Obtiene ID de secuencia en lotes, para minimizar los viajes de ida y vuelta a la base de datos.
- Genera un identificador legible por humanos, a diferencia de las técnicas que usan GUID.

EF Core admite [HiLo](#) con el método `UseHiLo`, como se muestra en el ejemplo anterior.

Asignar campos en lugar de propiedades

Con esta función, disponible desde EF Core 1.1, puede asignar directamente columnas a campos. Es posible no usar propiedades en la clase de entidad y solo asignar columnas de una tabla a campos. Un uso común para eso serían campos privados para cualquier estado interno al que no sea necesario acceder desde fuera de la entidad.

Puede hacer esto con campos individuales o también con colecciones, como un campo `List<>`. Este punto se mencionó anteriormente cuando discutimos el modelado de las clases del modelo de dominio, pero aquí puede ver cómo se realiza esa asignación con la configuración de `PropertyAccessMode.Field` resaltada en el código anterior.

Usar propiedades ocultas en EF Core, ocultas a nivel de infraestructura

Las propiedades de sombra en EF Core son propiedades que no existen en su modelo de clase de entidad. Los valores y estados de estas propiedades se mantienen puramente en la clase [ChangeTracker](#) a nivel de infraestructura.

Implementar el patrón de especificación de consulta

Como se presentó anteriormente en la sección de diseño, el patrón de especificación de consulta es un patrón de diseño controlado por dominio diseñado como el lugar donde puede colocar la definición de una consulta con lógica de ordenación y paginación opcional.

El patrón de especificación de consulta define una consulta en un objeto. Por ejemplo, para encapsular una consulta paginada que busca algunos productos, puede crear una especificación `PagedProduct` que tome los parámetros de entrada necesarios (número de página, tamaño de página, filtro, etc.). Luego, dentro de cualquier método de Repositorio (generalmente una sobrecarga `List()`) aceptaría una `IQuerySpecification` y ejecutaría la consulta esperada basada en esa especificación.

Un ejemplo de una interfaz de especificación genérica es el siguiente código, que es similar al código utilizado en la [aplicación de referencia eShopOnWeb](#).

```
// INTERFAZ DE ESPECIFICACIONES
GENÉRICAS // https://github.com/dotnet-architecture/eShopOnWeb

interfaz pública ISpecification<T> {

    Expression<Func<T, bool>> Criterios { get; }
    List<Expression<Func<T, object>>> Incluye { get; }
    Lista<cadena> IncluirCadenas { get; }
}
```

Entonces, la implementación de una clase base de especificación genérica es la siguiente.

```
// IMPLEMENTACIÓN DE ESPECIFICACIONES GENÉRICAS (CLASE BASE) //
https://github.com/dotnet-architecture/eShopOnWeb

clase abstracta pública BaseSpecification<T> : ISpecification<T> {

    Public BaseSpecification(Expression<Func<T, bool>> criterio)
```

```

{
    Criterios = criterios;

} Expresión pública <Func<T, bool>> Criterios { get; }

public List<Expression<Func<T, object>>> Incluye { get; } = nueva Lista<Expresión<Func<T,
    objeto>>>();

public List<cadena> Incluir Cadenas { get; } = nueva Lista<cadena>();

vacío virtual protegido AddInclude(Expression<Func<T, object>> includeExpression) {

    Incluye.Añadir(incluirExpresión);
}

// las inclusiones basadas en cadenas permiten incluir hijos de hijos // por ejemplo,
Basket.Items.Product protected virtual void AddInclude(string includeString) {

    Incluir cadenas. Agregar (incluir cadena);
}
}

```

La siguiente especificación carga una única entidad de cesta dada la ID de la cesta o la ID del comprador al que pertenece la cesta. Cargará ansiosamente la colección de artículos de la cesta .

```

// MUESTRA DE IMPLEMENTACIÓN DE ESPECIFICACIÓN DE CONSULTA

Clase pública BasketWithItemsSpecification: BaseSpecification<Basket> {

    BasketWithItemsSpecification público (int basketId) : base (b => b.Id ==
        basketId)
    {
        AddInclude(b => b.Artículos);
    }

    public BasketWithItemsSpecification(cadena ID del comprador) :
        base(b => b.Id del comprador == Id del comprador)
    {
        AddInclude(b => b.Artículos);
    }
}

```

Y finalmente, puede ver a continuación cómo un Repositorio de EF genérico puede usar dicha especificación para filtrar y cargar datos relacionados con una entidad tipo T dada.

```

// REPOSITORIO EF GENÉRICO CON ESPECIFICACIONES //
https://github.com/dotnet-architecture/eShopOnWeb

public IEnumerable<T> List(I Specification<T> spec) {

    // buscar un IQueryable que incluya todas las inclusiones basadas en expresiones var
    queryableResultWithIncludes = spec.Includes
        .Agregado(_dbContext.Set<T>().AsQueryable(),
            (actual, incluir) => actual.Incluir(incluir));

    // modificar el IQueryable para incluir declaraciones de inclusión basadas en cadenas var secondResult =
    spec.IncludeStrings
        .Agregado(queryableResultWithIncludes, (actual, incluir)
            => actual.Incluir(incluir));
}

```

```
// devuelve el resultado de la consulta utilizando la expresión de criterios de la especificación return
secondResult .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 recomendamos devolver IQueryable desde un repositorio, está perfectamente bien usarlos dentro del repositorio para generar un conjunto de resultados. Puede ver este enfoque utilizado en el método List anterior, que utiliza expresiones IQueryable intermedias para crear la lista de consultas de inclusiones antes de ejecutar la consulta con los criterios de la especificación en la última línea.

Aprenda [cómo se aplica el patrón de especificación en el ejemplo de eShopOnWeb](#).

Recursos adicionales

- Asignación de tablas <https://docs.microsoft.com/ef/core/modeling/relational/tables>
- Use HiLo para generar claves con Entity Framework Core <https://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/>
- Campos de respaldo <https://docs.microsoft.com/ef/core/modeling/backing-field>
- Steve Smith. Colecciones encapsuladas en Entity Framework Core <https://ardalis.com/encapsulated-collections-in-entity-framework-core>
- Propiedades de sombra <https://docs.microsoft.com/ef/core/modeling/shadow-properties>
- El patrón de especificación <https://deviq.com/specification-pattern/>

Utilice bases de datos NoSQL como infraestructura de persistencia

Cuando usa bases de datos NoSQL para su nivel de datos de infraestructura, normalmente no usa 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 usa 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 cómo puede hacerlo en EF Core, en lo que respecta a la identificación de raíces agregadas, clases de entidades secundarias y clases de objetos de valor. Pero, en última instancia, la selección de la base de datos tendrá un impacto en su diseño.

Cuando usa una base de datos orientada a documentos, implementa un agregado como un solo documento, serializado en JSON u otro formato. Sin embargo, el uso de la base de datos es transparente desde el punto de vista del código del modelo de dominio. Cuando usa una base de datos NoSQL, todavía está usando clases de entidad y

clases raíz agregadas, pero con más flexibilidad que cuando se usa EF Core porque la persistencia no es relacional.

La diferencia está en cómo persistes en ese modelo. Si implementó su modelo de dominio basado en clases de entidad POCO, independiente de la persistencia de la infraestructura, podría parecer que podría pasar a una infraestructura de persistencia diferente, incluso de relacional a NoSQL. Sin embargo, ese no debe ser tu objetivo. Siempre existen restricciones y compensaciones en las diferentes tecnologías de base de datos, por lo que no podrá tener el mismo modelo para bases de datos relacionales o NoSQL. Cambiar los modelos de persistencia no es una tarea 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 agregada tenga varias propiedades de colección secundaria. En una base de datos relacional, la consulta de varias propiedades de colección secundaria no se optimiza fácilmente, ya que obtiene una instrucción UNION ALL SQL de EF. Tener el mismo modelo de dominio para bases de datos relacionales o bases de datos NoSQL no es sencillo y no debe intentar hacerlo. Realmente tiene que diseñar su modelo con una comprensión de cómo se utilizarán los datos en cada base de datos en particular.

Una ventaja de usar bases de datos NoSQL es que las entidades están más desnormalizadas, por lo que no establece una asignación de tabla. Su modelo de dominio puede ser más flexible que cuando usa una base de datos relacional.

Cuando diseña su modelo de dominio basado en agregados, pasar a NoSQL y bases de datos orientadas a documentos puede 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 una implementación de muestra de un pedido agregado cuando se usa una base de datos orientada a documentos. Es similar al agregado de pedidos que implementamos en la muestra de eShopOnContainers, pero sin usar EF Core debajo.

```
{
  "id": "2017001",
  "orderDate": "2/25/2017",
  "buyerId": "1234567", "address": [
    { "street": "100 One Microsoft
      Way", "city": "Redmond",
      "estado": "WA", "código postal": "98052",
      "país": "EE. UU." }

    ],
  "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, "descuento": 0} ]
}
```

Introducción a Azure Cosmos DB y 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 milisegundos de un solo dígito en el percentil 99, [cinco niveles de consistencia bien definidos](#) y alta disponibilidad garantizada, todo respaldado por [SLA líderes en la industria](#). Azure Cosmos DB [indexa automáticamente los datos](#) sin necesidad de que usted se ocupe de la administración de índices y esquemas. Es multimodelo y admite modelos de datos de documentos, valores clave, gráficos y columnas.

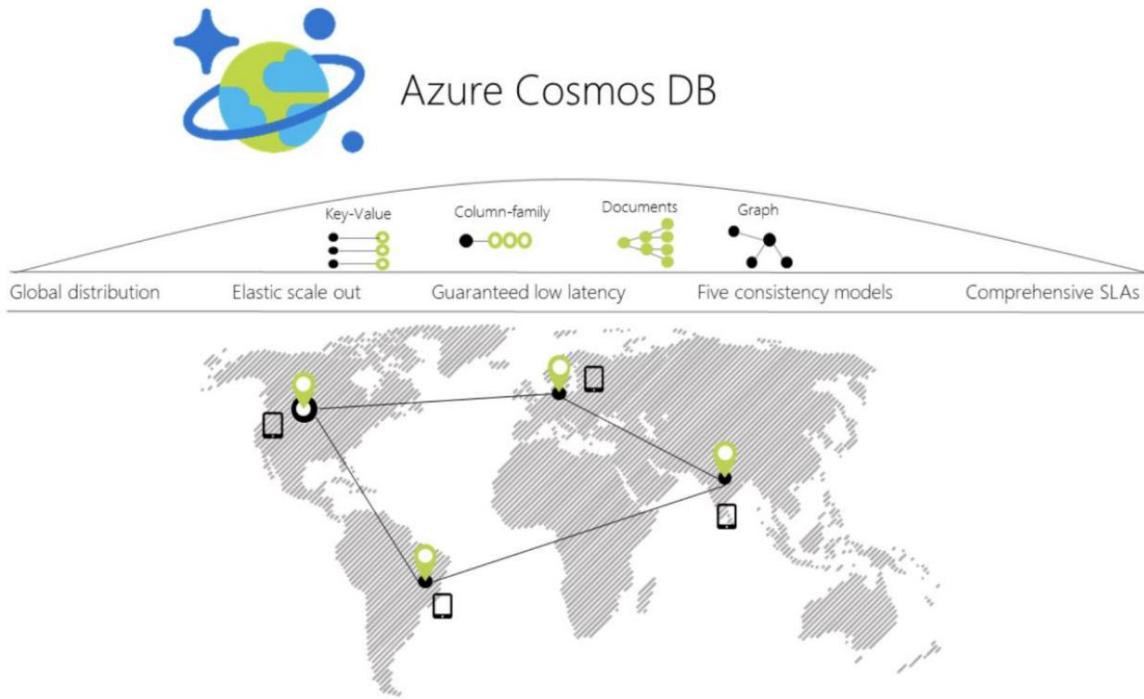


Figura 7-19. Distribución global de Azure Cosmos DB

Cuando usa un modelo de C# para implementar el agregado que utilizará la API de Azure Cosmos DB, el agregado puede ser similar a las clases POCO de C# que se usan con EF Core. La diferencia está en la forma de usarlos desde las capas de aplicación e infraestructura, como en el siguiente código:

```
// EJEMPLO DE C# DE UN AGREGADO DE PEDIDO QUE SE PERSISTE CON LA API DE BASE DE DATOS DE AZURE COSMOS // *** Código de modelo de dominio *** // Agregado: cree un objeto de pedido con sus entidades secundarias y/u objetos de valor.  
// Luego, use los métodos de AggregateRoot para agregar los objetos anidados para que los invariantes y // la lógica sea coherente en todas las propiedades anidadas (objetos de valor y entidades).
```

```
Pedido orderAggregate = nuevo Pedido {  
  
    Identificación = "2017001",  
    FechaPedido = nueva FechaHora (2005, 7, 1),  
    ID del comprador = "1234567",  
    Número de orden de compra = "PO18009186470"  
}  
  
Dirección dirección = nueva dirección {  
  
    Calle = "100 One Microsoft Way", Ciudad =  
    "Redmond", Estado = "WA", Código postal =  
    "98052", País = "EE. UU."
```

```

}

orderAggregate.UpdateAddress(dirección);

Artículo de pedido artículo de pedido1 = nuevo artículo de pedido
{
    Id = 20170011,
    ProductId = "123456",
    ProductName = ".NET T-Shirt", UnitPrice
    = 25, Units = 2, Discount = 0;

};

//Utilizando métodos con lógica de dominio dentro de la entidad. Sin modelo de dominio anémico
orderAggregate.AddOrderItem(orderItem1); // *** Fin del código del modelo de dominio **

// *** Código de infraestructura mediante la API de cliente de Cosmos DB ***
Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName,
    nombreColección);

espera cliente.CreateDocumentAsync(colecciónUri, orderAggregate);

// A medida que su aplicación evoluciona, digamos que su objeto tiene un nuevo esquema. Puede insertar //
objetos OrderV2 sin ningún cambio en el nivel de la base de datos.
Pedido2 nuevoPedido = GetOrderV2Sample("IdForSalesOrder2"); espera
cliente.CreateDocumentAsync(collectionUri, newOrder);

```

Puede ver que la forma en que trabaja con su modelo de dominio puede ser similar a la forma en que lo usa en su capa de modelo de dominio cuando la infraestructura es EF. Todavía usa los mismos métodos raíz agregados para garantizar la coherencia, las invariantes y las validaciones dentro del agregado.

Sin embargo, cuando persiste su modelo en la base de datos NoSQL, el código y la API cambian drásticamente en comparación con el código EF Core o cualquier otro código relacionado con las bases de datos relacionales.

Implementar código .NET dirigido a MongoDB y Azure Cosmos DB

Usar 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 bases de datos de Azure Cosmos DB.

Sin embargo, existe una limitación en Azure Cosmos DB desde el punto de vista del entorno de desarrollo de Docker.

Aunque hay un [emulador de Azure Cosmos DB local](#) que puede ejecutarse en una máquina de desarrollo local, solo es compatible con Windows. Linux y macOS no son compatibles.

También existe la posibilidad de ejecutar este emulador en Docker, pero solo en Windows Containers, no en Linux Containers. Esa es una desventaja inicial para el entorno de desarrollo si su aplicación se implementa como contenedores de Linux, ya que actualmente no puede implementar contenedores de Linux y Windows en Docker para Windows al mismo tiempo. Todos los contenedores que se implementen deben ser para Linux o para Windows.

La implementación ideal y más sencilla para una solución de desarrollo/prueba es poder implementar sus sistemas de base de datos como contenedores junto con sus contenedores personalizados para que sus entornos de desarrollo/prueba sean siempre coherentes.

[Utilice la API de MongoDB para contenedores locales de desarrollo/prueba de Linux/Windows más Azure Cosmos](#)

base de datos

Las bases de datos de Cosmos DB admiten la API de MongoDB para .NET, así como el protocolo de conexión nativo de MongoDB. Esto significa que al usar los controladores existentes, su aplicación escrita para MongoDB ahora puede comunicarse con Cosmos DB y usar bases de datos de Cosmos DB en lugar de bases de datos de MongoDB, como se muestra en la Figura 7-20.

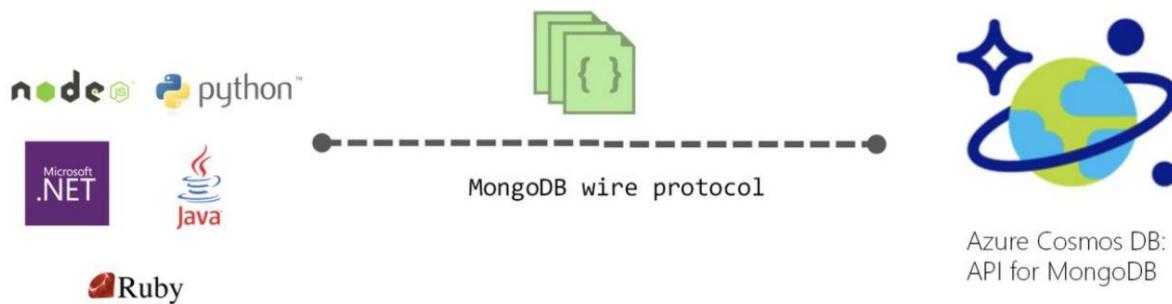


Figura 7-20. Uso de la API y el protocolo de MongoDB para acceder a Azure Cosmos DB

Este es un enfoque muy conveniente para la prueba de conceptos en entornos Docker con contenedores Linux porque la [imagen MongoDB Docker](#) es [una imagen de varias arquitecturas](#) que admite contenedores Docker Linux y contenedores Docker Windows.

Como se muestra en la siguiente imagen, mediante el uso de la API de MongoDB, eShopOnContainers admite contenedores de MongoDB Linux y Windows para el entorno de desarrollo local, pero luego puede pasar a una solución de nube PaaS escalable como Azure Cosmos DB simplemente [cambiando la cadena de conexión de MongoDB a apunte a Azure Cosmos DB](#).

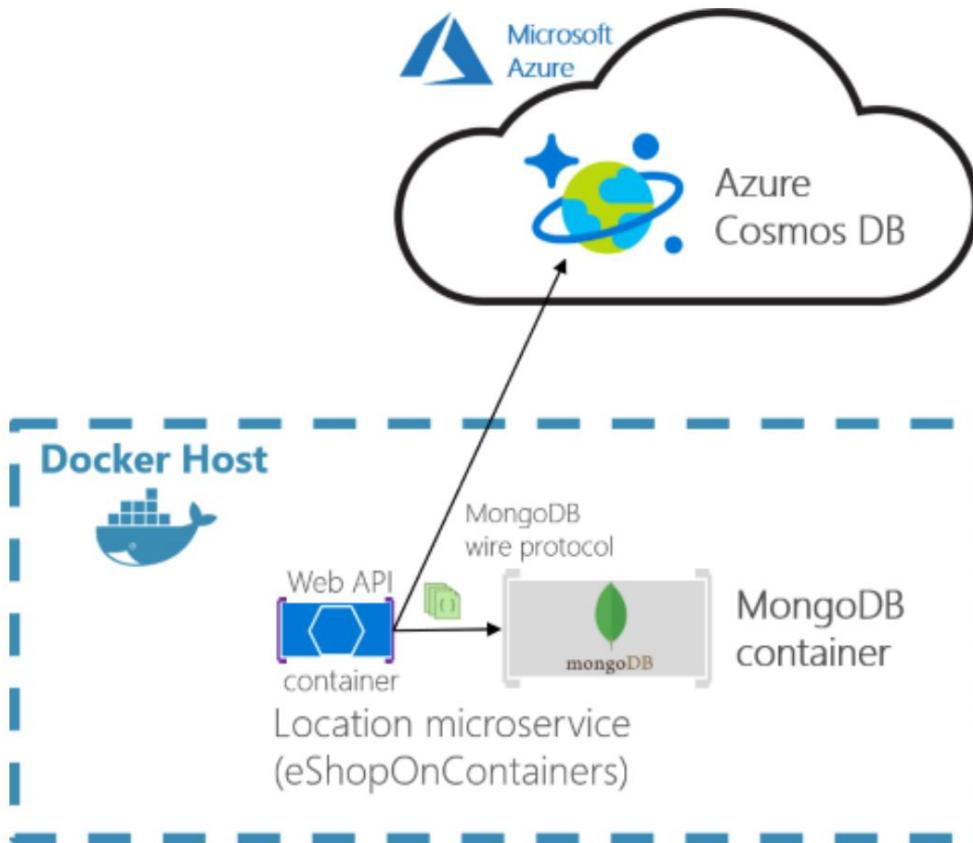


Figura 7-21. eShopOnContainers usando contenedores MongoDB para dev-env o Azure Cosmos DB para producción

La producción de Azure Cosmos DB se ejecutaría en la nube de Azure como PaaS y un servicio escalable.

Sus contenedores .NET personalizados pueden ejecutarse en un host de Docker de desarrollo local (que usa Docker para Windows en una máquina con Windows 10) o implementarse en un entorno de producción, como Kubernetes en Azure AKS o Azure Service Fabric. En este segundo entorno, implementaría solo los contenedores personalizados de .NET, pero no el contenedor de MongoDB, ya que usaría Azure Cosmos DB en la nube para manejar los datos en producción.

Un beneficio claro de usar 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 usar una API nativa (es decir, la API nativa de Cosmos DB) para aprovechar al máximo las capacidades de un motor de base de datos específico.

Para obtener una comparación más detallada entre el simple uso de MongoDB y Cosmos DB en la nube, consulte los [Beneficios de usar Azure Cosmos DB en esta página](#).

Analice su enfoque para aplicaciones de producción: MongoDB API vs. Cosmos DB API

En eShopOnContainers, usamos la API de MongoDB porque nuestra prioridad era fundamentalmente tener un entorno de desarrollo/prueba consistente usando una base de datos NoSQL que también podría 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, debe analizar las diferencias en las capacidades y el rendimiento al usar la API de MongoDB para acceder a las bases de datos de Azure Cosmos DB en comparación con el uso de la API nativa de Azure Cosmos DB. Si es similar, puede usar la API de MongoDB y obtiene el beneficio de admitir dos motores de base de datos NoSQL al mismo tiempo.

También podría usar los clústeres de MongoDB como la base de datos de producción en la nube de Azure, con [MongoDB Azure Service](#). Pero ese no es un servicio PaaS proporcionado por Microsoft. En este caso, Azure solo hospeda esa solución proveniente de MongoDB.

Básicamente, esto es solo un descargo de responsabilidad que indica que no siempre debe usar la API de MongoDB contra Azure Cosmos DB, como hicimos en eShopOnContainers porque era una opción conveniente para los contenedores de Linux. La decisión debe basarse en las necesidades y pruebas específicas que debe realizar para su aplicación de producción.

El código: usar la API de MongoDB en aplicaciones .NET

MongoDB API para .NET se basa en paquetes NuGet que debe agregar a sus proyectos, como en el proyecto Locations.API que se muestra en la siguiente figura.

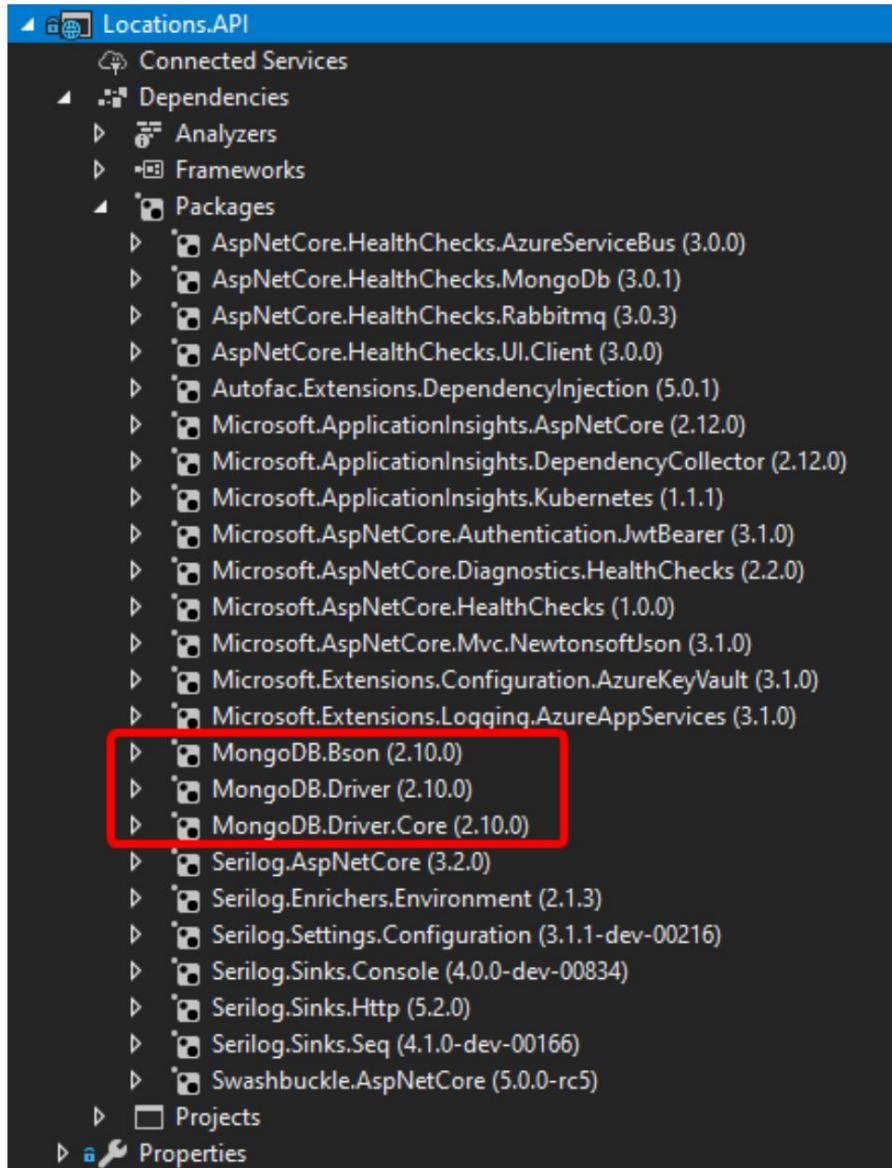


Figura 7-22. MongoDB API NuGet empaqueta referencias en un proyecto .NET

Investigaremos el código en las siguientes secciones.

[Un modelo utilizado por la API de MongoDB](#)

Primero, debe definir un modelo que contendrá los datos provenientes de la base de datos en el espacio de memoria de su aplicación. Este es un ejemplo del modelo utilizado para ubicaciones en eShopOnContainers.

```
utilizando MongoDB.Bson;
usando MongoDB.Bson.Serialization.Attributes; usando
MongoDB.Driver.GeoJsonObjectModel; usando
System.Collections.Generic;
```

```
Ubicaciones de clase pública {
```

```
[IdBson]
[BsonRepresentation(BsonType.ObjectId)] public
string Id { get; colocar; } public int LocationId { get;
colocar; } Código de cadena pública { obtener;
colocar; }
[BsonRepresentation(BsonType.ObjectId)] public
string Parent_Id { get; colocar; } public string
Descripción { get; colocar; } public double Latitud { get;
colocar; } doble longitud pública { obtener; colocar; }
Public GeoJsonPoint<GeoJson2DGeographicCoordinates>
Ubicación
{ obtener; conjunto privado; }
GeoJsonPolygon público <GeoJson2DGeographicCoordinates> Polígono
{ obtener; conjunto privado; }
public void SetLocation(doble lon, doble lat) => SetPosition(lon, lat); public void
SetArea(Lista<GeoJson2DGeographicCoordinates> coordenadasLista)
=> EstablecerPolígono(listacoordenadas);

Private void SetPosition(doble largo , doble latitud) {

    Latitud = lat;
    Longitud = longitud;
    Ubicación = new GeoJsonPoint<GeoJson2DGeographicCoordinates>( new
        GeoJson2DGeographicCoordinates(lon, lat));
}

privado void SetPolygon(Lista<GeoJson2DGeographicCoordinates> coordenadasLista) {

    Polígono = new GeoJsonPolygon<GeoJson2DGeographicCoordinates>(
        new GeoJsonPolygonCoordinates<GeoJson2DGeographicCoordinates>( new
            GeoJsonLinearRingCoordinates<GeoJson2DGeographicCoordinates>(coordinatesList)));
}

}
```

Puede ver que hay algunos atributos y tipos que provienen de los paquetes MongoDB NuGet.

Las bases de datos NoSQL suelen ser muy adecuadas para trabajar con datos jerárquicos no relacionales. En este ejemplo, usamos tipos de MongoDB creados especialmente para ubicaciones geográficas, 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 MongoCollections, como en el siguiente código.

```
contexto de ubicación de clase pública {

    IMongoDatabase privado de solo lectura _database = null;

    LocationsContext público (Configuración de IOptions<LocationSettings>) {

        var cliente = new MongoClient(settings.Value.ConnectionString); if (cliente != nulo)
            _database = cliente.GetDatabase(settings.Value.Database);

    }

    public IMongoCollection<Ubicaciones> Ubicaciones {
```

```

        conseguir
    {
        return _database.GetCollection<Ubicaciones>("Ubicaciones");
    }
}

```

Recuperar los datos

En el código C#, como los controladores de la API web o la implementación de repositorios personalizados, puede escribir un código similar al siguiente al realizar consultas a través de la API de MongoDB. Tenga en cuenta que el objeto `_context` es una instancia de la clase `LocationsContext` anterior.

```

tarea asincrónica pública <Ubicaciones> GetAsync(int locationId) {

    var filter = Constructores<Ubicaciones>.Filter.Eq("LocationId", locationId); regresar esperar
    _contexto.Ubicaciones .Buscar(filtrar)

        .FirstOrDefaultAsync();
}

```

Use un env-var en el archivo docker-compose.override.yml para la cadena de conexión MongoDB

Al crear un objeto MongoClient, necesita un parámetro fundamental que es precisamente el parámetro `ConnectionString` que apunta a la base de datos correcta. En el caso de eShopOnContainers, la cadena de conexión puede apuntar a un contenedor local de MongoDB Docker o a una base de datos de Azure Cosmos DB de "producción". Esa cadena de conexión proviene de las variables de entorno definidas en los archivos `docker compose.override.yml` que se usan al implementar con `docker-compose` o Visual Studio, como en el siguiente código yml.

```

# docker-compose.override.yml versión:
'3.4'
servicios:
    # Otras ubicaciones
    de servicios-api:
        entorno:
            # Otros ajustes
            - Cadena de conexión=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}

```

La variable de entorno `ConnectionString` se resuelve de esta manera: si la variable global `ESHOP_AZURE_COSMOSDB` está definida en el archivo `.env` con la cadena de conexión de Azure Cosmos DB, la usará para acceder a la base de datos de Azure Cosmos DB en la nube. Si no está definido, tomará el valor `mongodb://nosqldata` y utilizará el contenedor MongoDB de desarrollo.

El siguiente código muestra el archivo `.env` con la variable de entorno global de la cadena de conexión de Azure Cosmos DB, tal como se implementa en eShopOnContainers:

```

# Archivo .env, en la carpeta raíz de eShopOnContainers #
Otras variables de entorno de Docker

ESHOP_EXTERNAL_DNS_NAME_OR_IP=host.docker.internal
ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=<SuDockerHostIP>

#ESHOP_AZURE_COSMOSDB=<SuAzureCosmosDBConnData>

```

```
#Otras variables de entorno para recursos de infraestructura de Azure adicionales
#ESHOP_AZURE_REDIS_BASKET_DB=<Información de su cesta de AzureRedis>
#ESHOP_AZURE_STORAGE_CATALOG_URL=<SuAzureStorage_Catalog_BLOB_URL>
#ESHOP_AZURE_SERVICE_BUS=<TuAzureServiceBusInfo>
```

Quite el comentario de la línea ESHOP_AZURE_COSMOSDB y actualícela con su cadena de conexión de Azure Cosmos DB obtenida de Azure Portal como se explica en [Conexión de una aplicación MongoDB a Azure Cosmos DB](#).

Si la variable global ESHOP_AZURE_COSMOSDB está vacía, lo que significa que está comentada en el archivo .env , entonces el contenedor usa una cadena de conexión MongoDB predeterminada. Esta cadena de conexión apunta al contenedor MongoDB local implementado en eShopOnContainers que se llama nosqlData y se definió en el archivo docker-compose, como se muestra en el siguiente código .yml:

```
# docker-compose.yml versión:
'3.4' servicios: # ...Otras
servicios... nosqlData: imagen:
  mongo
```

Recursos adicionales

- Modelado de datos de documentos para bases de datos
[NoSQL](https://docs.microsoft.com/azure/cosmos-db/modeling-data) <https://docs.microsoft.com/azure/cosmos-db/modeling-data>
- Vaughn Vernon. ¿La tienda agregada de diseño impulsada por dominio ideal?
<https://kalele.io/blog-posts/the-ideal-domain-driven-design-aggregate-store/>
- Introducción a Azure Cosmos DB: API para MongoDB <https://docs.microsoft.com/azure/cosmos-db/mongodb-introduction>
- Azure Cosmos DB: cree una aplicación web de la API de MongoDB con .NET y Azure Portal <https://docs.microsoft.com/azure/cosmos-db/create-mongodb-dotnet>
- Utilice el emulador de Azure Cosmos DB para el desarrollo y las pruebas locales
<https://docs.microsoft.com/azure/cosmos-db/local-emulator>
- Conecte una aplicación MongoDB a Azure Cosmos DB <https://docs.microsoft.com/azure/cosmos-db/connect-mongodb-account>
- La imagen de Docker del emulador de Cosmos DB (contenedor de Windows)
<https://hub.docker.com/r/microsoft/azure-cosmosdb-emulator/>
- La imagen Docker de MongoDB (Linux y Windows Container)
https://hub.docker.com/_/mongo/
- Use MongoChef (Studio 3T) con Azure Cosmos DB: API para la cuenta de MongoDB <https://docs.microsoft.com/azure/cosmos-db/mongodb-mongochef>

Diseñar la capa de aplicación de microservicios y Web API

Use principios SOLID e Inyección de Dependencia

Los principios SOLID son técnicas críticas 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:

- Principio de responsabilidad única
- Principio abierto/cerrado
- Principio de sustitución de Liskov
- Principio de segregación de interfaz
- Principio de inversión de dependencia

SOLID trata más sobre cómo diseña su aplicación o capas internas de microservicios y sobre cómo desacoplar las 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, 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 dependencia (DI) es una forma de implementar el principio de inversión de dependencia. Es una técnica para lograr un acoplamiento débil entre los objetos y sus dependencias. En lugar de instanciar directamente a los colaboradores o usar referencias estáticas (es decir, usar nuevas...), los objetos que una clase necesita para realizar sus acciones se proporcionan a la clase (o se "inyectan en ella"). La mayoría de las veces, las clases declararán sus dependencias a través de su constructor, lo que les permitirá seguir el principio de dependencias explícitas. La inyección de dependencia generalmente se basa en contenedores de inversión de control (IoC) específicos. ASP.NET Core proporciona un contenedor IoC integrado simple, pero también puede usar su contenedor IoC favorito, como Autofac o Ninject.

Al seguir los principios de SOLID, sus clases tenderán naturalmente a ser pequeñas, bien factorizadas y fáciles de probar. Pero, ¿cómo puede saber si se inyectan demasiadas dependencias en sus clases? Si usa DI a través del constructor, será fácil detectarlo con solo mirar la cantidad de parámetros para su constructor. Si hay demasiadas dependencias, esto generalmente es una señal (un [olor a código](#)) de que su clase está tratando de hacer demasiado y probablemente está violando el principio de responsabilidad única.

Se necesitaría otra guía para cubrir SOLID en detalle. Por lo tanto, esta guía requiere que solo tenga un conocimiento mínimo de estos temas.

Recursos adicionales

- SOLID: Principios fundamentales de programación orientada a objetos <https://deviq.com/solid/>

- Inversión de Contenedores de Control y el patrón de Inyección de Dependencia <https://martinfowler.com/articles/injection.html>
- Steve Smith. Nuevo es pegamento
<https://ardalis.com/nuevo-es-pegamento>

Implementar la capa de aplicación de microservicios mediante la API web

Use Inyección de dependencia para injectar objetos de infraestructura en su capa de aplicación

Como se mencionó anteriormente, la capa de aplicación se puede implementar como parte del artefacto (ensamblaje) que está creando, 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 biblioteca de API web. Si desea separar lo que proviene de ASP.NET Core (su infraestructura más sus controladores) del código de su capa de aplicación personalizada, también puede colocar su capa de aplicación en una biblioteca 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.

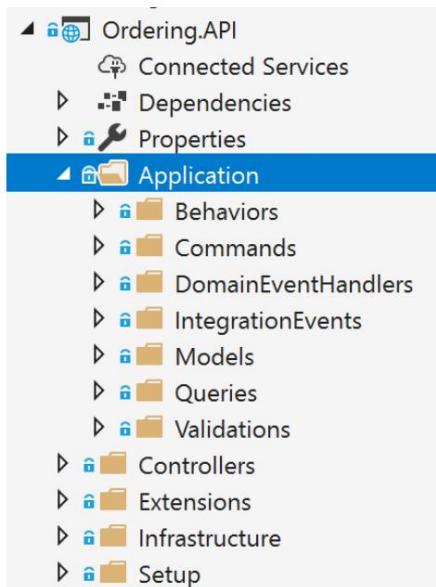


Figura 7-23. La capa de aplicación en el proyecto Ordering.API ASP.NET Core Web API

ASP.NET Core incluye un [contenedor IoC integrado simple](#) (representado por la interfaz `IServiceProvider`) que admite la inyección de constructor de forma predeterminada, y ASP.NET hace que ciertos servicios estén disponibles a través de DI. ASP.NET Core usa el término servicio para cualquiera de los tipos que registre que se inyectarán a través de DI. Configura los servicios del contenedor incorporado en el método `ConfigureServices` en su

Clase de inicio de la aplicación. Tus dependencias se implementan en los servicios que necesita un tipo y que registras en el contenedor IoC.

Por lo general, desea inyectar dependencias que implementen objetos de infraestructura. Una dependencia 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 EF DbContext), porque DbContext también es la implementación de sus objetos de persistencia de infraestructura.

En el siguiente ejemplo, puede ver cómo .NET inyecta los objetos de repositorio necesarios a través del constructor. La clase es un controlador de comandos, que se tratará en la siguiente sección.

```

clase pública CreateOrderCommandHandler
    : IRequestHandler<CreateOrderCommand, bool>
{
    privado de solo lectura IOrderRepository _orderRepository; privado de
    solo lectura IIdentityService _identityService; privado de solo lectura
    IMediator _mediator; privado de solo lectura
    IOrderingIntegrationEventService _orderingIntegrationEventService; privado de solo lectura
    ILogger<CreateOrderCommandHandler> _logger;

    // Usar DI para inyectar persistencia de infraestructura Repositorios public
    CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService servicio de identidad,
        registrador ILogger<CreateOrderCommandHandler>)
    {
        _orderRepository = orderRepository ?? tirar nuevo
        ArgumentNullException(nombre de(orderRepository));
        _identidadServicio = identidadServicio ?? tirar nuevo
        ArgumentNullException(nombre de(servicioidentidad));
        _mediador = mediator ?? lanzar una nueva excepción ArgumentNullException (nombre de (mediador));
        _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new
        ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = registrador ?? lanzar una nueva excepción ArgumentNullException (nombre de (registrador));
    }

    public async Task<bool> Handle(mensaje CreateOrderCommand, CancellationToken cancellationToken)
    {

        // Añadir evento de integración para limpiar la cesta var
        orderStartedIntegrationEvent = new
        OrderStartedIntegrationEvent(message.UserId); esperar

        _orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegrationEvent);

        // Agregar/Actualizar el Agregado Raíz del Comprador
        // Comentario de patrones DDD: Agregar entidades secundarias y objetos de valor a través de la Orden
        Raíz agregada
        // métodos y constructor para validaciones, invariantes y lógica de negocios // asegúrese de que se
        conserve la consistencia en todo el agregado var dirección = nueva dirección (mensaje.Calle,
        mensaje.Ciudad, mensaje.Estado, mensaje.País, mensaje.Código postal); var order = new
        Order(message.UserId, message.UserName, address, message.CardType, message.CardNumber,
        message.CardSecurityNumber, message.CardHolderName, message.CardExpiration);
    }
}

```

```

foreach ( elemento var en mensaje.OrderItems) {

    order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice, item.Discount,
item.PictureUrl, item.Units);
}

_logger.LogInformation("----- Creando pedido - Pedido: {@Order}", pedido);

_orderRepository.Add(pedido);

volver esperar _orderRepository.UnitOfWork
    .SaveEntitiesAsync(token de cancelación);
}
}

```

La clase usa los repositorios inyectados para ejecutar la transacción y persistir los cambios de estado. No importa si esa clase es un controlador de comandos, un método de controlador de ASP.NET Core Web API o un [servicio de aplicación DDD](#). En última instancia, es una clase simple que usa repositorios, entidades de dominio y otra coordinación de aplicaciones de manera similar a un controlador de comandos. La inyección de dependencia funciona de la misma manera para todas las clases mencionadas, como en el ejemplo que usa DI basado en el constructor.

Registre los tipos de implementación de dependencia y las interfaces o abstracciones

Antes de usar los objetos inyectados a través de constructores, necesita saber dónde registrar las interfaces y las clases que producen los objetos inyectados en las clases de su aplicación a través de DI. (Como DI basado en el constructor, como se muestra anteriormente).

Utilice el contenedor IoC integrado proporcionado por ASP.NET Core

Cuando utiliza el contenedor IoC integrado proporcionado por ASP.NET Core, registra los tipos que desea inyectar en el método ConfigureServices en el archivo Startup.cs, como en el siguiente código:

```

// Registro de tipos en el contenedor integrado de ASP.NET Core public void
ConfigureServices(IServiceCollection services) {

    // Registre los servicios de marco listos para usar.
    services.AddDbContext<CatalogContext>(c =>
        c.UseSqlServer(Configuration["ConnectionString"]),
        ServiceLifetime.Scoped);

    servicios.AddMvc(); //
    Registra dependencias de aplicaciones personalizadas.
    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 del contenedor IoC a través de cualquier constructor, solicita un objeto de cierto tipo de interfaz. Por ejemplo, en el ejemplo anterior, la última línea indica que cuando alguno de sus constructores tiene una dependencia de IMyCustomRepository (interfaz o abstracción), el contenedor IoC inyectará una instancia de la clase de implementación MyCustomSQLServerRepository.

Use la biblioteca Scrutor para el registro automático de tipos

Al usar DI en .NET, es posible que desee poder escanear un ensamblaje y registrar automáticamente sus tipos por convención.

Esta función no está disponible actualmente en ASP.NET Core. Sin embargo, puede usar la biblioteca [Scrutor](#) para eso. Este [enfoque](#) es conveniente cuando tiene docenas de tipos que deben registrarse en su contenedor IoC.

Recursos adicionales

- Mateo Rey. Registro de servicios con Scrutor
<https://www.mking.net/blog/registrar-servicios-con-scrutor>
- Kristian Hellang. escrutor. repositorio de GitHub.
<https://github.com/khellang/Scrutor>

Utilice Autofac como contenedor IoC

También puede usar contenedores IoC adicionales y conectarlos a la canalización de ASP.NET Core, como en el microservicio de pedidos en eShopOnContainers, que usa [Autofac](#). Cuando usa Autofac, [normalmente](#) registra los tipos a través de módulos, lo que le permite dividir los tipos de registro entre varios archivos según dónde estén sus tipos, de la misma manera que podría tener los tipos de aplicaciones distribuidos en varias bibliotecas de clases.

Por ejemplo, el siguiente es el módulo de la [aplicación Autofac](#) para el [proyecto Ordering.API Web API](#) con los tipos que querrá inyectar.

```
Módulo de aplicación de clase pública : Autofac.Module {
    public string QueriesConnectionString { get; } Módulo de
    aplicación pública (cadena qconstr) {

        QueriesConnectionString = qconstr;
    }

    anulación protegida carga vacía (ContainerBuilder builder) {

        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>()
            .InstancePerLifetimeScope();
        constructor.RegisterType<BuyerRepository>()
            .As<IBuyerRepository>()
            .InstancePerLifetimeScope();
        constructor.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();
        constructor.RegisterType<RequestManager>()
            .As<Administrador de Solicitudes>()
            .InstancePerLifetimeScope();
    }
}
```

Autofac también tiene una función para [escanear ensamblajes y registrar tipos por convenciones de nombres](#).

El proceso de registro y los conceptos son muy similares a la forma en que puede registrar tipos con el contenedor ASP.NET Core IoC integrado, pero la sintaxis cuando se usa 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 cada vez que un constructor declara 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 la instancia determina cómo se comparte una instancia entre las solicitudes del mismo servicio o dependencia. Cuando se realiza una solicitud para una dependencia, el contenedor IoC puede devolver lo siguiente:

- Una sola instancia por ámbito de por vida (mencionado en el contenedor ASP.NET Core IoC como ámbito).
- Una nueva instancia por dependencia (mencionado en el contenedor ASP.NET Core IoC como transitorio).
- Una única instancia compartida entre todos los objetos que usan el contenedor IoC (al que se hace referencia en ASP.NET Contenedor Core IoC como singleton).

Recursos adicionales

- Introducción a la inyección de dependencia en ASP.NET Core <https://docs.microsoft.com/aspnet/core/fundamentals/dependency-injection>
 - Autofac. Documentación oficial.
<https://docs.autofac.org/en/latest/>
 - Comparación de la vida útil del servicio de contenedor de ASP.NET Core IoC con los ámbitos de instancia de contenedor de Autofac IoC: César de la Torre. <https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-y-autofac-ioc-instance-scopes/>
-

Implementar los patrones de comando y controlador de comandos

En el ejemplo de DI a través del constructor que se muestra en la sección anterior, el contenedor IoC estaba inyectando repositorios a través de un constructor en una clase. Pero, ¿exactamente dónde se inyectaron? En una API web simple (por ejemplo, el microservicio de catálogo en `eShopOnContainers`), los inyecta en el nivel de los controladores MVC, en un constructor de controladores, como parte de la canalización de solicitudes 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 a través del constructor de un controlador de comandos en particular. Expliquemos qué es un controlador de comandos y por qué querría usarlo.

El patrón Command está intrínsecamente relacionado con el patrón CQRS que se presentó anteriormente en esta guía. CQRS tiene dos lados. La primera área son las consultas, utilizando consultas simplificadas con `Dapper` [micro ORM](#), que se explicó anteriormente. La segunda área son los comandos, que son el punto de partida para las transacciones y el canal de entrada desde fuera 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, conservar los estados con transacciones.

High level “Writes-side” in CQRS

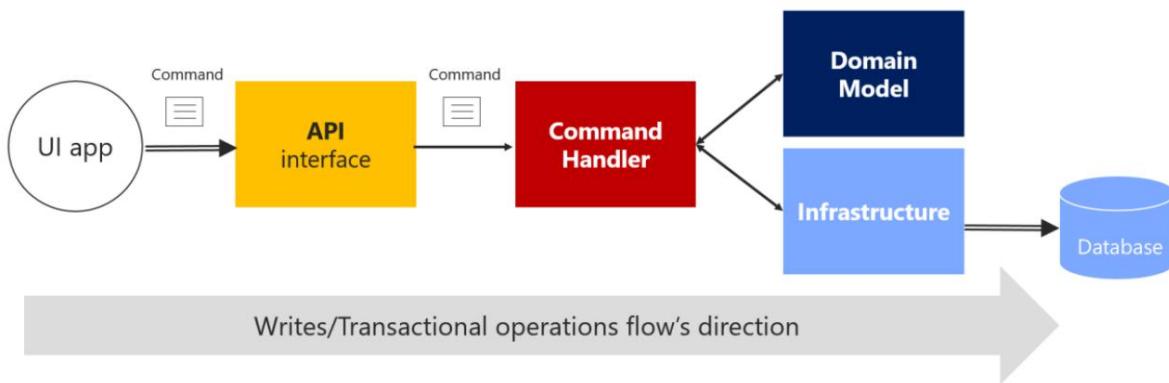


Figura 7-24. Vista de alto nivel de los comandos o “lado transaccional” en un patrón CQRS

La Figura 7-24 muestra que la aplicación de la interfaz de usuario envía un comando a través de la API que llega a un controlador de comandos, que depende del modelo de dominio y la infraestructura, para actualizar la base de datos.

La clase de comando

Un comando es una solicitud para que el sistema realice una acción que cambia el estado del sistema.

Los comandos son imperativos y deben procesarse solo una vez.

Dado que los comandos son imperativos, normalmente se nombran con un verbo en modo imperativo (por ejemplo, "crear" o "actualizar"), y pueden incluir el tipo agregado, como CreateOrderCommand. A diferencia de un evento, un mandato no es un hecho del pasado; es solo una solicitud y, por lo tanto, puede ser rechazada.

Los comandos pueden originarse en la interfaz de usuario como resultado de un usuario que inicia una solicitud, o desde un administrador de procesos cuando el administrador de procesos está dirigiendo un agregado para realizar una acción.

Una característica importante de un comando es que debe ser procesado una sola vez por un solo receptor.

Esto se debe a que un comando es una sola acción o transacción que desea realizar en la aplicación.

Por ejemplo, el mismo comando de creación de pedidos no debe procesarse más de una vez. Esta es una diferencia importante entre comandos y eventos. Los eventos se pueden procesar varias veces, porque muchos sistemas o microservicios pueden estar interesados en el evento.

Además, es importante que un comando se procese una sola vez en caso de que el comando no sea idempotente. 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 según las reglas comerciales e invariantes de su dominio. Por ejemplo, para usar el mismo ejemplo, si por alguna razón (lógica de reintento, piratería, etc.) el mismo comando CreateOrder llega a su sistema varias veces, debería poder identificarlo y asegurarse de no crear múltiples pedidos. Para hacerlo, debe adjuntar algún tipo de identidad en las operaciones e identificar si el comando o la actualización ya se procesaron.

Envías un comando a un solo receptor; no publicas un comando. La publicación es para eventos que declaran un hecho: que algo sucedió y podría ser interesante para los receptores de eventos. En el caso de los eventos, el editor no tiene preocupaciones sobre qué receptores reciben el evento o qué lo hacen.

Pero los eventos de dominio o 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 que se necesita para ejecutar ese comando. Un comando es un tipo especial de objeto de transferencia de datos (DTO), que se usa específicamente para solicitar cambios o transacciones. El comando en sí se basa exactamente en la información que se necesita para procesar el comando, y nada más.

El siguiente ejemplo muestra la clase `CreateOrderCommand` simplificada. Este es un comando inmutable que se usa en el microservicio de pedidos en eShopOnContainers.

```
// Comentario de patrones DDD y CQRS: tenga en cuenta que se recomienda implementar inmutable
Comandos
// En este caso, su inmutabilidad se logra al tener todos los setters como privados // además de poder actualizar
los datos una sola vez, al crear el objeto a través de su
constructor.
// Referencias sobre comandos inmutables: //
http://cqrssnu/Faq // https://docs.spine3.org/
motivation/immutability.html // http://blog.gauffin.org/2012/06 /griffin-container-
introducing-command-support/ // https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/
how-to-implement-a-lightweight-class-with- propiedades implementadas automáticamente
```

```
[Contrato de datos]
clase pública CreateOrderCommand
    : ISolicitud<bool>
{
    [Miembro de
    datos] lista privada de solo lectura <OrderItemDTO> _orderItems;

    [DataMember]
    public string UserId { get; conjunto privado; }

    [DataMember]
    public string Nombre de usuario { get; conjunto privado; }

    [DataMember]
    cadena pública Ciudad { get; conjunto privado; }

    [DataMember]
    public string Calle { get; conjunto privado; }

    [DataMember]
    cadena pública Estado { get; conjunto privado; }

    [DataMember]
    cadena pública País { get; conjunto privado; }

    [DataMember]
    public string ZipCode { get; conjunto privado; }

    [DataMember]
    public string CardNumber { get; conjunto privado; }

    [Miembro de datos]
```

```

public string NombreTitularDeLaTarjeta { get; conjunto privado; }

[DataMember]
public DateTime CardExpiration { get; conjunto privado; }

[DataMember]
public string CardSecurityNumber { get; conjunto privado; }

[DataMember]
public int CardTypeid { get; conjunto privado; }

[DataMember]
public IEnumerable<OrderItemDTO> OrderItems => _orderItems;

Public CreateOrderCommand() {

    _orderItems = new List<OrderItemDTO>();
}

public CreateOrderCommand(List<BasketItem> basketItems, string userId, string userName,
ciudad de cadena , calle de cadena , estado de cadena , país de cadena , código postal de cadena ,
número de tarjeta de cadena , nombre de titular de tarjeta de cadena, fecha y hora de caducidad
de tarjeta, número de seguridad de tarjeta de cadena , int cardTypeid ) : this()
{
    _orderItems = basketItems.ToOrderItemsDTO().ToList(); ID de usuario =
ID de usuario; nombre de usuario = nombre de usuario; Ciudad = ciudad;
calle = calle; Estado = estado; País = país; Código Postal = código postal;
NúmeroTarjeta = NúmeroTarjeta; Nombre del Titular de la Tarjeta =
Nombre del Titular de la Tarjeta; ExpiraciónTarjeta = ExpiraciónTarjeta;
NúmeroSeguridadTarjeta = NúmeroSeguridadTarjeta; IdTipoTarjeta =
IdTipoTarjeta; ExpiraciónTarjeta = ExpiraciónTarjeta;

}

orden de artículo de clase pública DTO
{
    public int ProductId { obtener; colocar; }

    public string ProductName { get; colocar; }

    PrecioUnitario decimal público { get; colocar; }

    Descuento decimal público { get; colocar; }

    public int Unidades { get; colocar; }

    cadena pública PictureUrl { obtener; colocar; }
}

```

Básicamente, la clase de comando contiene todos los datos que necesita para realizar una transacción comercial utilizando los objetos del modelo de dominio. Por lo tanto, los comandos son simplemente estructuras de datos que contienen solo lectura

datos y ningún comportamiento. El nombre del comando indica su propósito. En muchos lenguajes como C#, los comandos se representan como clases, pero no son verdaderas clases en el sentido real orientado a objetos.

Como característica adicional, los comandos son inmutables, ya que el uso esperado es que sean procesados directamente por el modelo de dominio. No necesitan cambiar durante su vida proyectada.

En una clase de C#, la inmutabilidad se puede lograr al no tener setters u otros métodos que cambien el estado interno.

Tenga en cuenta que si pretende o espera que los comandos pasen por un proceso de serialización o deserialización, las propiedades deben tener un setter privado y el atributo [DataMember] (o [JsonProperty]).

De lo contrario, el deserializador no podrá reconstruir el objeto en el destino con los valores requeridos. También puede usar propiedades verdaderamente de solo lectura si la clase tiene un constructor con parámetros para todas las propiedades, con la convención de nomenclatura camelCase habitual, y anotar el constructor como [JsonConstructor]. Sin embargo, esta opción requiere más código.

Por ejemplo, la clase de comando para crear un pedido probablemente sea similar en términos de datos al pedido que desea crear, pero probablemente no necesite los mismos atributos. Por ejemplo, CreateOrderCommand no tiene un ID de pedido porque el pedido aún no se ha creado.

Muchas clases de comandos pueden ser simples y solo requieren unos pocos campos sobre algún estado que debe cambiarse. Ese sería el caso si solo está cambiando el estado de un pedido de "en proceso" a "pagado" o "enviado" usando un comando similar al siguiente:

```
[DataContract]
clase pública UpdateOrderStatusCommand
    :ISolicitud<bool>
{
    [DataMember]
    cadena pública Estado { get; conjunto privado; }

    [DataMember]
    public string OrderId { get; conjunto privado; }

    [DataMember]
    public string BuyerIdentityGuid { get; conjunto privado; }
}
```

Algunos desarrolladores hacen que sus objetos de solicitud de interfaz de usuario se separen de sus DTO de comando, pero eso es solo una cuestión de preferencia. Es una separación tediosa sin mucho valor adicional, y los objetos tienen casi exactamente la misma forma. Por ejemplo, en eShopOnContainers, algunos comandos provienen directamente del lado del cliente.

[La clase de controlador de comandos](#)

Debe implementar una clase de controlador de comandos específica para cada comando. Así es como funciona el patrón, y es donde usará el objeto de comando, los objetos de dominio y los objetos de repositorio de infraestructura. El controlador de comandos es, de hecho, el corazón de la capa de aplicación en términos de CQRS y DDD. Sin embargo, toda la lógica del dominio debe estar contenida en las clases de dominio, dentro de las raíces agregadas (entidades raíz), entidades secundarias o [servicios de dominio](#), pero no dentro del controlador de comandos, que es una clase de la capa de aplicación.

La clase de controlador de comandos ofrece un sólido trampolín en el camino para lograr el principio de responsabilidad única (SRP) mencionado en una sección anterior.

Un controlador de comandos recibe un comando y obtiene un resultado del agregado que se utiliza. El resultado debe ser una ejecución exitosa del comando o una excepción. En el caso de una excepción, el estado del sistema debe permanecer sin cambios.

El controlador 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 es válido (si no es validado por el mediador).
- Instancia la instancia raíz agregada que es el destino del comando actual.
- Ejecuta el método en la instancia raíz agregada y obtiene los datos necesarios del comando.
- Persiste el nuevo estado del agregado en su base de datos relacionada. Esta última operación es la transacción real.

Por lo general, un controlador de comandos se ocupa de un solo agregado impulsado por su raíz agregada (entidad raíz).

Si varios agregados deben verse afectados por la recepción de un solo comando, puede 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 del dominio debe estar dentro del modelo de dominio (los agregados), completamente encapsulada y lista para la prueba unitaria. El controlador de comandos solo actúa como una forma de obtener el modelo de dominio de la base de datos y, como paso final, para decirle a la capa de infraestructura (repositorios) que persistan los cambios cuando se cambia el modelo.

La ventaja de este enfoque es que puede refactorizar la lógica del dominio en un modelo de dominio aislado, completamente encapsulado, Enriquecido y de comportamiento sin cambiar el código en las capas de aplicación o infraestructura, que son el nivel de plomería (controladores de comandos, API web, repositorios, etc.) .).

Cuando los controladores de comandos se vuelven complejos, con demasiada lógica, eso puede ser un olor a código. Revíselos y, si encuentra lógica de dominio, refactorice el código para mover ese comportamiento de dominio a los métodos de los objetos de dominio (la raíz agregada y la entidad secundaria).

Como ejemplo de una clase de controlador de comandos, el siguiente código muestra la misma clase `CreateOrderCommandHandler` que vio al principio de este capítulo. En este caso, también destaca el método `Handle` y las operaciones con los objetos/agregados del modelo de dominio.

```
clase pública CreateOrderCommandHandler
    : IRequestHandler<CreateOrderCommand, bool>
{
    privado de solo lectura IOrderRepository _orderRepository; privado de
    solo lectura IIdentityService _identityService; privado de solo lectura
    IMediator _mediator; privado de solo lectura
    IOrderingIntegrationEventService _orderingIntegrationEventService; privado de solo lectura
    ILogger<CreateOrderCommandHandler> _logger;

    // Usar DI para inyectar persistencia de infraestructura Repositorios public
    CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
```

```

    IIdentityService servicio de identidad,
    registrador ILogger<CreateOrderCommandHandler>
    {
        _orderRepository = orderRepository ?? tirar nuevo
        ArgumentNullException(nombrede(orderRepository));
        _identidadServicio = identidadServicio ?? tirar nuevo
        ArgumentNullException(nombrede(servicioidentidad));
        _mediador = mediador?? lanzar una nueva excepción ArgumentNullException (nombre de (mediador));
        _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new
        ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = registrador ?? lanzar una nueva excepción ArgumentNullException (nombre de (registrador));
    }

    public async Task<bool> Handle(mensaje CreateOrderCommand, CancellationToken cancellationToken)
    {

        // Añadir evento de integración para limpiar la cesta var
        orderStartedIntegrationEvent = new
        OrderStartedIntegrationEvent(message.UserId); esperar

        _orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegrationEvent);

        // Agregar/Actualizar el Agregado Raíz del Comprador
        // Comentario de patrones DDD: Agregar entidades secundarias y objetos de valor a través de la Orden
        Aggregate-Root //
            métodos y constructor para validaciones, invariantes y lógica de negocios // asegúrese de que se
            conserve la consistencia en todo el agregado var address = new Address(message.Street,
            message.City, message.State, message.Country, message.Código postal); var order = new
            Order(message.UserId, message.UserName, address, message.CardTypeId, message.CardNumber,
            message.CardSecurityNumber, message.CardHolderName, message.CardExpiration);

        foreach ( elemento var en mensaje.OrderItems) {

            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
            artículo.Descuento, artículo.PictureUrl, artículo.Unidades);
        }

        _logger.LogInformation("----- Creando pedido - Pedido: {@Order}", pedido);

        _orderRepository.Add(pedido);

        volver esperar _orderRepository.UnitOfWork
            .SaveEntitiesAsync(token de cancelación);
    }
}

```

Estos son pasos adicionales que debe seguir un controlador de comandos:

- Utilice los datos del comando para operar con los métodos y el comportamiento de la raíz agregada.
- Internamente dentro de los objetos de dominio, genera eventos de dominio mientras se ejecuta la transacción, pero eso es transparente desde el punto de vista del controlador de comandos.
- Si el resultado de la operación del agregado es exitoso y una vez finalizada la transacción, genere eventos de integración. (Estos también pueden ser generados por clases de infraestructura como repositorios).

Recursos adicionales

- Mark Seemann. En los límites, las aplicaciones no están orientadas a objetos

[https://blog.ploeh.dk/2011/05/31/En los límites, las aplicaciones no están orientadas a objetos/](https://blog.ploeh.dk/2011/05/31/En%20los%20límites,%20las%20aplicaciones%20no%20están%20orientadas%20a%20objetos/)

- Comandos y eventos

<https://cqrs.nu/Faq/comandos-y-eventos/>

- ¿Qué hace un controlador de comandos?

<https://cqrs.nu/Faq/controladores-de-comandos/>

- Jimmy Bogard. Patrones de comando de dominio: controladores

<https://jimmybogard.com/domain-command-patterns-handlers/>

- Jimmy Bogard. Patrones de comando de dominio: validación

<https://jimmybogard.com/domain-command-patterns-validation/>

La canalización del proceso de comandos: cómo activar un controlador de comandos

La siguiente pregunta es cómo invocar un controlador de comandos. Puede llamarlo manualmente desde cada controlador ASP.NET Core relacionado. Sin embargo, ese enfoque sería demasiado acoplado y no es ideal.

Las otras dos opciones principales, que son las opciones recomendadas, son:

- A través de un artefacto de patrón Mediator en memoria.
- Con una cola de mensajes asíncrona, entre controladores y manejadores.

Usar el patrón Mediator (en memoria) en la canalización de comandos

Como se muestra en la Figura 7-25, en un enfoque CQRS se utiliza un mediador inteligente, similar a un bus en memoria, que es lo suficientemente inteligente como para redirigir al controlador de comandos correcto según el tipo de comando o DTO que se recibe. Las flechas negras individuales entre los componentes representan las dependencias entre los objetos (en muchos casos, inyectados a través de DI) con sus interacciones relacionadas.

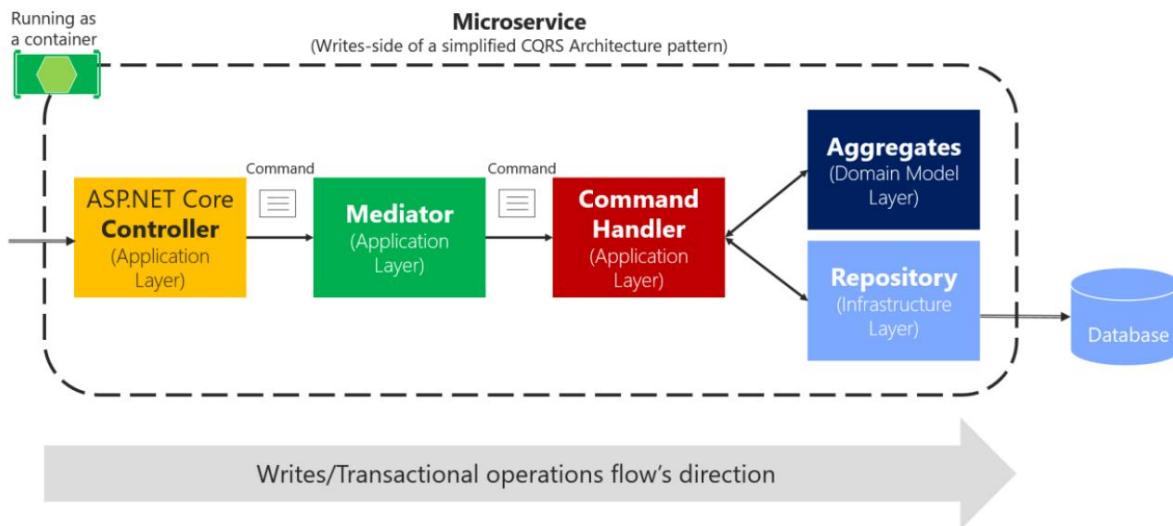


Figura 7-25. Uso del patrón Mediator en proceso en un solo microservicio CQRS

El diagrama anterior muestra un acercamiento de la imagen 7-24: el controlador ASP.NET Core envía el comando a la canalización de comandos de MediatR, para que lleguen al controlador adecuado.

La razón por la que tiene sentido usar el patrón Mediator es que en las aplicaciones empresariales, las solicitudes de procesamiento pueden complicarse. Desea poder agregar un número abierto de preocupaciones transversales como registro, validaciones, auditoría y seguridad. En estos casos, puede confiar en una canalización de mediador (consulte [Patrón de mediador](#)) para proporcionar un medio para estos comportamientos adicionales o preocupaciones transversales.

Un mediador es un objeto que encapsula el "cómo" de este proceso: coordina la ejecución en función del estado, la forma en que se invoca un controlador de comandos o la carga útil que proporciona al controlador. Con un componente de mediador, puede aplicar preocupaciones transversales de forma centralizada y transparente mediante la aplicación de decoradores (o [comportamientos de canalización](#) desde [MediatR 3](#)). Para obtener más información, consulte el [patrón Decorador](#).

Los decoradores y los comportamientos son similares a la [programación orientada a aspectos \(AOP\)](#), solo que se aplican a una tubería de proceso específica administrada por el componente mediador. Los aspectos en AOP que implementan preocupaciones transversales se aplican en función de los tejedores de aspectos inyectados en el momento de la compilación o en función de la intercepción de llamadas de objetos. A veces se dice que ambos enfoques típicos de AOP funcionan "como magia", porque no es fácil ver cómo funciona AOP. Cuando se trata de problemas o errores graves, AOP puede ser difícil de depurar. Por otro lado, estos decoradores/comportamientos son explícitos y se aplican solo en el contexto del mediador, por lo que la depuración es mucho más predecible y sencilla.

Por ejemplo, en el microservicio de pedidos eShopOnContainers, tiene una implementación de dos comportamientos de muestra, 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](#).

Usar colas de mensajes (fuera de proceso) en la canalización del comando

Otra opción es usar mensajes asincrónicos basados en intermediarios o colas de mensajes, como se muestra en la figura 7-26. Esa opción también podría combinarse con el componente mediador justo antes del controlador de comandos.

Writes-side of a CQRS Architecture pattern using messaging

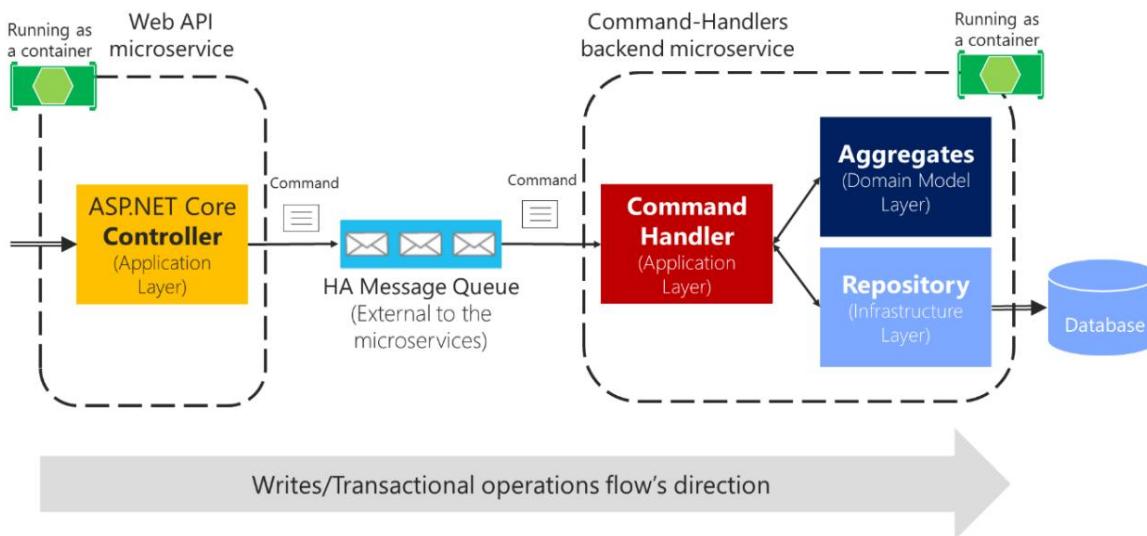


Figura 7-26. Uso de colas de mensajes (fuera del proceso y comunicación entre procesos) con comandos CQRS

La canalización del comando también puede ser manejada por una cola de mensajes de alta disponibilidad para entregar los comandos al controlador adecuado. El uso de colas de mensajes para aceptar los comandos puede complicar aún más la canalización de su comando, porque probablemente necesitará dividir la canalización en dos procesos conectados a través de la cola de mensajes externa. Aún así, debe usarse si necesita mejorar la escalabilidad y el rendimiento en función de la mensajería asíncrona. Considere que en el caso de la Figura 7-26, el controlador simplemente publica el mensaje de comando en la cola y regresa. Luego, los controladores de comandos procesan los mensajes a su propio ritmo. Ese es un gran beneficio de las colas: la cola de mensajes puede actuar como un búfer en los casos en que se necesita hiperescalabilidad, como para acciones o cualquier otro escenario con un alto volumen de datos de entrada.

Sin embargo, debido a la naturaleza asincrónica de las colas de mensajes, debe averiguar cómo comunicarse con la aplicación cliente sobre el éxito o el fracaso del proceso del comando. Como regla general, nunca debe usar los comandos "disparar y olvidar". Toda aplicación comercial necesita saber si un comando se procesó correctamente o, al menos, se validó o aceptó.

Por lo tanto, poder responder al cliente después de validar un mensaje de comando que se envió a una cola asincrónica agrega complejidad a su sistema, en comparación con un proceso de comando en proceso que devuelve el resultado de la operación después de ejecutar la transacción. Al usar colas, es posible que deba devolver el resultado del proceso de comando a través de otros mensajes de resultados de operaciones, lo que requerirá componentes adicionales y comunicación personalizada en su sistema.

Además, los comandos asíncronos son comandos unidireccionales, que en muchos casos pueden no ser necesarios, como se explica en el siguiente intercambio interesante entre Burtsel Alexey y Greg Young en una [conversación en línea](#):

[Burtsel Alexey] Encuentro mucho código en el que las personas usan el manejo de comandos asíncronos o la mensajería de comandos unidireccional sin ninguna razón para hacerlo (no están haciendo una operación larga, no están ejecutando código asíncrono externo, ni siquiera cruzan). límite de la aplicación para utilizar el bus de mensajes). ¿Por qué introducen esta complejidad innecesaria? Y, de hecho, hasta ahora no he visto un ejemplo de código CQRS con controladores de comandos de bloqueo, aunque funcionará 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 plantear un evento si no estoy de acuerdo, ya no es que me digas que haga algo [es decir, no es un comando]. Eres tú diciéndome que se ha hecho algo. Esto parece una ligera diferencia al principio, pero tiene muchas implicaciones.

Los comandos asíncronos aumentan considerablemente la complejidad de un sistema, porque no existe una forma sencilla de indicar fallas. Por lo tanto, los comandos asíncronos no se recomiendan excepto cuando se necesitan requisitos de escalado o en casos especiales cuando se comunican los microservicios internos a través de mensajes. En esos casos, debe diseñar un sistema de informes y recuperación separado para fallas.

En la versión inicial de eShopOnContainers, se decidió utilizar el procesamiento de comandos síncrono, iniciado a partir de solicitudes HTTP e impulsado por el patrón Mediator. Eso le permite devolver fácilmente 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 comerciales de su aplicación o microservicio.

[Implementar la canalización del proceso de comando con un patrón de mediador \(MediatR\)](#)

Como ejemplo de implementación, esta guía propone el uso de la canalización en proceso basada en el patrón Mediator para impulsar la ingestión de comandos y enrutar los comandos, en la memoria, a los controladores de comandos correctos. La guía también propone aplicar [comportamientos](#) para separar las preocupaciones transversales.

Para la implementación en .NET, existen varias bibliotecas de código abierto disponibles que implementan el patrón Mediator. La biblioteca utilizada en esta guía es la biblioteca de código abierto [MediatR](#) (creada por Jimmy Bogard), pero podría usar otro enfoque. MediatR es una biblioteca pequeña y simple que le permite procesar mensajes en memoria como un comando, mientras aplica decoradores o comportamientos.

El uso del patrón Mediator lo ayuda a reducir el acoplamiento y a aislar las preocupaciones del trabajo solicitado, mientras se conecta automáticamente al controlador que realiza ese trabajo, en este caso, a los controladores de comandos.

Jimmy Bogard explicó otra buena razón para usar el patrón Mediator al revisar esta guía:

Creo que podría valer la pena mencionar aquí las pruebas: proporciona una buena ventana consistente al comportamiento de su sistema. Solicitud de entrada, respuesta de salida. Hemos encontrado ese aspecto bastante valioso en la construcción de pruebas que se comportan consistentemente.

Primero, veamos un controlador WebAPI de muestra en el que realmente usaría el objeto mediador. Si no estuviera utilizando el objeto mediador, necesitaría inyectar todas las dependencias para ese controlador, cosas como un objeto registrador y otros. Por lo tanto, el constructor sería complicado. Por otro lado, si usa el objeto mediador, el constructor de su controlador puede ser mucho más simple, con solo unas pocas dependencias en lugar de muchas dependencias si tuviera una por operación transversal, como en el siguiente ejemplo:

```
clase pública MyMicroserviceController: Controlador {

    public MyMicroserviceController(IMediator mediador,
                                    IMyMicroserviceQueries microserviceQueries)
    {
        // ...
    }
}
```

Puede ver que el mediador proporciona un constructor de controlador de API web limpio y eficiente. Además, dentro de los métodos del controlador, el código para enviar un comando al objeto mediador es casi una línea:

```
[Ruta("nueva")]
[HttpPost]
tarea asíncrona pública <ActionResult> ExecuteBusinessOperation([FromBody]RunOpCommand
                                                                ejecutar comando de operación)
{
    var commandResult = await _mediator.SendAsync(runOperationCommand);

    devolver comandoResultado? (ActionResult)Ok() : (ActionResult)BadRequest();
}
```

Implementar comandos idempotentes

En eShopOnContainers, un ejemplo más avanzado que el anterior es enviar un objeto CreateOrderCommand desde el microservicio de pedidos. Pero dado que el proceso comercial de pedidos es un poco más complejo y, en nuestro caso, en realidad comienza en el microservicio Basket, esta acción de enviar el objeto CreateOrderCommand se realiza desde un controlador de eventos de integración llamado [UserCheckoutAcceptedIntegrationEventHandler](#) en lugar de un controlador WebAPI simple [llamado desde la aplicación cliente como en el ejemplo anterior más simple](#).

No obstante, 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);
resultado = esperar _mediador.Enviar(solicitarCrearPedido);
```

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 si el mismo mensaje llega duplicado a través de la red, por cualquier motivo, como reintentos, la misma orden comercial se procesará solo una vez.

Esto se implementa envolviendo el comando comercial (en este caso, CreateOrderCommand) e incrustándolo en un IdentifiedCommand genérico, que se rastrea mediante una identificación de cada mensaje que llega a través de la red que tiene que ser idempotente.

En el código a continuación, puede ver que IdentifiedCommand no es más que un DTO con un ID más el objeto de comando comercial envuelto.

```
clase pública IdentifiedCommand<T, R> : IRequest<R> donde T :
    IRequest<R>
{
    Comando público T { obtener; } ID
    de Guid público { get; } comando
    identificado público ( comando T, ID de Guid) {

        comando = comando;
        identificación = identificación;
    }
}
```

Luego, CommandHandler para IdentifiedCommand llamado [IdentifiedCommandHandler.cs](#) básicamente comprobará si el ID que viene como parte del mensaje ya existe en una tabla. Si ya existe, ese comando no se volverá a procesar, por lo que se comporta como un comando idempotente. Ese código de infraestructura lo realiza la siguiente llamada al método `_requestManager.ExistAsync`.

```
// IdentifiedCommandHandler.cs public
class IdentifiedCommandHandler<T, R> : IRequestHandler<IdentifiedCommand<T, R>, R>
    donde T : IRequest<R>
{
    privado de solo lectura IMediator _mediador; privado
    de solo lectura IRequestManager _requestManager; privado de solo
    lectura ILogger<IdentifiedCommandHandler<T, R>> _logger;

    public IdentifiedCommandHandler(
        mediador mediador,
        Administrador de solicitudes IRequestManager,
        ILogger<IdentifiedCommandHandler<T, R>> registrador)
    {
        _mediador = mediador;
        _requestManager = requestManager; _logger
        = registrador ?? lanzar una nueva System.ArgumentNullException(nameof(logger));
    }

    /// <resumen>
    /// Crea el valor de resultado para devolver si se encontró una solicitud anterior /// </summary> ///
    <returns></returns>
```

```

    protegido virtual R CreateResultForDuplicateRequest() {
        devolver por defecto (R);
    }

    /// <summary> ///
    Este método maneja el comando. Simplemente asegura que no exista ninguna otra solicitud con
    el mismo ID, y si este es el caso /// simplemente
    pone en cola el comando interno original. /// </summary> ///
    <param name="message">IdentifiedCommand que contiene
    tanto el comando original como el ID de solicitud</param> /// <returns> Valor devuelto del comando interno o valor
    predeterminado si el mismo ID de solicitud fue encontrado</returns> public async Task<R> Handle(IdentifiedCommand<T,
    R> mensaje, CancellationToken cancellationToken) {

        var ya existe = espera _requestManager.ExistAsync(mensaje.Id); si (ya existe) {

            devuelve CreateResultForDuplicateRequest();

        } más
        {
            esperar _requestManager.CreateRequestForCommandAsync<T>(mensaje.Id); prueba {

                var comando = mensaje.Comando; var
                nombreComando = comando.GetGenericTypeName(); var
                idProperty = cadena.Vacio; var commandId = cadena.Vacio;

                cambiar (comando) {

                    case CreateOrderCommand createOrderCommand:
                        idProperty = nombrede(createOrderCommand.UserId);
                        commandId = createOrderCommand.UserId; descanso;

                    case CancelOrderCommand cancelOrderCommand:
                        idProperty = nameof(cancelOrderCommand.OrderNumber); commandId
                        = $"{cancelOrderCommand.OrderNumber}"; descanso;

                    case ShipOrderCommand shipOrderCommand:
                        idProperty = nameof(shipOrderCommand.OrderNumber); commandId
                        = $"{shipOrderCommand.OrderNumber}"; descanso;

                    por defecto:
                        idPropiedad = "¿Id?";
                        comandold = "n/a";
                        descanso;
                }
            }

            _logger.Información de registro(
                "----- Enviando comando: {CommandName} - {IdProperty}: {CommandId}"
                ((@Dominio)),
                nombrecomando,
                idProperty,
                commandId,
            );
        }
    }
}

```

```
dominio);

// Envía el comando comercial incorporado al mediador para que ejecute su comando relacionado

Manejador de comandos

    var result = await _mediator.Send(comando, cancelacionToken);

    _logger.Información de registro(
        "----- Resultado del comando: {@Result} - {CommandName} - {IdProperty}:"
{CommandId} {@Command}",
        resultado,
        commandName,
        idProperty,
        commandId,
        comando);

    resultado devuelto ;

} atrapar
{
    devolver por defecto (R);
}
}
```

Dado que IdentifiedCommand actúa como el sobre de un comando comercial, cuando el comando comercial debe procesarse porque no es una ID repetida, toma ese comando comercial interno y lo vuelve a enviar a Mediator, como en la última parte del código que se muestra arriba cuando ejecutando `_mediator.Send(message.Command)`, desde [IdentifiedCommandHandler.cs](#).

Al hacerlo, vinculará y ejecutará el controlador de comandos comerciales, en este caso, [CreateOrderCommandHandler](#), que ejecuta transacciones en la base de datos de pedidos, como se muestra en el siguiente código.

```
// CreateOrderCommandHandler.cs clase
pública CreateOrderCommandHandler :
    IRequestHandler<CreateOrderCommand, bool>
{
    privado de solo lectura IOrderRepository _orderRepository; privado de
    solo lectura IIdentityService _identityService; privado de solo lectura
    IMediator _mediator; privado de solo lectura IOrderingIntegrationEventService
    _orderingIntegrationEventService; privado de solo lectura ILogger<CreateOrderCommandHandler> _logger;

// Usar DI para injectar persistencia de infraestructura Repositorios public
CreateOrderCommandHandler(IMediator mediator,
    IOrderingIntegrationEventService orderingIntegrationEventService,
    IOrderRepository orderRepository,
    IIdentityService servicio de identidad,
    registrador ILogger<CreateOrderCommandHandler>)

{
    _orderRepository = orderRepository ?? tirar nuevo
    ArgumentNullException(nombre de(orderRepository));
    _identidadServicio = identidadServicio ?? tirar nuevo
    ArgumentNullException(nombre de(servicioidentidad));
    _mediador = mediador?? lanzar una nueva excepción ArgumentNullException (nombre de (mediador))
    _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new
    ArgumentNullException(nameof(orderingIntegrationEventService));
    logger = registrador ?? lanzar una nueva excepción ArgumentNullException (nombre de (registrador));
}
```

```

    }

    public async Task<bool> Handle(mensaje CreateOrderCommand, CancellationToken cancellationToken)
    {

        // Añadir evento de integración para limpiar la cesta var
        orderStartedIntegrationEvent = new
        OrderStartedIntegrationEvent(message.UserId); esperar

        _orderingIntegrationService.AddAndSaveEventAsync(orderStartedIntegrationEvent);

        // Agregar/Actualizar el Agregado Raíz del Comprador
        // Comentario de patrones DDD: Agregar entidades secundarias y objetos de valor a través de la Orden
        Aggregate-Root //
            métodos y constructor para validaciones, invariantes y lógica de negocios // asegúrese de que se
            conserve la consistencia en todo el agregado var address = new Address(message.Street,
            message.City, message.State, message.Country, message.Código postal); var order = new
        Order(message.UserId, message.UserName, address, message.CardType, message.CardNumber,
            message.CardSecurityNumber, message.CardHolderName, message.CardExpiration);

        foreach ( elemento var en mensaje.OrderItems) {

            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice, item.Discount,
            item.PictureUrl, item.Units);
        }

        _logger.LogInformation("----- Creando pedido - Pedido: {@Order}", pedido);

        _orderRepository.Add(pedido);

        volver esperar _orderRepository.UnitOfWork
            .SaveEntitiesAsync(token de cancelación);
    }
}

```

Registrar los tipos utilizados por MediatR

Para que MediatR conozca sus clases de controlador de comandos, debe registrar las clases de mediador y las clases de controlador de comandos en su contenedor IoC. De forma predeterminada, MediatR usa Autofac como contenedor de IoC, pero también puede usar el contenedor ASP.NET Core IoC integrado o cualquier otro contenedor compatible con MediatR.

El siguiente código muestra cómo registrar los tipos y comandos de Mediator al usar módulos de Autofac.

```

Clase pública MediatorModule: Autofac.Module {

    anulación protegida carga vacía (ContainerBuilder builder) {

        constructor.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .ComInterfacesImplementadas();

        // Registre todas las clases de comandos (implementan IRequestHandler) // en ensamblado
        que contiene el generador de comandos.RegisterAssemblyTypes
        (typeof(CreateOrderCommand).GetTypeInfo().Assembly)
            .AsClosedTypesOf(typeof(IRequestHandler<,>));
    }
}

```

```
// Registro de otros tipos //...
}
```

Aquí es donde “sucede la magia” con MediatR.

Como cada controlador de comandos implementa la interfaz genérica `IRequestHandler<T>`, cuando registra los ensamblados mediante el método `RegisteredAssemblyTypes`, todos los tipos marcados como `IRequestHandler` también se registran con sus comandos. Por ejemplo:

```
clase pública CreateOrderCommandHandler
    : IRequestHandler<CreateOrderCommand, bool>
{
```

Ese es el código que correlaciona los comandos con los controladores de comandos. El controlador es solo una clase simple, pero hereda de `RequestHandler<T>`, donde `T` es el tipo de comando y MediatR se asegura de que se invoque con la carga correcta (el comando).

Aplicar preocupaciones transversales al procesar comandos con el Comportamientos en MediatR

Hay una cosa más: poder aplicar preocupaciones transversales a la canalización del mediador. También puede ver al final del código del módulo de registro de Autofac cómo registra un tipo de comportamiento, específicamente, una clase `LoggingBehavior` personalizada y una clase `ValidatorBehavior`. Pero también podría agregar otros comportamientos personalizados.

```
Clase pública MediatorModule: Autofac.Module {
    anulación protegida carga vacía (ContainerBuilder builder) {
        constructor.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .ComInterfacesImplementadas();

        // Registre todas las clases de comandos (implementan IRequestHandler) // en
        // ensamblado que contiene el generador de comandos.RegisterAssemblyTypes
        ( typeof(CreateOrderCommand).GetTypeInfo().Assembly)

        AsClosedTypesOf(typeof(IRequestHandler<,>)); // 
        Registro de otros tipos //... builder.RegisterGeneric(typeof(LoggingBehavior<,>));

        builder.RegisterGeneric(typeof(ValidatorBehavior<,>));
            Como(tipode(IPipelineBehavior<,>));
            Como(tipode(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 tuvo éxito o no.

```
clase pública LoggingBehavior<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
{
    privado de solo lectura ILogger<LoggingBehavior<TRequest, TResponse>> _logger; Public
    LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> registrador) =>
```

```

    _registrar = registrar;

Asincrónica pública Task<TResponse> Handle(TRequest request,
                                              RequestHandlerDelegate<TResponse> siguiente)
{
    _logger.LogInformation($"Manejo {typeof(TRequest).Name}"); var respuesta =
        esperar siguiente(); _logger.LogInformation($"Manejado
{typeof(TResponse).Name}"); respuesta de retorno ;
}

}

```

Simplemente implementando esta clase de comportamiento y registrándola en la canalización (en el MediatorModule anterior), todos los comandos procesados a través de MediatR registrarán información sobre la ejecución.

El microservicio de pedidos eShopOnContainers también aplica un segundo comportamiento para las validaciones básicas, la clase [ValidatorBehavior](#) que se basa en la [biblioteca FluentValidation](#), como se muestra en el siguiente código:

```

clase pública ValidatorBehavior<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
{
    privado de solo lectura IValidator<TRequest>[] _validadores; public
    ValidatorBehavior(IValidator<TRequest>[] validadores) =>
        _validadores = validadores;

    Asincrónica pública Task<TResponse> Handle(TRequest request,
                                              RequestHandlerDelegate<TResponse> siguiente)
    {
        var fallas = _validadores .Select (v
            => v.Validate(solicitud))
            .SelectMany(resultado => resultado.errores)
            .Dónde(error => error != nulo)
            .Listar();

        si (falla.Cualquiera()) {

            lanzar una nueva OrderingDomainException
                ( $"Errores de validación de comando para el tipo {typeof(TRequest).Name}",
                    new ValidationException(" Excepción de validación", fallas));
        }

        var respuesta = esperar siguiente();
        respuesta de retorno ;
    }
}

```

Aquí, el comportamiento genera una excepción si falla la validación, pero también puede devolver un objeto de resultado, que contenga el resultado del comando si tuvo éxito o los mensajes de validación en caso de que no. Esto probablemente facilitaría la visualización de los resultados de la validación al usuario.

Luego, según la [biblioteca FluentValidation](#), crearía una validación para los datos pasados con CreateOrderCommand, como en el siguiente código:

```

clase pública CreateOrderCommandValidator : AbstractValidator<CreateOrderCommand> {

    Public CreateOrderCommandValidator() {

```

```

        RuleFor(comando => comando.Ciudad).NotEmpty();
        RuleFor(comando => comando.Calle).NotEmpty();
        RuleFor(comando => comando.Estado).NotEmpty();
        RuleFor(comando => comando.País).NotEmpty();
        RuleFor(comando => comando.ZipCode).NotEmpty();
        RuleFor(comando => comando.CardNumber).NotEmpty().Length(12, 19); RuleFor(comando
=> comando.CardHolderName).NotEmpty(); RuleFor(command =>
command.CardExpiration).NotEmpty().Must(BeValidExpirationDate).WithMessage("Por
favor , especifique una fecha de caducidad de tarjeta válida"); RuleFor(comando =>
comando.CardSecurityNumber).NotEmpty().Length(3); RuleFor(comando => comando.CardTypeId).NotEmpty();
        RuleFor(command => command.OrderItems).Must(ContainOrderItems).WithMessage("No

artículos de pedido encontrados"); }

    bool privado BeValidExpirationDate(DateTime dateTime) {

        volver fechaHora >= FechaHora.UtcNow;
    }

    bool privado ContainOrderItems(IEnumerable<OrderItemDTO> orderItems) {

        volver orderItems.Any();
    }
}

```

Podría crear validaciones adicionales. Esta es una forma muy limpia y elegante de implementar sus validaciones de comandos.

De manera similar, podría implementar otros comportamientos para aspectos adicionales o preocupaciones transversales que deseé aplicar a los comandos cuando los manipule.

Recursos adicionales

El patrón del mediador

- Patrón de mediador
https://en.wikipedia.org/wiki/Mediator_pattern

El patrón del decorador

- Patrón de decorador
https://en.wikipedia.org/wiki/Decorator_pattern

MediatR (Jimmy Bogard)

- MediatR. repositorio de
[GitHub. https://github.com/bogard/mediatr](https://github.com/bogard/mediatr)
- CQRS con MediatR y AutoMapper
<https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/>
- Ponga sus controladores a dieta: POST y comandos. <https://lostechies.com/jimmybogard/2013/12/19/pon-tus-controladores-en-una-dieta-mensajes-y-comandos/>

- Abordar inquietudes transversales con una canalización de mediadores
<https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/>
-

- CQRS y REST: la combinación perfecta
<https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/>
-

- Ejemplos de canalización de MediatR
<https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/>
 - Dispositivos de prueba de corte vertical para MediatR y ASP.NET Core
<https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/>
-

- Lanzamiento de extensiones de MediatR para inyección de dependencia de Microsoft
<https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/>
-

Validación fluida

- Jeremy Skinner. FluentValidation. repositorio de GitHub.
<https://github.com/JeremySkinner/FluentValidation>

Implementar aplicaciones resilientes

Sus microservicios y aplicaciones basadas en la nube deben aceptar las fallas parciales que seguramente ocurrirán eventualmente. Debe diseñar su aplicación para que sea resistente a esos errores 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 ocurrirán y responder a ellas de una manera que evite el tiempo de inactividad o la pérdida de datos. El objetivo de la resiliencia es devolver la aplicación a un estado de pleno funcionamiento después de un error.

Ya es bastante desafiante diseñar e implementar una aplicación basada en microservicios. Pero también necesita mantener su aplicación ejecutándose en un entorno en el que es seguro algún tipo de falla. Por lo tanto, su aplicación debe ser resistente. Debe diseñarse para hacer frente a fallas parciales, como cortes de red o nodos o máquinas virtuales que fallan en la nube. Incluso los microservicios (contenedores) que se mueven a un nodo diferente dentro de un clúster pueden causar fallas breves e intermitentes dentro de la aplicación.

Los muchos componentes individuales de su aplicación también deben incorporar funciones de monitoreo de salud. Si sigue las pautas de este capítulo, puede crear una aplicación que pueda funcionar sin problemas a pesar del tiempo de inactividad transitorio o los contratiempos normales que ocurren en las implementaciones complejas y basadas en la nube.

Importante

eShopOnContainer había estado usando la [biblioteca Polly](#) para implementar la resiliencia usando [Typed Clients](#) hasta la versión 3.0.0.

A partir de la versión 3.0.0, la resiliencia de las llamadas HTTP se implementa mediante una [malla de Linkerd](#), que maneja los reintentos de manera transparente y configurable, dentro de un clúster de Kubernetes, sin tener que manejar esas preocupaciones en el código.

La biblioteca Polly todavía se usa para agregar resiliencia a las conexiones de la base de datos, especialmente al iniciar los servicios.

Advertencia

Todas las muestras de código y las imágenes de esta sección eran válidas antes de usar Linkerd y no están actualizadas para reflejar el código real actual. Así que tienen sentido en el contexto de esta sección.

Manejar fallas parciales

En los sistemas distribuidos, como las aplicaciones basadas en microservicios, existe un riesgo constante de falla parcial. Por ejemplo, un solo microservicio/contenedor puede fallar o no estar disponible para responder por un corto tiempo, o una sola máquina virtual o servidor puede fallar. Dado que 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 muy lentamente a las solicitudes o que simplemente no esté disponible durante un breve periodo 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 pedidos 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 usara RPC sincrónicos sin tiempo de espera, bloquearía los subprocessos en espera indefinidamente. Por una respuesta Ademáis de crear una mala experiencia para el usuario, cada espera que no responde consume o bloquea un subprocesso, y los subprocessos son extremadamente valiosos en aplicaciones altamente escalables. Si hay muchos subprocessos bloqueados, eventualmente el tiempo de ejecución de la aplicación puede quedarse sin subprocessos. En ese caso, la aplicación puede dejar de responder globalmente en lugar de solo parcialmente, como se muestra en la Figura 8-1.

Partial failures

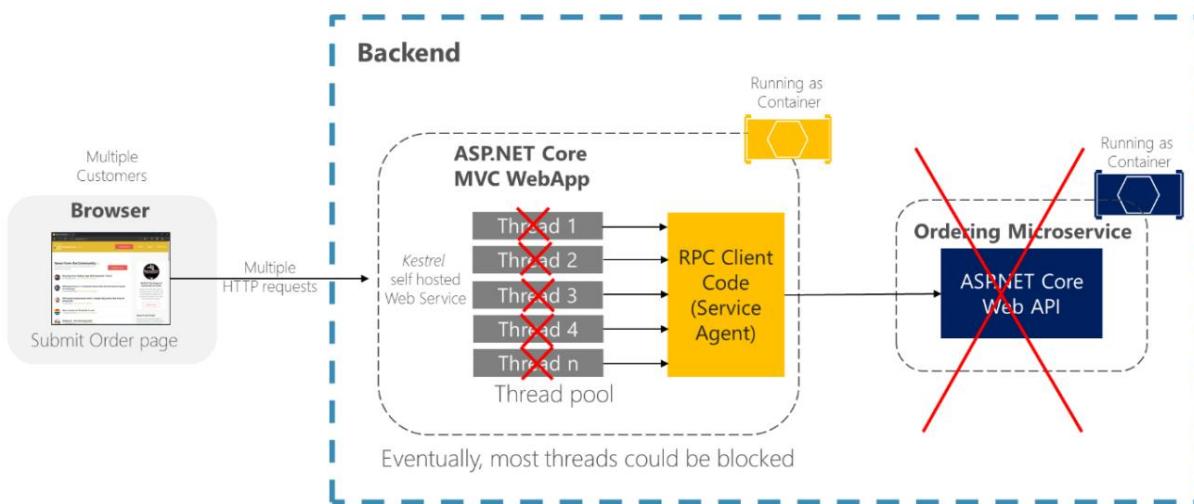


Figura 8-1. Errores parciales debido a dependencias que afectan la disponibilidad de subprocessos de servicio

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 sincrónicas (lo que se considera un patrón anti). Piense en un sistema que recibe millones de llamadas entrantes por día. Si su sistema tiene un mal diseño que se basa en largas cadenas de llamadas HTTP sincrónicas, estas llamadas entrantes pueden generar muchos más millones de llamadas salientes (supongamos una proporción de 1:4) a docenas de microservicios internos como dependencias sincrónicas. Esta situación se muestra en la Figura 8-2, especialmente la dependencia #3, que inicia una cadena, llamando a la dependencia #4, que luego llama a la #5.

Multiple distributed dependencies

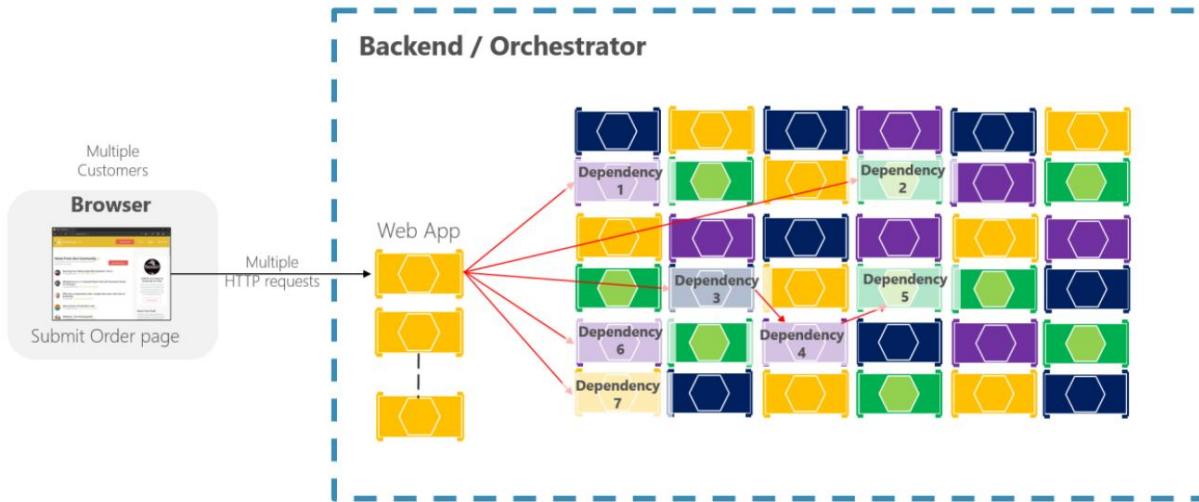


Figura 8-2. El impacto de tener un diseño incorrecto con largas cadenas de solicitudes HTTP

La falla intermitente está garantizada en un sistema distribuido y basado en la nube, incluso si cada dependencia en sí tiene una excelente disponibilidad. Es un hecho que debes considerar.

Si no diseña e implementa técnicas para garantizar la tolerancia a fallas, incluso los tiempos de inactividad pequeños pueden amplificarse. Por ejemplo, 50 dependencias cada una con un 99,99 % de disponibilidad darían como resultado varias horas de inactividad cada mes debido a este efecto dominó. Cuando una dependencia de microservicio falla mientras maneja un gran volumen de solicitudes, esa falla puede saturar rápidamente todos los subprocessos de solicitud disponibles en cada servicio y bloquear toda la aplicación.

Partial Failure Amplified in Microservices

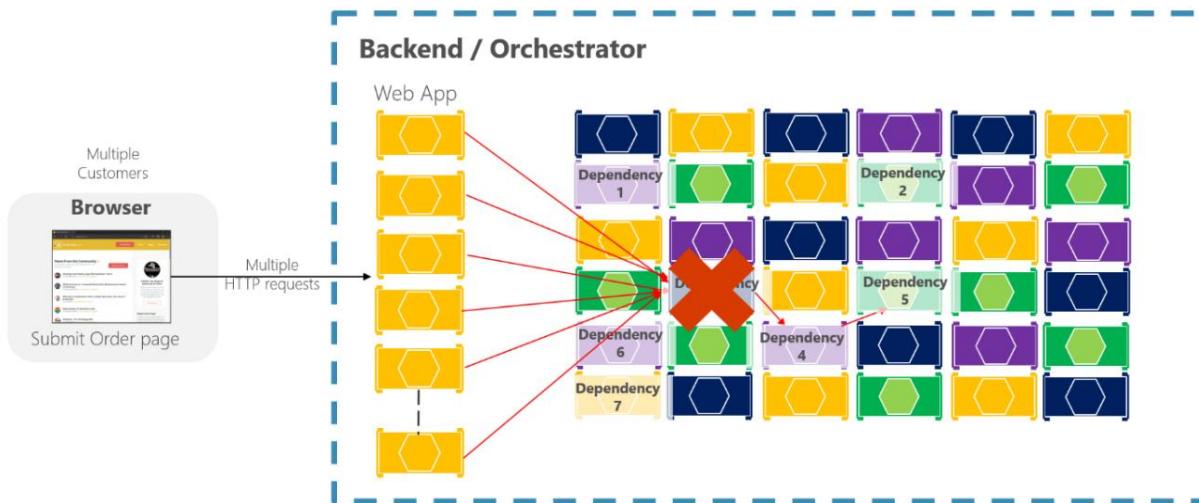


Figura 8-3. Fallo parcial amplificado por microservicios con largas cadenas de llamadas HTTP síncronas

Para minimizar este problema, en la sección [La integración de microservicios asincrónicos hace cumplir la autonomía de los microservicios, esta guía lo alienta a usar la comunicación asíncrona en los microservicios internos.](#)

Además, es esencial que diseñe sus microservicios y aplicaciones cliente para manejar fallas parciales, es decir, para crear microservicios y aplicaciones cliente resistentes.

Estrategias para manejar fallas parciales

Para lidiar con fallas parciales, use una de las estrategias descritas aquí.

Utilice la comunicación asíncrona (por ejemplo, la comunicación basada en mensajes) en los microservicios internos. Es muy recomendable no crear largas cadenas de llamadas HTTP sincrónicas a través de los microservicios internos porque ese diseño incorrecto eventualmente se convertirá en la causa principal de las malas interrupciones. Por el contrario, a excepción de las comunicaciones front-end entre las aplicaciones cliente y el primer nivel de microservicios o puertas de enlace de API de granularidad fina, se recomienda utilizar solo comunicación asíncrona (basada en mensajes) una vez pasado el ciclo inicial de solicitud/respondida, a través de los microservicios internos. La consistencia eventual y las arquitecturas basadas en eventos ayudarán a minimizar los efectos dominó. Estos enfoques imponen un mayor nivel de autonomía de los microservicios y, por lo tanto, evitan el problema señalado aquí.

Utilice reintentos con retroceso exponencial. Esta técnica ayuda a evitar fallas cortas e intermitentes al realizar reintentos de llamada un cierto número de veces, en caso de que el servicio no estuviera disponible solo por un corto tiempo. Esto puede ocurrir debido a problemas de red intermitentes o cuando un microservicio/contenedor se mueve a un nodo diferente en un clúster. Sin embargo, si estos reintentos no están diseñados correctamente con disyuntores, pueden agravar los efectos dominó y, en última instancia, incluso causar una [denegación de servicio \(DoS\)](#).

Resuelva los tiempos de espera de la red. En general, los clientes deben estar diseñados para no bloquearse indefinidamente y para usar siempre tiempos de espera cuando esperan una respuesta. El uso de tiempos de espera garantiza que los recursos nunca estén inmovilizados indefinidamente.

Utilice el patrón Disyuntor. En este enfoque, el proceso del cliente realiza un seguimiento del número de solicitudes fallidas. Si la tasa de error supera un límite configurado, se dispara un "disyuntor" para que los intentos posteriores fallen inmediatamente. (Si falla una gran cantidad de solicitudes, eso sugiere que el servicio no está disponible y que enviar solicitudes no tiene sentido). Despues de un período de tiempo de espera, el cliente debe volver a intentarlo y, si las nuevas solicitudes tienen éxito, cerrar el interruptor de circuito.

Proporcionar respaldos. En este enfoque, el proceso del cliente realiza una lógica alternativa cuando falla una solicitud, como devolver datos almacenados en caché o un valor predeterminado. Este es un enfoque adecuado para consultas y es más complejo para actualizaciones o comandos.

Límite el número de solicitudes en cola. Los clientes también deben imponer un límite superior a la cantidad de solicitudes pendientes que un microservicio de cliente puede enviar a un servicio en particular. Si se ha alcanzado el límite, probablemente no tenga sentido realizar solicitudes adicionales y esos intentos deberían fallar de inmediato. En términos de implementación, la política de [aislamiento de mamparos](#) de Polly se puede utilizar para cumplir con este requisito. [Este enfoque es esencialmente un acelerador de paralelización con SemaphoreSlim](#) como implementación. También permite una "cola" fuera del mamparo. Puede [eliminar el exceso de carga de manera proactiva](#) incluso antes de la ejecución (por ejemplo, porque la capacidad se considera llena). Esto hace que su respuesta a

ciertos escenarios de falla más rápido de lo que sería un interruptor automático, ya que el interruptor automático espera las fallas. El objeto BulkheadPolicy en [Polly expone](#) qué tan llenos están el mamparo y la cola, y ofrece eventos de desbordamiento, por lo que también se puede usar para impulsar el escalado horizontal automatizado.

Recursos adicionales

- Patrones de resistencia
<https://docs.microsoft.com/azure/architecture/framework/resiliency/reliability-patterns>
- Adición de resiliencia y optimización del rendimiento
[https://docs.microsoft.com/versiones-anteriores/msp-np/jj591574\(v=pandp.10\)](https://docs.microsoft.com/versiones-anteriores/msp-np/jj591574(v=pandp.10))
- Mamparo. repositorio de GitHub. Implementación con política Polly.
<https://github.com/App-vNext/Polly/wiki/Bulkhead>
- Diseño de aplicaciones resilientes para Azure <https://docs.microsoft.com/azure/architecture/framework/resiliency/app-design>
- Manejo de fallas transitorias
<https://docs.microsoft.com/azure/architecture/best-practices/transient-faults>

Implementar reintentos con retroceso exponencial

[Los reintentos con retraso exponencial](#) son una técnica que vuelve a intentar una operación, con un tiempo de espera que aumenta exponencialmente, hasta que se alcanza un número máximo de reintentos (el [retraso exponencial](#)). Esta técnica abarca el hecho de que los recursos de la nube pueden no estar disponibles de forma intermitente durante más de unos segundos por cualquier motivo. Por ejemplo, un orquestador podría estar moviendo un contenedor a otro nodo en un clúster para equilibrar 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 equilibrar 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 resilientes de Entity Framework Core SQL

Para Azure SQL DB, Entity Framework (EF) Core ya proporciona resiliencia de conexión de base de datos interna y lógica de reintento. Pero debe habilitar la estrategia de ejecución de Entity Framework para cada [DbContext si desea tener conexiones resistentes de EF Core](#).

Por ejemplo, el siguiente código en el nivel de conexión de EF Core habilita conexiones SQL resistentes que se vuelven a intentar si falla la conexión.

```
// Startup.cs desde cualquier clase pública de ASP.NET Core
Web API Startup {

    // Otro código...
}
```

```

public IServiceProvider ConfigureServices(servicios IServiceCollection) {
    // ...
    servicios.AddDbContext<CatalogContext>(opciones => {
        opciones.UseSqlServer(Configuración["ConnectionString"], sqlServerOptionsAction:
        sqlOptions => {
            sqlOptions.EnableRetryOnFailure( maxRetryCount:
            10, 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 realiza con EF Core se convierte en su propia operación reintentable. Cada consulta y cada llamada a SaveChanges se volverán a intentar como una unidad si se produce un error transitorio.

Sin embargo, si su código inicia una transacción mediante BeginTransaction, está definiendo su propio grupo de operaciones que deben tratarse como una unidad. Todo lo que está dentro de la transacción debe revertirse si ocurre una falla.

Si intenta ejecutar esa transacción cuando usa una estrategia de ejecución EF (política de reintento) y llama SaveChanges de múltiples DbContexts, obtendrá una excepción como esta:

System.InvalidOperationException: la estrategia de ejecución configurada 'SqlServerRetryingExecutionStrategy' no admite transacciones iniciadas por el usuario. Utilice la estrategia de ejecución devuelta por 'DbContext.Database.CreateExecutionStrategy()' para ejecutar todas las operaciones en la transacción como una unidad recuperable.

La solución es invocar manualmente la estrategia de ejecución de EF con un delegado que represente todo lo que debe ejecutarse. Si se produce un error transitorio, la estrategia de ejecución volverá a invocar al delegado. Por ejemplo, el siguiente código muestra cómo se implementa en eShopOnContainers con dos DbContext múltiples (_catalogContext e IntegrationEventLogContext) al actualizar un producto y luego guardar el objeto ProductPriceChangedIntegrationEvent, que necesita usar un DbContext diferente.

```

Public Async Task<IActionResult> UpdateProduct(
    [FromBody]CatalogItem productToUpdate)
{
    // Otro código...

    var precioantiguo = catalogItem.Precio; var
    raiseProductPriceChangedEvent = oldPrice != productToUpdate.Price;

    // Actualizar producto catalogItem
    actual = productToUpdate;
}

```

```

// Guarda los datos del producto y publica el evento de integración a través del Event Bus // si el
precio ha cambiado if (raiseProductPriceChangedEvent) {

    //Crear evento de integración para ser publicado a través del Event Bus var
    priceChangedEvent = new ProductPriceChangedIntegrationEvent(
        catalogItem.Id , productToUpdate.Price , oldPrice);

    // Lograr la atomicidad entre la operación de la base de datos del catálogo original y //
    IntegrationEventLog gracias a una transacción local await
    _catalogIntegrationEventService.SaveEventAndCatalogContextChangesAsync( priceChangedEvent);

    // Publicar a través de Event Bus y marcar el evento guardado como publicado await
    _catalogIntegrationEventService.PublishThroughEventBusAsync(
        precioCambiadoEvento);
}

// Simplemente guarde el producto actualizado porque el precio del producto no ha cambiado. más {
    esperar _catalogContext.SaveChangesAsync();
}
}

```

El primer `DbContext` es `_catalogContext` y el segundo `DbContext` está dentro del objeto `_catalogIntegrationEventService`. La acción Confirmar se realiza en todos los objetos `DbContext` mediante una estrategia de ejecución de EF.

Para lograr esta confirmación múltiple de `DbContext` , `SaveEventAndCatalogContextChangesAsync` usa un Clase `ResilientTransaction` , como se muestra en el siguiente código:

```

clase pública CatalogIntegrationEventService: ICatalogIntegrationEventService {

    //...
    Tarea pública asíncrona
    SaveEventAndCatalogContextChangesAsync( IntegrationEvent evt)
    {
        // Uso de una estrategia de resiliencia de EF Core cuando se usan varios DbContexts // dentro
        // de un BeginTransaction() explícito: // https://docs.microsoft.com/ef/core/miscellaneous/
        // connection-resiliency await ResilientTransaction.New(_catalogContext) .ExecuteAsync(async () => {

            // Lograr la atomicidad entre la base de datos del catálogo original // la
            // operación y el IntegrationEventLog gracias a una transacción local await
            _catalogContext.SaveChangesAsync(); esperar _eventLogService.SaveEventAsync(evt,
                _catalogContext.Database.CurrentTransaction.GetDbTransaction());

        });
    }
}

```

El método `ResilientTransaction.ExecuteAsync` básicamente comienza una transacción desde el `DbContext` pasado (`_catalogContext`) y luego hace que `EventLogService` use esa transacción para guardar los cambios desde `IntegrationEventLogContext` y luego confirma la transacción completa.

```

clase pública ResilientTransaction {

    privado DbContext_contexto;
}

```

```

Transacción resistente privada (contexto DbContext ) =>
    _contexto = contexto ?? lanzar una nueva excepción ArgumentNullException (nombre de (contexto));

public static ResilientTransaction New (contexto DbContext) => new
    ResilientTransaction(context);

tarea asincrónica pública ExecuteAsync(Func<Tarea> acción) {

    // Uso de una estrategia de resistencia de EF Core cuando se usan varios DbContexts //
    dentro de un BeginTransaction() explícito: // https://docs.microsoft.com/ef/core/miscellaneous/
    connection-resiliency var estrategia = _context.Database. CrearEstrategiaEjecución(); esperar
    estrategia.ExecuteAsync(async ()=> {

        esperar usando var transacción = esperar _context.Database.BeginTransactionAsync(); esperar
        acción(); esperar transacción.CommitAsync();

    });
}
}

```

Recursos adicionales

- Resiliencia de conexión e interceptación de comandos con EF en una aplicación ASP.NET MVC <https://docs.microsoft.com/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/connection-resiliency-and-interception-of-commands-with-the-entity-framework-mvc-application-part-2>
- César de la Torre. Uso de conexiones y transacciones de Resilient Entity Framework Core SQL
<https://devblogs.microsoft.com/cesardelatorre/using-resilient-entity-framework-core-sql-connections-and-transactions-retries-with-exponential-backoff/>

Use IHttpClientFactory para implementar solicitudes HTTP resilientes

[IHttpClientFactory](#) es un contrato implementado por DefaultHttpClientFactory, una fábrica obstinada, disponible desde .NET Core 2.1, para crear instancias de [HttpClient](#) para usar en sus aplicaciones.

Problemas con la clase HttpClient original disponible en .NET

La clase [HttpClient](#) original y bien conocida se puede usar fácilmente, pero en algunos casos, muchos desarrolladores no la usan correctamente.

Aunque esta clase implementa IDisposable, no se prefiere declararlo ni instanciarlo dentro de una declaración de uso porque cuando se elimina el objeto [HttpClient](#), el socket subyacente no se libera de inmediato, lo que puede provocar un problema de agotamiento del socket. Para obtener más información sobre este problema, consulte la publicación de blog [Está utilizando HttpClient incorrectamente y está desestabilizando su software.](#)

Por lo tanto, [HttpClient](#) está diseñado para ser instanciado una vez y reutilizado a lo largo de la vida de una aplicación. Instanciar una clase [HttpClient](#) para cada solicitud agotará la cantidad de sockets

disponible bajo cargas pesadas. Ese problema dará como resultado errores de `SocketException`. Los posibles enfoques para resolver ese problema se basan en la creación del objeto `HttpClient` como único o estático, como se explica en este [artículo de Microsoft sobre el uso de `HttpClient`](#). Esta puede ser una buena solución para aplicaciones de consola de corta duración o similares, que se ejecutan varias veces al día.

Otro problema con el que se encuentran los desarrolladores es cuando usan una instancia compartida de `HttpClient` en procesos de ejecución prolongada. En una situación en la que se crea una instancia de `HttpClient` como un objeto único o estático, no puede manejar los cambios de DNS como se describe en esta [edición del repositorio de GitHub dotnet/runtime](#).

Sin embargo, el problema no es realmente con `HttpClient` por sí mismo, sino con el [constructor predeterminado para `HttpClient`](#), porque crea una nueva instancia concreta de `HttpMessageHandler`, que [es la que tiene problemas de agotamiento de sockets y cambios de DNS mencionados anteriormente](#).

Para abordar los problemas mencionados anteriormente y hacer que las [instancias de `HttpClient` sean manejables](#), .NET Core 2.1 introduce la interfaz [IHttpClientFactory](#) que se puede usar para configurar y crear instancias de `HttpClient` en una aplicación a través de Inyección de dependencia (DI). También proporciona extensiones para middleware basado en Polly para aprovechar los controladores de delegación en `HttpClient`.

[Polly](#) es una biblioteca de manejo de fallas transitorias que ayuda a los desarrolladores a agregar resiliencia a sus aplicaciones mediante el uso de algunas políticas predefinidas de manera fluida y segura para subprocessos.

Beneficios de usar `IHttpClientFactory`

La implementación actual de [IHttpClientFactory](#), que también implementa [IHttpMessageHandlerFactory](#), ofrece los siguientes beneficios:

- Proporciona una ubicación central para nombrar y configurar objetos `HttpClient` lógicos. Por ejemplo, puede configurar un cliente (agente de servicio) que esté preconfigurado para acceder a un microservicio específico.
- Codifique el concepto de middleware saliente a través de la delegación de controladores en `HttpClient` e implemente middleware basado en Polly para aprovechar las políticas de Polly para la resiliencia.
- `HttpClient` ya tiene el concepto de delegar controladores que podrían vincularse para las solicitudes HTTP salientes. Puede registrar clientes HTTP en la fábrica y puede usar un controlador de Polly para usar políticas de Polly para Reintentar, Cortacircuitos, etc.
- Administre la vida útil de [HttpMessageHandler](#) para evitar los problemas/problemas mencionados que pueden ocurrir cuando administra usted mismo la vida útil de `HttpClient`.

Consejo

Las instancias de `HttpClient` inyectadas por DI se pueden desechar de forma segura, porque el `HttpMessageHandler` asociado es administrado por la fábrica. De hecho, las instancias de `HttpClient` inyectadas tienen un alcance desde una perspectiva DI.

Nota

La implementación de `IHttpClientFactory` (`DefaultHttpClientFactory`) está estrechamente ligada a la implementación de DI en el paquete `Microsoft.Extensions.DependencyInjection` NuGet. Para obtener más información sobre el uso de otros contenedores DI, consulte esta [discusión de GitHub](#).

Múltiples formas de usar IHttpclientFactory

Hay varias formas de usar IHttpclientFactory en su aplicación:

- Uso básico
- Usar clientes con nombre
- Usar clientes tipificados
- Usar clientes generados

En aras de la brevedad, esta guía muestra la forma más estructurada de usar IHttpclientFactory, que es usar clientes con tipo (patrón de agente de servicio). Sin embargo, todas las opciones están documentadas y actualmente se enumeran en este [artículo que cubre el uso de IHttpclientFactory](#).

Cómo usar clientes tipificados con IHttpclientFactory

Entonces, ¿qué es un "cliente tipificado"? Es solo un HttpClient que está preconfigurado para algún uso específico. Esta configuración puede incluir valores específicos, como el servidor base, encabezados HTTP o tiempos de espera.

El siguiente diagrama muestra cómo se utilizan los clientes tipificados con IHttpclientFactory:

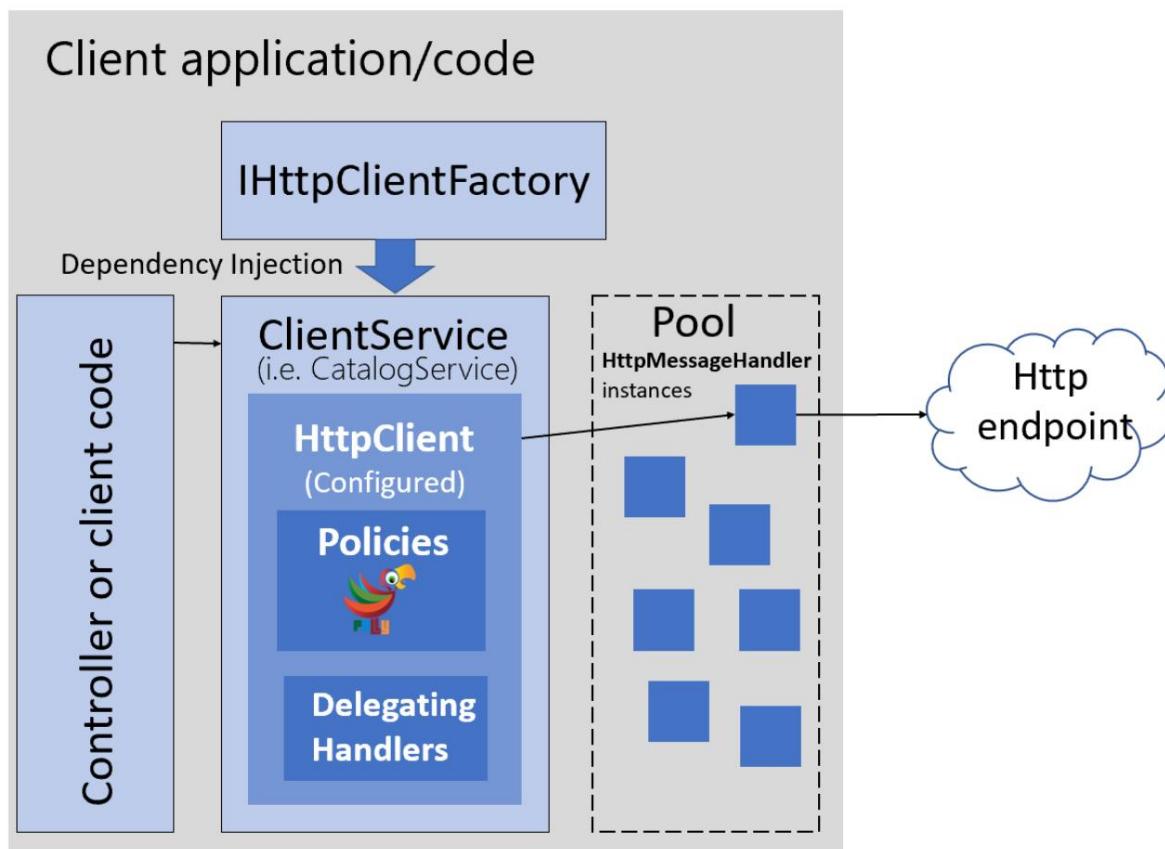


Figura 8-4. Uso de IHttpclientFactory con clases de Typed Client.

En la imagen de arriba, un ClientService (usado por un controlador o código de cliente) usa un HttpClient creado por el IHttpClientFactory registrado. Esta fábrica asigna un HttpResponseMessageHandler de un grupo al HttpClient. El HttpClient se puede configurar con las políticas de Polly al registrar el IHttpClientFactory en el contenedor DI con el método de extensión `AddHttpClient`.

Para configurar la estructura anterior, agregue `IHttpClientFactory` en su aplicación instalando el paquete Microsoft.Extensions.Http NuGet que incluye el método de extensión `AddHttpClient` para `IServiceCollection`. Este método de extensión registra la clase `DefaultHttpClientFactory` interna que se usará como singleton para la interfaz `IHttpClientFactory`. Define una configuración transitoria para `HttpMessageHandlerBuilder`. Este controlador de mensajes (`objeto HttpMessageHandler`), tomado de un grupo, lo usa el `HttpClient` devuelto de fábrica.

En el siguiente código, puede ver cómo se puede usar `AddHttpClient()` para registrar Clientes tipificados (Agentes de servicio) que necesitan usar `HttpClient`.

```
// Startup.cs //
Agregar servicios de cliente http en ConfigureServices(IServiceCollection services)
services.AddHttpClient<ICatalogService, CatalogService>(); services.AddHttpClient<IBasketService,
BasketService>(); services.AddHttpClient<IOrderingService, OrderingService>();
```

Al registrar los servicios de cliente como se muestra en el código anterior, `DefaultClientFactory` crea un `HttpClient` estándar para cada servicio. El cliente escrito se registra como transitorio con el contenedor DI. En el código anterior, `AddHttpClient()` registra `CatalogService`, `BasketService`, `OrderingService` como servicios transitorios para que puedan inyectarse y consumirse directamente sin necesidad de registros adicionales.

También podría agregar una configuración específica de la instancia en el registro para, por ejemplo, configurar la dirección base y agregar algunas políticas de resistencia, como se muestra en el siguiente código:

```
services.AddHttpClient<ICatalogService, CatalogService>(cliente => {
    cliente.BaseAddress = new Uri(Configuración["BaseUrl"]);
})
    .AddPolicyHandler(GetRetryPolicy())
    .AddPolicyHandler(GetCircuitBreakerPolicy());
```

Solo como ejemplo, puede ver una de las políticas anteriores en el siguiente código:

```
IAsyncPolicy<HttpResponseMessage> GetRetryPolicy() {
    volver HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
    retryAttempt))); }
```

Puede encontrar más detalles sobre el uso de Polly en el [artículo siguiente](#).

Tiempos de vida de `HttpClient`

Cada vez que obtiene un objeto `HttpClient` de `IHttpClientFactory`, se devuelve una nueva instancia. Pero cada `HttpClient` usa un `HttpMessageHandler` que se agrupa y reutiliza por el

IHttpClientFactory para reducir el consumo de recursos, siempre que la vida útil de HttpResponseMessageHandler no haya expirado.

La agrupación de controladores es deseable, ya que cada controlador normalmente administra sus propias conexiones HTTP subyacentes; la creación de más controladores de los necesarios puede provocar retrasos en la conexión. Algunos controladores también mantienen las conexiones abiertas indefinidamente, lo que puede evitar que el controlador reaccione a los cambios de DNS.

Los objetos HttpResponseMessageHandler en el grupo tienen una duración que es el tiempo que se puede reutilizar una instancia de HttpResponseMessageHandler en el grupo. El valor predeterminado es de dos minutos, pero se puede anular por Cliente con tipo. Para anularlo, llama a SetHandlerLifetime() en el IHttpClientBuilder que se devuelve al crear el cliente, como se muestra en el siguiente código:

```
//Establezca 5 min como tiempo de vida para los objetos HttpResponseMessageHandler en el grupo utilizado para los servicios de cliente con tipo de catálogo.AddHttpClient<ICatalogService, CatalogService>()
```

```
.SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

Cada cliente con tipo puede tener su propio valor de duración del controlador configurado. Establezca el tiempo de vida en InfiniteTimeSpan para deshabilitar la caducidad del controlador.

Implemente sus clases de Typed Client que usan el inyectado y configurado Cliente Http

Como paso previo, debe tener definidas las clases de Typed Client, como las clases en el código de muestra, como 'BasketService', 'CatalogService', 'OrderingService', etc. – Un Typed Client es una clase que acepta un HttpClient objeto (inyectado a través de su constructor) y lo usa para llamar a algún servicio HTTP remoto. Por ejemplo:

```
Servicio de catálogo de clase pública: ICatalogService {  
  
    HttpClient privado de solo lectura _httpClient;  
    cadena privada de solo lectura _remoteServiceBaseUrl;  
  
    Servicio de catálogo público (HttpClient httpClient) {  
  
        _httpClient = httpClient;  
    }  
  
    Tarea asincrónica pública <Catálogo> GetCatalogItems ( página int , toma int,  
                ¿En t? marca, ent? escribe)  
    {  
        var uri = API.Catalog.GetAllCatalogItems(_remoteServiceBaseUrl, página, toma,  
                                                marca, tipo);  
  
        var responseString = esperar _httpClient.GetStringAsync(uri);  
  
        var catálogo = JsonConvert.DeserializeObject<Catalog>(responseString); catálogo de  
        devolución ;  
    }  
}
```

El Typed Client (CatalogService en el ejemplo) se activa mediante DI (Dependency Injection), lo que significa que puede aceptar cualquier servicio registrado en su constructor, además de HttpClient.

Un cliente con tipo es efectivamente un objeto transitorio, lo que significa que se crea una nueva instancia cada vez que se necesita una. Recibe una nueva instancia de HttpClient cada vez que se construye. Sin embargo, los objetos `HttpMessageHandler` del grupo son los objetos que varias instancias de HttpClient reutilizan.

Utilice sus clases de cliente tipificado

Finalmente, una vez que haya implementado sus clases escritas, puede registrarlas y configurarlas con `AddHttpClient()`. Después de eso, puede usarlos donde sea que DI inyecte servicios. Por ejemplo, en un código de página Razor o controlador de una aplicación web MVC, como en el siguiente código de eShopOnContainers:

```
espacio de nombres Microsoft.eShopOnContainers.WebMVC.Controllers {

    clase pública CatalogController: Controlador {

        ICatalogService privado _catalogSvc;

        Controlador de catálogo público (ICatalogService catalogSvc) =>
            _catalogSvc = catalogSvc;

        Public async Task<IActionResult> Index(int? BrandFilterApplied, int? TypesFilterApplied,
                                                int? page, [FromQuery]string
                                                errorMsg)

        {
            var elementosPágina = 10;
            var catalogo = esperar _catalogSvc.GetCatalogItems(page ?? 0, itemsPage,
                                                               BrandFilterApplied,
                                                               TypesFilterApplied);

            //... Código adicional
        }

    }
}
```

Hasta este punto, el fragmento de código anterior solo ha mostrado el ejemplo de la realización de solicitudes HTTP regulares. Pero la 'magia' viene en las siguientes secciones, donde muestra cómo todas las solicitudes HTTP realizadas por HttpClient pueden tener políticas resistentes, como reintentos con retroceso exponencial, disyuntores, características de seguridad que usan tokens de autenticación o incluso cualquier otra característica personalizada. Y todo esto se puede hacer simplemente agregando políticas y delegando controladores a sus Clientes tipificados registrados.

Recursos adicionales

- Usando `HttpClientFactory` en .NET <https://docs.microsoft.com/aspnet/core/fundamentals/http-requests>
- Código fuente de `HttpClientFactory` en el repositorio de GitHub dotnet/extensions <https://github.com/dotnet/extensions/tree/v3.1.8/src/HttpClientFactory>
- Polly (biblioteca de resistencia y manejo de fallas transitorias de .NET)
 <https://theppyproject.azurewebsites.net/>

- Uso de `IHttpClientFactory` sin inyección de dependencia (problema de GitHub) <https://github.com/dotnet/extensions/issues/1345>

Implemente reintentos de llamadas HTTP con retroceso exponencial con políticas `IHttpClientFactory` y Polly

El enfoque recomendado para los reintentos con retroceso exponencial es aprovechar las bibliotecas .NET más avanzadas, como la [biblioteca Polly de código abierto](#).

Polly es una biblioteca de .NET que proporciona resistencia y capacidades de manejo de fallas transitorias. Puede implementar esas capacidades mediante la aplicación de políticas de Polly, como Reintentar, Disyuntor, Aislamiento de mamparo, Tiempo de espera y Respaldo. Polly apunta a .NET Framework 4.x y .NET Standard 1.0, 1.1 y 2.0 (que admite .NET Core y versiones posteriores).

Los siguientes pasos muestran cómo puede usar los reintentos Http con Polly integrado en `IHttpClientFactory`, que se explica en la sección anterior.

Haga referencia a los paquetes de .NET 6

`IHttpClientFactory` está disponible desde .NET Core 2.1; sin embargo, le recomendamos que use los paquetes .NET 6 más recientes de NuGet en su proyecto. Por lo general, también debe hacer referencia al paquete de extensión `Microsoft.Extensions.Http.Polly`.

Configure un cliente con la política de reintentos de Polly, en Inicio

Como se muestra en las secciones anteriores, debe definir una configuración `HttpClient` de cliente con nombre o tipo en su método `Startup.ConfigureServices(...)` estándar, pero ahora, agrega código incremental que especifica la política para los reintentos Http con retroceso exponencial, como se muestra a continuación:

```
//ConfigureServices() - Startup.cs
services.AddHttpClient<IBasketService, BasketService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5)) //Establece el tiempo de vida en cinco
    .AddPolicyHandler(GetRetryPolicy());
```

El método `AddPolicyHandler()` es lo que agrega políticas a los objetos `HttpClient` que utilizará. En este caso, está agregando una política de Polly para reintentos Http con retroceso exponencial.

Para tener un enfoque más modular, la política de reintentos Http se puede definir en un método separado dentro del archivo `Startup.cs`, como se muestra en el siguiente código:

```
IAsyncPolicy<HttpResponseMessage> GetRetryPolicy() {
    volver HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
            reintentarIntento)));
}
```

Con Polly, puede definir una política de reintentos con la cantidad de reintentos, la configuración de retroceso exponencial y las acciones a realizar cuando hay una excepción HTTP, como registrar el error. En este caso, la política está configurada para intentar seis veces con un reintentos exponencial, a partir de dos segundos.

Agregue una estrategia de fluctuación a la política de reintento

Una política de reintentos regular puede afectar su sistema en casos de alta simultaneidad y escalabilidad y bajo alta contención. Para superar los picos de reintentos similares provenientes de muchos clientes en interrupciones parciales, una buena solución es agregar una estrategia de fluctuación al algoritmo/política de reintento. Esta estrategia puede mejorar el rendimiento general del sistema de extremo a extremo. Como se recomienda en [Polly: reintentar con fluctuación, se puede implementar una buena estrategia de fluctuación mediante intervalos de reintento suaves y distribuidos uniformemente aplicados con un retraso de reintento inicial medio bien controlado en un retroceso exponencial. Este enfoque ayuda a distribuir los picos cuando surge el problema.](#) El principio se ilustra con el siguiente ejemplo:

```
var delay = Backoff.DecorrelatedJitterBackoffV2(medianFirstRetryDelay: TimeSpan.FromSeconds(1),
retryCount: 5);

var retryPolicy = Política
    .Manejar<FooException>()
    .WaitAndRetryAsync(retraso);
```

Recursos adicionales

- Patrón de reintento <https://docs.microsoft.com/azure/architecture/patterns/retry>
- Polly y IHttpConnectionFactory <https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory>
- Polly (biblioteca de resiliencia .NET y manejo de fallas transitorias) <https://github.com/App-vNext/Polly>
- Polly: Reintentar con Jitter <https://github.com/App-vNext/Polly/wiki/Retry-with-jitter>
- Marc Brooker. Jitter: mejorar las cosas con aleatoriedad <https://brooker.co.za/blog/2015/03/21/backoff.html>

Implementar el patrón del disyuntor

Como se señaló anteriormente, debe manejar las fallas que pueden tardar una cantidad de tiempo variable en recuperarse, como podría suceder cuando intenta conectarse a un servicio o recurso remoto. Manejar este tipo de falla puede mejorar la estabilidad y la resistencia de una aplicación.

En un entorno distribuido, las llamadas a recursos y servicios remotos pueden fallar debido a fallas transitorias, como conexiones de red lentas y tiempos de espera, o si los recursos responden lentamente o no están disponibles temporalmente. Estas fallas generalmente se corregirán solas después de un corto tiempo, y una aplicación en la nube sólida debe estar preparada para manejarlas mediante 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 total de un servicio. En estas situaciones, puede que no tenga sentido que una aplicación vuelva a intentar continuamente una operación que es poco probable que tenga éxito.

En su lugar, la aplicación debe codificarse para aceptar que la operación ha fallado y manejar la falla en consecuencia.

El uso descuidado de los reintentos Http podría dar lugar a la creación de un ataque de denegación de servicio (**DoS**) dentro de su propio software. Cuando un microservicio falla o funciona con lentitud, es posible que varios clientes vuelvan a intentar repetidamente las solicitudes fallidas. Eso crea un riesgo peligroso de aumentar exponencialmente el tráfico dirigido al servicio defectuoso.

Por lo tanto, necesita algún tipo de barrera de defensa para que las solicitudes excesivas se detengan cuando no vale la pena seguir intentándolo. Esa barrera de defensa es precisamente el disyuntor.

El patrón del disyuntor tiene un propósito diferente al del "patrón de reintento". El "patrón de reintento" permite que una aplicación vuelva a intentar una operación con la expectativa de que la operación finalmente se realice correctamente. El patrón Circuit Breaker evita que una aplicación realice una operación que probablemente falle. Una aplicación puede combinar estos dos patrones. Sin embargo, la lógica de reintento debe ser sensible a cualquier excepción devuelta por el interruptor automático y debe abandonar los intentos de reintento si el interruptor automático indica que una falla no es transitoria.

Implementar el patrón Circuit Breaker con IHttpclientFactory y Polly

Al igual que cuando se implementan reintentos, el enfoque recomendado para los interruptores automáticos es aprovechar las bibliotecas .NET comprobadas como Polly y su integración nativa con IHttpclientFactory.

Agregar una política de disyuntor en su canalización de middleware saliente de IHttpclientFactory es tan simple como agregar una sola pieza de código incremental a lo que ya tiene cuando usa IHttpclientFactory.

La única adición aquí al código utilizado para los reintentos de llamadas HTTP es el código en el que agrega la política de disyuntor a la lista de políticas a usar, como se muestra en el siguiente código incremental, parte del método ConfigureServices() .

```
//ConfigureServices() - Startup.cs var
retryPolicy = GetRetryPolicy(); var política de
interruptor de circuito = GetCircuitBreakerPolicy ();

services.AddHttpClient<IBasketService, BasketService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5)) // Ejemplo: el tiempo de vida predeterminado es 2
    minutos
    .AddHttpMessageHandler<HttpClientAuthorizationDelegatingHandler>()
    .AddPolicyHandler(reintentarPolítica)
    .AddPolicyHandler(política de disyuntor);
```

El método AddPolicyHandler() es lo que agrega políticas a los objetos HttpClient que utilizará. En este caso, está agregando una póliza Polly para un disyuntor.

Para tener un enfoque más modular, la política de disyuntores se define en un método separado llamado GetCircuitBreakerPolicy(), como se muestra en el siguiente código:

```
IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy() {
    volver HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(5, Intervalo de tiempo.FromSeconds (30));
}
```

En el ejemplo de código anterior, la política del disyuntor está configurada para interrumpir o abrir el circuito cuando ha habido cinco fallas consecutivas al volver a intentar las solicitudes Http. Cuando eso suceda, el circuito se romperá durante 30 segundos: en ese período, el interruptor de circuito fallará inmediatamente en las llamadas en lugar de que realmente se realicen. La política interpreta automáticamente [las excepciones relevantes y los códigos de estado HTTP como fallas.](#)

Los disyuntores también se deben usar para redirigir las solicitudes a una infraestructura alternativa si tuvo problemas en un recurso en particular que se implementa en un entorno diferente al de la aplicación cliente o el servicio que realiza la llamada HTTP. De esa forma, si hay una interrupción en el centro de datos que afecta solo a sus microservicios de backend pero no a sus aplicaciones cliente, las aplicaciones cliente pueden redirigir a los servicios alternativos. Polly está planeando una nueva política para automatizar este escenario de [política de conmutación por error .](#)

Todas esas características son para casos en los que está administrando la conmutación por error desde el código .NET, en lugar de que Azure la administre automáticamente, con transparencia de ubicación.

Desde el punto de vista del uso, cuando se usa HttpClient, no es necesario agregar nada nuevo aquí porque el código es el mismo que cuando se usa HttpClient con IHttpClientFactory, como se muestra en las secciones anteriores.

Pruebe los reintentos de Http y los disyuntores en eShopOnContainers

Cada vez que inicia la solución eShopOnContainers en un host de Docker, debe iniciar varios contenedores. Algunos de los contenedores son más lentos para iniciarse e inicializarse, como el contenedor de SQL Server. Esto es especialmente cierto la primera vez que implementa la aplicación eShopOnContainers en Docker porque necesita configurar las imágenes y la base de datos. El hecho de que algunos contenedores se inicien más lentamente que otros puede hacer que el resto de los servicios arrojen inicialmente excepciones HTTP, incluso si establece dependencias entre contenedores en el nivel de composición de docker, como se explicó en secciones anteriores. Esas dependencias de composición de ventana acopiable entre contenedores están solo en el nivel de proceso. Es posible que se haya iniciado el proceso del punto de entrada del contenedor, pero es posible que SQL Server no esté listo para las consultas. El resultado puede ser una cascada de errores y la aplicación puede obtener una excepción al intentar consumir ese contenedor en particular.

También puede ver este tipo de error en el inicio cuando la aplicación se implementa en la nube. En ese caso, los orquestadores podrían mover contenedores de un nodo o máquina virtual a otro (es decir, iniciar nuevas instancias) al equilibrar la cantidad de contenedores en los nodos del clúster.

La forma en que 'eShopOnContainers' resuelve esos problemas al iniciar todos los contenedores es mediante el patrón de reinicio ilustrado anteriormente.

Pruebe el disyuntor en eShopOnContainers

Hay algunas formas de romper/abrir el circuito y probarlo con eShopOnContainers.

Una opción es reducir el número permitido de reintentos a 1 en la política de disyuntores y volver a implementar toda la solución en Docker. Con un solo reinicio, existe una buena posibilidad de que una solicitud HTTP falle durante la implementación, se abra el interruptor de circuito y obtenga un error.

Otra opción es usar middleware personalizado que se implementa en el microservicio Basket . Cuando este middleware está habilitado, detecta todas las solicitudes HTTP y devuelve el código de estado 500. Puede habilitar el middleware realizando una solicitud GET a la URI fallida, como la siguiente:

- GET http://localhost:5103/failing Esta solicitud devuelve el estado actual del middleware. Si el middleware está habilitado, la solicitud devuelve el código de estado 500. Si el middleware está deshabilitado, no hay respuesta.
- GET http://localhost:5103/failing?enable Esta solicitud habilita el middleware.
- GET http://localhost:5103/failing?disable Esta solicitud deshabilita el middleware.

Por ejemplo, una vez que la aplicación se está ejecutando, puede habilitar el middleware realizando una solicitud utilizando el siguiente URI en cualquier navegador. Tenga en cuenta que el microservicio de pedidos utiliza el puerto 5103.

<http://localhost:5103/failing?enable>

Luego puede verificar el estado usando el URI <http://localhost:5103/failing>, como se muestra en la Figura 8-5.

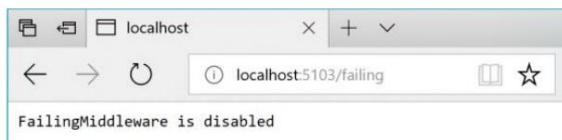


Figura 8-5. Comprobación del estado del middleware ASP.NET "fallido": en este caso, deshabilitado.

En este punto, el microservicio Basket responde con el código de estado 500 cada vez que lo llama.

Una vez que se ejecuta el middleware, puede intentar realizar 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 amigable que le indica que espere.

```
clase pública CartController: Controlador {
    //...
    tarea asincrónica pública <ActionResult> Index() {
        prueba
        {
            var usuario = _appUserParser.Parse(HttpContext.User); //Solicitudes
            Http utilizando el cliente escrito (agente de servicios) var vm = await
            _basketSvc.GetBasket(user); volver Ver (vm);

        } captura (Excepción de circuito roto) {

            // Captura el error cuando Basket.api está en modo de circuito abierto
            HandleBrokenCircuitException();

        } volver Ver();
    }

    privado vacío HandleBrokenCircuitException () {
        TempData["BasketInoperativeMsg"] = "El servicio de cesta no está operativo, inténtelo más tarde"
    }
}
```

```

    en. (Mensaje comercial debido a Circuit-Breaker)");
}
}

```

Aquí hay un resumen. La política de reintentos intenta varias veces realizar la solicitud HTTP y obtiene errores HTTP.

Cuando el número de reintentos alcanza el número máximo establecido para la política del disyuntor (en este caso, 5), la aplicación genera una `BrokenCircuitException`. El resultado es un mensaje amigable, como se muestra en la Figura 8-6.

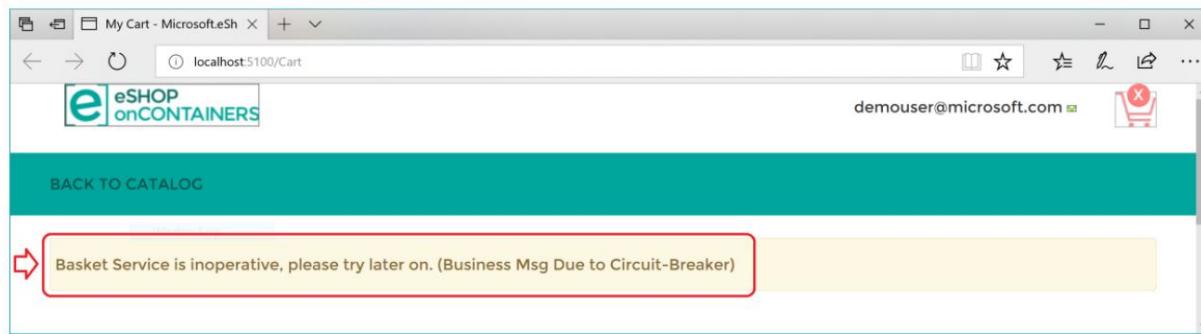


Figura 8-6. Disyuntor que devuelve un error a la interfaz de usuario

Puede implementar una lógica diferente para cuándo abrir/romper el circuito. O puede probar una solicitud HTTP en un microservicio de back-end diferente si hay un centro de datos alternativo o un sistema de back-end redundante.

Finalmente, otra posibilidad para `CircuitBreakerPolicy` es usar `Isolate` (que fuerza la apertura y mantiene abierto el circuito) y `Reset` (que lo cierra de nuevo). Estos podrían usarse para crear un punto final HTTP de utilidad que invoque `Aislar` y `Restablecer` directamente en la política. Dicho punto final HTTP también podría usarse, debidamente protegido, en producción para aislar temporalmente un sistema descendente, como cuando desea actualizarlo. O podría disparar el circuito manualmente para proteger un sistema aguas abajo que sospecha que está fallando.

Recursos adicionales

- Patrón de disyuntor <https://docs.microsoft.com/azure/architecture/patterns/circuit-breaker>

Vigilancia de la salud

El monitoreo de salud puede permitir información casi en tiempo real sobre el estado de sus contenedores y microservicios. La supervisión del estado es fundamental para varios aspectos del funcionamiento de los microservicios y es especialmente importante cuando los organizadores realizan actualizaciones parciales de aplicaciones en fases, como se explica más adelante.

Las aplicaciones basadas en microservicios a menudo usan latidos o comprobaciones de estado para permitir que sus monitores de rendimiento, programadores y orquestadores realicen un seguimiento de la multitud de servicios. Si los servicios no pueden enviar algún tipo de señal de "Estoy vivo", ya sea a pedido o según un cronograma, su aplicación podría enfrentar riesgos cuando implemente actualizaciones, o podría detectar fallas demasiado tarde y no poder detener fallas en cascada que puede terminar en apagones importantes.

En el modelo típico, los servicios envían informes sobre su estado y esa información se agrega para brindar una vista general del estado de salud de su aplicación. Si usa un orquestador, puede proporcionar información de estado al clúster de su orquestador, para que el clúster pueda actuar en consecuencia. Si invierte en informes de estado de alta calidad personalizados para su aplicación, puede detectar y solucionar problemas de su aplicación en ejecución mucho más fácilmente.

Implementar comprobaciones de estado en los servicios de ASP.NET Core

Al desarrollar un microservicio ASP.NET Core o una aplicación web, puede usar la función de verificación de estado integrada que se lanzó en ASP .NET Core 2.2

[\(Microsoft.Extensions.Diagnostics.HealthChecks\)](#). Al igual que muchas funciones de ASP.NET Core, las comprobaciones de estado vienen con un conjunto de servicios y un middleware.

Los servicios de control de estado y el middleware son fáciles de usar y brindan capacidades que le permiten validar si algún recurso externo necesario para su aplicación (como una base de datos de SQL Server o una API remota) funciona correctamente. Cuando usa esta función, también puede decidir qué significa que el recurso está en buen estado, como explicaremos más adelante.

Para usar esta función de manera efectiva, primero debe configurar los servicios en sus microservicios. En segundo lugar, necesita una aplicación de front-end que consulte los informes de salud. Esa aplicación front-end podría ser una aplicación de informes personalizada, o podría ser un orquestador en sí mismo que pueda reaccionar de acuerdo con los estados de salud.

Use la función HealthChecks en sus microservicios ASP.NET de back-end

En esta sección, aprenderá a implementar la función HealthChecks en una aplicación de API web de ASP.NET Core 6.0 de muestra al usar el paquete [Microsoft.Extensions.Diagnostics.HealthChecks](#) . La implementación de esta función en microservicios a gran escala como eShopOnContainers se explica en la siguiente sección.

Para comenzar, debe definir qué constituye un estado saludable para cada microservicio. En la aplicación de ejemplo, definimos que el microservicio está en buen estado si se puede acceder a su API a través de HTTP y su base de datos de SQL Server relacionada también está disponible.

En .NET 6, con las API integradas, puede configurar los servicios, agregar un Health Check para el microservicio y su base de datos de SQL Server dependiente de esta manera:

```
// Ejemplo de Startup.cs de .NET 6 Web API //
public void ConfigureServices(IServiceCollection services) {
    //...
    // Registra los servicios necesarios para los servicios de control de salud.AddHealthChecks()
    // Agregar una verificación de estado para una base de datos de SQL Server .AddCheck("OrderingDB-check", new SqlConnectionHealthCheck(Configuration["ConnectionString"]),
        HealthStatus.Unhealthy, new string[] { "orderingdb" });
}
```

En el código anterior, el método `services.AddHealthChecks()` configura una verificación HTTP básica que devuelve un código de estado 200 con "Saludable". Además, el método de extensión `AddCheck()` configura un `SqlConnectionHealthCheck` personalizado que verifica el estado de la base de datos SQL relacionada.

El método `AddCheck()` agrega una nueva verificación de estado con un nombre específico y la implementación del tipo `IHealthCheck`. Puede agregar varias comprobaciones de estado mediante el método `AddCheck`, por lo que un microservicio no proporcionará un estado "bueno" hasta que todas sus comprobaciones estén en buen estado.

`SqlConnectionHealthCheck` es una clase personalizada que implementa `IHealthCheck`, que toma una cadena de conexión como parámetro del constructor y ejecuta una consulta simple para verificar si la conexión a la base de datos SQL es exitosa. Devuelve `HealthCheckResult.Healthy()` si la consulta se ejecutó correctamente y un `estado de falla` con la excepción real cuando falla.

```
// Ejemplo de comprobación del estado de la conexión SQL
public class SqlConnectionHealthCheck : IHealthCheck {
```

```
    private const string DefaultTestQuery = "Seleccionar 1";

    public string ConnectionString { get; }

    cadena pública TestQuery { obtener; }

    public SqlConnectionHealthCheck(string connectionString) : this(connectionString,
        testQuery: DefaultTestQuery)

    {}

    public SqlConnectionHealthCheck(string connectionString, string testQuery) {

        Cadena de conexión = Cadena de conexión ?? tirar nuevo
        ArgumentNullException (nombre de (cadena de conexión));
        Consulta de prueba = Consulta de prueba;
    }

    tarea asincrónica pública <HealthCheckResult> CheckHealthAsync(contexto HealthCheckContext,
        CancellationToken cancellationToken = predeterminado (CancellationToken))
    {
        usando (var conexión = nueva SqlConnection(ConnectionString)) {

            prueba
            {
                esperar conexión.OpenAsync(cancellationToken);

                if (PruebaConsulta != nulo) {

                    var comando = conexión.CreateCommand();
                    comando.CommandText = TestQuery;

                    esperar comando.ExecuteNonQueryAsync(cancellationToken);
                }
            }

            } captura (DbException ex) {

                devolver nuevo HealthCheckResult (estado: context.Registration.FailureStatus,
                    excepción: ex); }

        }
```

```

    volver HealthCheckResult.Saludable();
}
}

```

Tenga en cuenta que en el código anterior, `Select 1` es la consulta utilizada para comprobar el estado de la base de datos. Para monitorear la disponibilidad de sus microservicios, los orquestadores como Kubernetes realizan periódicamente controles de estado mediante el envío de solicitudes para probar los microservicios. Es importante mantener la eficiencia de las consultas de la base de datos para que estas operaciones sean rápidas y no resulten en una mayor utilización de los recursos.

Finalmente, agregue un middleware que responda a la ruta de URL `/hc`:

```

// Ejemplo de Startup.cs de .NET 6 Web Api // public
void Configure (aplicación IApplicationBuilder,
entorno IHostingEnvironment) {

    //...
    app.UseEndpoints(puntos finales => {

        //...
        puntos finales.MapHealthChecks("/hc"); //...

    });
}

```

Cuando se invoca el punto final `<yourmicroservice>/hc`, ejecuta todas las comprobaciones de estado configuradas en el método `AddHealthChecks()` en la clase `Startup` y muestra el resultado.

Implementación de HealthChecks en eShopOnContainers

Los microservicios en `eShopOnContainers` dependen de múltiples servicios para realizar su tarea. Por ejemplo, el microservicio `Catalog.API` de `eShopOnContainers` depende de muchos servicios, como `Azure Blob Storage`, `SQL Server` y `RabbitMQ`. Por lo tanto, tiene varios controles de salud agregados mediante el método `AddCheck()`. Para cada servicio dependiente, sería necesario agregar una implementación personalizada de `IHealthCheck` que defina su estado de salud respectivo.

El proyecto de código abierto [AspNetCore.Diagnostics.HealthChecks](#) resuelve este problema al proporcionar implementaciones de verificación de estado personalizadas para cada uno de estos servicios empresariales, que se construyen sobre `.NET 6`. Cada verificación de estado está disponible como un paquete NuGet individual que puede ser fácilmente añadido al proyecto. `eShopOnContainers` los utiliza ampliamente en todos sus microservicios.

Por ejemplo, en el microservicio `Catalog.API`, se agregaron los siguientes paquetes NuGet:

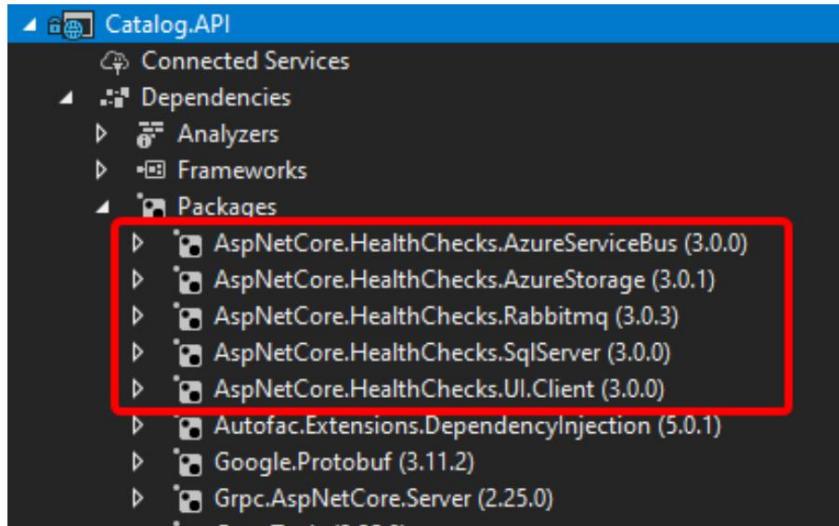


Figura 8-7. Chequeos de salud personalizados implementados en Catalog.API usando AspNetCore.Diagnostics.HealthChecks

En el siguiente código, se agregan las implementaciones de verificación de estado para cada servicio dependiente y luego se configura el middleware:

```
// Startup.cs del microservicio Catalog.api // public static
IServiceCollection AddCustomHealthCheck(este servicio de
IServiceCollection, configuración de IConfiguration) {

    var accountName = configuración.GetValue<string>("AzureStorageAccountName"); var accountKey =
    configuración.GetValue<string>("AzureStorageAccountKey");

    var hcBuilder = servicios.AddHealthChecks();

    hcBuilder .AddSqlServer( configuración["ConnectionString"],
        nombre: "CatalogDB-check", etiquetas: nueva
        cadena[] { "catalogdb" });

    if (!string.IsNullOrEmpty(nombreCuenta) && !string.IsNullOrEmpty(claveCuenta)) {
        hcBuilder
            .AddAzureBlobStorage(
                $"DefaultEndpointsProtocol=https;AccountName={accountName};AccountKey={accountKey};Endpoint Suffix=core.windows.net",
                nombre: "catalog-storage-check", etiquetas:
                nueva cadena [] { "catalogstorage" });

    } if (configuración.GetValue<bool>("AzureServiceBusEnabled")) {

        hcBuilder .AddAzureServiceBusTopic( configuración["EventBusConnection"],
            topicName: "eshop_event_bus", nombre: "catalog-
            servicebus-check", etiquetas: nueva cadena[]
            { "servicebus" });

    } más
}
```

```
{
    hcBuilder

        .AddRabbitMQ( $"amqp://{configuration["EventBusConnection"]}", nombre:
            "catalog-rabbitmqbus-check", etiquetas: nueva cadena[]
            { "rabbitmqbus" });

    }

    servicios de devolución ;
}
```

Finalmente, agregue el middleware HealthCheck para escuchar el punto final "/hc":

```
// Aplicación de middleware
HealthCheck.UseHealthChecks ("/hc", new HealthCheckOptions() {

    Predicado = => verdadero,
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
```

Consulta tus microservicios para informar sobre su estado de salud

Cuando haya configurado las comprobaciones de estado como se describe en este artículo y tenga el microservicio ejecutándose en Docker, puede comprobar directamente desde un navegador si está en buen estado. Debe publicar el puerto del contenedor en el host de Docker, de modo que pueda acceder al contenedor a través de la IP del host de Docker externo o a través de host.docker.internal, como se muestra en la figura 8-8.

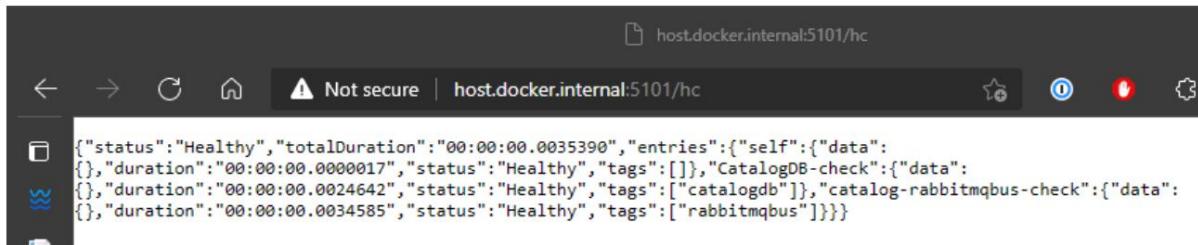


Figura 8-8. Comprobación del estado de salud de un solo servicio desde un navegador

En esa prueba, puede ver que el microservicio Catalog.API (que se ejecuta en el puerto 5101) está en buen estado y devuelve el estado HTTP 200 y la información de estado en JSON. El servicio también verificó el estado de su dependencia de la base de datos de SQL Server y RabbitMQ, por lo que la verificación de estado se informó como saludable.

Usar perros guardianes

Un perro guardián es un servicio independiente que puede observar el estado y la carga entre servicios, e informar sobre el estado de los microservicios consultando con la biblioteca HealthChecks presentada anteriormente. Esto puede ayudar a evitar errores que no se detectarían en función de la vista de un solo servicio. Los vigilantes también son un buen lugar para hospedar código que puede realizar acciones de remediación para condiciones conocidas sin la interacción del usuario.

El ejemplo de eShopOnContainers contiene una página web que muestra informes de verificación de estado de muestra, como se muestra en la Figura 8-9. Este es el perro guardián más simple que podría tener, ya que solo muestra el estado de los microservicios y las aplicaciones web en eShopOnContainers. Por lo general, un perro guardián también toma medidas cuando detecta estados no saludables.

Afortunadamente, [AspNetCore.Diagnostics.HealthChecks](#) también proporciona el paquete [NuGet AspNetCore.HealthChecks.UI](#) que se puede usar para mostrar los resultados de las comprobaciones de estado de los URI configurados.

The screenshot shows a browser window titled "Health Checks UI" with the URL "localhost:5107/hc-ui#/healthchecks". The interface displays a table of "Health Checks status" with columns: NAME, HEALTH, ON STATE FROM, and LAST EXECUTION. There are two sections of data. The first section has a header row with columns: NAME, HEALTH, DESCRIPTION, and DURATION. The second section is a detailed list of checks. A mouse cursor is hovering over the "Ordering HTTP Check" in the first section. The data is as follows:

NAME	HEALTH	DESCRIPTION	DURATION
self	Healthy		00:00:00.0000031
orderingDB-check	Healthy		00:00:00.0008153
ordering-rabbitmqbus-check	Healthy		00:00:00.0097614
+ Basket HTTP Check	Healthy	Healthy 2 minutes ago	12/12/2019, 3:41:17 PM
+ Catalog HTTP Check	Healthy	Healthy 2 minutes ago	12/12/2019, 3:41:17 PM
+ Identity HTTP Check	Healthy	Healthy 2 minutes ago	12/12/2019, 3:41:17 PM
+ Marketing HTTP Check	Healthy	Healthy 2 minutes ago	12/12/2019, 3:41:17 PM
+ Locations HTTP Check	Healthy	Healthy 2 minutes ago	12/12/2019, 3:41:18 PM
+ Payments HTTP Check	Healthy	Healthy 2 minutes ago	12/12/2019, 3:41:18 PM
+ Ordering SignalRHub HTTP Check	Healthy	Healthy 2 minutes ago	12/12/2019, 3:41:18 PM

Figura 8-9. Ejemplo de informe de control de estado en eShopOnContainers

En resumen, este servicio de vigilancia consulta el punto final "/hc" de cada microservicio. Esto ejecutará todas las comprobaciones de estado definidas en él y devolverá un estado de salud general en función de todas esas comprobaciones. HealthChecksUI es fácil de consumir con algunas entradas de configuración y dos líneas de código que deben agregarse a Startup.cs del servicio de vigilancia.

Archivo de configuración de muestra para la interfaz de usuario de comprobación de estado:

```
// Configuración {
    "HealthChecksUI": {
        "Comprobaciones de salud": [
            {
                "Nombre": "Comprobación HTTP de pedido", "Uri": "http://host.docker.internal:5102/hc", {

                "Nombre": "Solicitud de verificación de antecedentes HTTP", "Uri": "http://host.docker.internal:5111/hc" }, //... ]}
        ]
    }
}
```

Archivo Startup.cs que agrega HealthChecksUI:

```
// Startup.cs del servicio WebStatus(Watch Dog) // public void
ConfigureServices(IServiceCollection services) {

    ...
    // Registra los servicios necesarios para los controles de salud
    services.AddHealthChecksUI();

} //... public void Configurar (aplicación IApplicationBuilder, entorno IHostingEnvironment) {

    ...
    app.UseHealthChecksUI(config => config.UIPath = "/hc-ui"); //...

}
```

Comprobaciones de estado al usar orquestadores

Para monitorear la disponibilidad de sus microservicios, los orquestadores como Kubernetes y Service Fabric realizan periódicamente comprobaciones de estado mediante el envío de solicitudes para probar los microservicios. Cuando un orquestador determina que un servicio/contenedor no está en buen estado, deja de enrutar las solicitudes a esa instancia. También suele crear una nueva instancia de ese contenedor.

Por ejemplo, la mayoría de los orquestadores pueden usar comprobaciones de estado para administrar implementaciones sin tiempo de inactividad. Solo cuando el estado de un servicio/contenedor cambie a saludable, el orquestador comenzará a enrutar el tráfico a las instancias del servicio/contenedor.

La supervisión del estado es especialmente importante cuando un orquestador realiza una actualización de la aplicación. Algunos orquestadores (como Azure Service Fabric) actualizan los servicios en fases; por ejemplo, pueden actualizar una quinta parte de la superficie del clúster para cada actualización de la aplicación. El conjunto de nodos que se actualizan al mismo tiempo se denomina dominio de actualización. Después de que cada dominio de actualización se haya actualizado y esté disponible para los usuarios, ese dominio de actualización debe pasar controles de estado antes de que la implementación pase al siguiente dominio de actualización.

Otro aspecto del estado del servicio es informar las métricas del servicio. Esta es una capacidad avanzada del modelo de estado de algunos orquestadores, como Service Fabric. Las métricas son importantes cuando se usa un orquestador porque se usan para equilibrar el uso de recursos. Las métricas también pueden ser un indicador de la salud del sistema. Por ejemplo, es posible que tenga una aplicación que tenga muchos microservicios y cada instancia informe una métrica de solicitudes por segundo (RPS). Si un servicio usa más recursos (memoria, procesador, etc.) que otro servicio, el orquestador podría mover las instancias de servicio en el clúster para tratar de mantener una utilización uniforme de los recursos.

Tenga en cuenta que Azure Service Fabric proporciona su propio [modelo de Supervisión de estado](#), que es más avanzado que las simples comprobaciones de estado.

Monitoreo avanzado: visualización, análisis y alertas

La parte final del monitoreo es visualizar el flujo de eventos, informar sobre el rendimiento del servicio y alertar cuando se detecta un problema. Puede utilizar diferentes soluciones para este aspecto de la supervisión.

Puede usar aplicaciones personalizadas simples que muestren el estado de sus servicios, como la página personalizada que se muestra a [Explicar AspNetCore Diagnostics HealthChecks](#). O podría usar herramientas más avanzadas [como Azure Monitor](#) para generar alertas según el flujo de eventos.

Finalmente, si está almacenando todos los flujos de eventos, puede usar Microsoft Power BI u otras soluciones como Kibana o Splunk para visualizar los datos.

Recursos adicionales

- HealthChecks y la interfaz de usuario de HealthChecks para ASP.NET Core
<https://github.com/Xabril/AspNetCore.Diagnostics.HealthChecks>
- Introducción a la supervisión del estado de Service Fabric
<https://docs.microsoft.com/azure/service-fabric/service-fabric-health-introduction>
- Monitor azul
<https://azure.microsoft.com/services/monitor/>

Cree

microservicios .NET y Web seguros

Aplicaciones

Hay tantos aspectos sobre la seguridad en microservicios y aplicaciones web que el tema fácilmente podría llevar varios libros como este. Por lo tanto, en esta sección, nos centraremos en la autenticación, la autorización y los secretos de las aplicaciones.

Implementar autenticación en microservicios .NET y aplicaciones web

A menudo, es necesario que los recursos y las API publicados por un servicio se limiten a determinados usuarios o clientes de confianza. El primer paso para tomar este tipo de decisiones de confianza a nivel de API es la autenticación.

La autenticación es el proceso de verificar de manera confiable la identidad de un usuario.

En escenarios de microservicios, la autenticación generalmente se maneja de forma centralizada. Si usa una puerta de enlace API, la puerta de enlace es un buen lugar para autenticarse, como se muestra en la Figura 9-1. Si utiliza este enfoque, asegúrese de que no se pueda acceder directamente a los microservicios individuales (sin API Gateway) a menos que exista seguridad adicional para autenticar los mensajes, ya sea que provengan de la puerta de enlace o no.

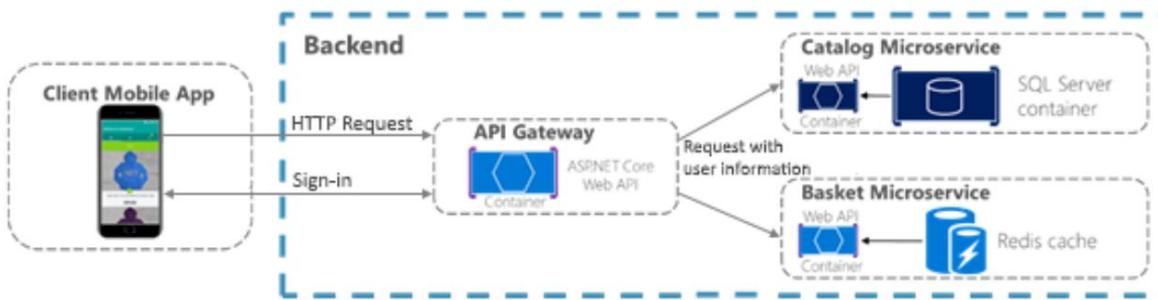


Figura 9-1. Autenticación centralizada con API Gateway

Cuando API Gateway centraliza la autenticación, agrega información del usuario al reenviar solicitudes a los microservicios. Si se puede acceder a los servicios directamente, un servicio de autenticación como Azure Active

Se puede usar un directorio 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. (Estos tokens se pueden compartir entre las aplicaciones de ASP.NET Core, si es necesario, mediante la implementación del [uso compartido de cookies](#)). Este patrón se ilustra en la figura 9-2.

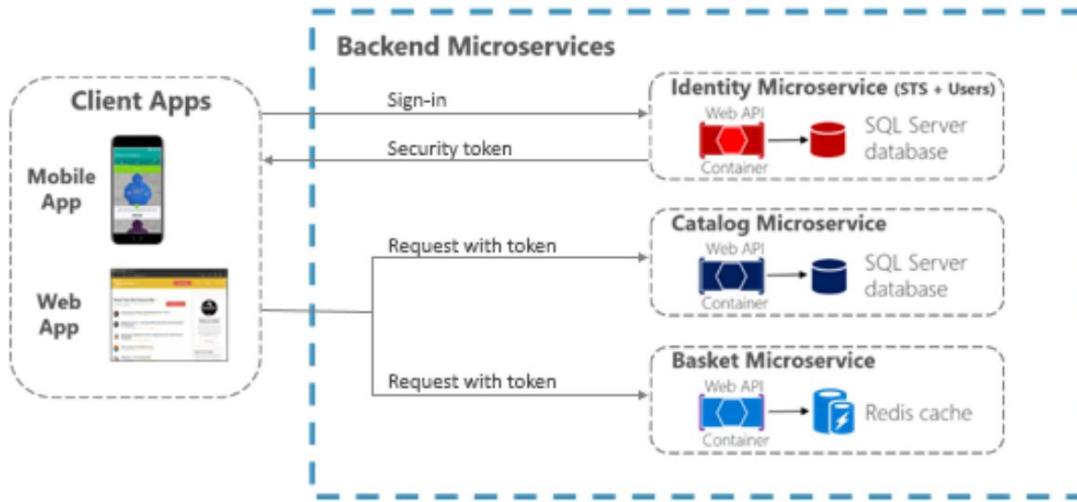


Figura 9-2. Autenticación por microservicio de identidad; la confianza se comparte mediante un token de autorización

Cuando se accede directamente a los microservicios, la confianza, que incluye autenticación y autorización, es manejada por un token de seguridad emitido por un microservicio dedicado, compartido entre microservicios.

Autenticarse 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 de ASP.NET Core Identity](#). ASP.NET Core Identity almacena la información del usuario (incluida la información de inicio de sesión, los roles y las notificaciones) en un almacén de datos configurado por el desarrollador. Normalmente, el almacén de datos de ASP.NET Core Identity es un almacén de Entity Framework proporcionado en el paquete Microsoft.AspNetCore.Identity.EntityFrameworkCore. Sin embargo, las tiendas personalizadas u otros paquetes de terceros se pueden usar para almacenar información de identidad en Azure Table Storage, CosmosDB u otras ubicaciones.

Consejo

ASP.NET Core 2.1 y versiones posteriores proporcionan [ASP.NET Core Identity](#) como [una biblioteca de clases de Razor](#), por lo que no verá gran parte del código necesario en su proyecto, como sucedía con las versiones anteriores. Para obtener detalles sobre cómo personalizar el código de identidad para que se ajuste a sus necesidades, [consulte Scaffold Identity en proyectos de ASP.NET Core](#).

El siguiente código se toma de la plantilla de proyecto MVC 3.1 de la aplicación web principal de ASP.NET con la autenticación de cuenta de usuario individual seleccionada. Muestra cómo configurar ASP.NET Core Identity mediante Entity Framework Core en el método Startup.ConfigureServices .

```

public void ConfigureServices(IServiceCollection servicios) {
    //...
    servicios.AddDbContext<ApplicationDbContext>(opciones =>
        opciones.UseSqlServer (

```

```

        Configuration.GetConnectionString("DefaultConnection"));

    services.AddDefaultIdentity<IdentityUser>(opciones =>
opciones.SignIn.RequireConfirmedAccount = verdadero)
    .AddEntityFrameworkStores<ApplicationContext>();

    servicios.AddRazorPages(); //...

}

```

Una vez que ASP.NET Core Identity está configurado, lo habilita agregando `app.UseAuthentication()` y `endpoints.MapRazorPages()` como se muestra en el siguiente código en el método `Startup.Configure` del servicio:

```

public void Configurar (aplicación IApplicationBuilder, entorno IWebHostEnvironment) {

    //...
    aplicación.UsarRouting();

    aplicación.UseAuthentication();
    aplicación.UseAuthorization();

    app.UseEndpoints(puntos finales => {

        puntos finales.MapRazorPages();

    }); //...
}

```

Importante

Las líneas del código anterior DEBEN ESTAR EN EL ORDEN MOSTRADO para que Identity funcione correctamente.

El uso de ASP.NET Core Identity permite varios escenarios:

- Cree nueva información de usuario utilizando el tipo `UserManager` (`userManager.CreateAsync`).
- Autentique a los usuarios utilizando el tipo `SignInManager`. Puede usar `signInManager.SignInAsync` para iniciar sesión directamente o `signInManager.PasswordSignInAsync` para confirmar que la contraseña del usuario es correcta y luego iniciar sesión.
- Identifique a un usuario en función de la información almacenada en una cookie (que lee el middleware ASP.NET Core Identity) para que las solicitudes posteriores de un navegador incluyan la identidad y los reclamos de un usuario que inició sesión.

ASP.NET Core Identity también admite [la autenticación de dos factores](#).

Para escenarios de autenticación que hacen uso de un almacén de datos de usuario local y que conservan la identidad entre las solicitudes que usan cookies (como es habitual en las aplicaciones web de MVC), ASP.NET Core Identity es una solución recomendada.

Autenticarse con proveedores externos

ASP.NET Core también admite el uso de [proveedores de autenticación externos](#) para permitir que los usuarios inicien sesión a través de flujos de [OAuth 2.0](#). Esto significa que los usuarios pueden iniciar sesión utilizando procesos de autenticación existentes de proveedores como Microsoft, Google, Facebook o Twitter y asociar esas identidades con una identidad de ASP.NET Core en su aplicación.

Para usar la autenticación externa, además de incluir el middleware de autenticación como se mencionó anteriormente, usando el método `app.UseAuthentication()`, también debe registrar el proveedor externo en el Inicio como se muestra en el siguiente ejemplo:

```
public void ConfigureServices(IServiceCollection servicios) {
    //...
    servicios.AddDefaultIdentity<IdentityUser>(opciones =>
        opciones.SignIn.RequireConfirmedAccount = verdadero)
            .AddEntityFrameworkStores<ApplicationContext>();

    servicios.AddAuthentication()
        .AddMicrosoftAccount(microsoftOptions => {
            microsoftOptions.ClientId = Configuración["Autenticación:Microsoft:ClientId"];
            microsoftOptions.ClientSecret = Configuración["Autenticación:Microsoft:ClientSecret"];
        })
        .AddGoogle(googleOpciones => { ... })
        .AddTwitter(opciones de twitter => { ... })
        .AddFacebook(facebookOptions => { ... });
    //}
}
```

Los proveedores de autenticación externos populares y sus paquetes NuGet asociados se muestran en la siguiente tabla:

Proveedor	Paquete
Microsoft	<code>Microsoft.AspNetCore.Authentication.MicrosoftAccount</code>
Google	<code>Microsoft.AspNetCore.Authentication.Google</code>
Facebook	<code>Microsoft.AspNetCore.Authentication.Facebook</code>
Gorjeo	<code>Microsoft.AspNetCore.Authentication.Twitter</code>

En todos los casos, debe completar un procedimiento de registro de la aplicación que depende del proveedor y que generalmente involucra:

1. Obtener una ID de aplicación de cliente.
2. Obtener un secreto de aplicación de cliente.
3. Configurar una URL de redirección, que es manejada por el middleware de autorización y el proveedor registrado
4. Opcionalmente, configurar una URL de cierre de sesión para gestionar correctamente el cierre de sesión en un escenario de inicio de sesión único (SSO).

Para obtener detalles sobre cómo configurar su aplicación para un proveedor externo, consulte [Autenticación de proveedor externo en la documentación de ASP.NET Core](#).

Consejo

Todos los detalles son manejados por el middleware de autorización y los servicios mencionados anteriormente. Entonces, solo debe elegir la opción de autenticación Cuenta de usuario individual cuando crea el proyecto de aplicación web ASP.NET Core en Visual Studio, como se muestra en la Figura 9-3, además de registrar los proveedores de autenticación mencionados anteriormente.

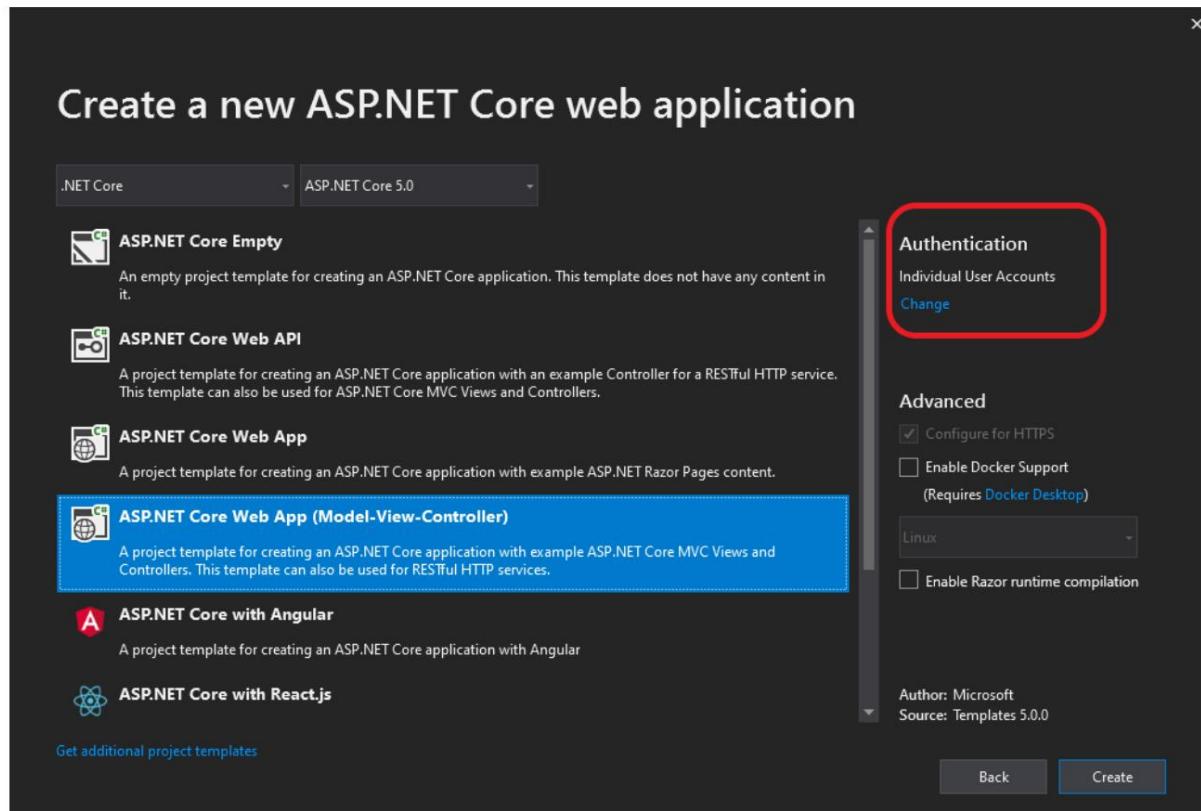


Figura 9-3. Seleccionar la opción Cuentas de usuario individuales, para usar autenticación externa, al crear un proyecto de aplicación web en Visual Studio 2019.

Además de los proveedores de autenticación externos enumerados anteriormente, hay paquetes de terceros disponibles que proporcionan middleware para usar muchos más proveedores de autenticación externos. Para obtener una lista, consulte el repositorio [AspNet.Security.OAuth.Providers](#) en GitHub.

También puede crear su propio middleware de autenticación externo para resolver alguna necesidad especial.

Autenticar con tokens de portador

La autenticación con ASP.NET Core Identity (o Identity más proveedores de autenticación externos) funciona bien para muchos escenarios de aplicaciones web en los que es apropiado almacenar información del usuario en una cookie. Sin embargo, en otros escenarios, las cookies no son un medio natural de persistencia y transmisión de datos.

Por ejemplo, en una API web de ASP.NET Core que expone extremos RESTful a los que pueden acceder aplicaciones de una sola página (SPA), clientes nativos o incluso otras API web, normalmente desea usar la autenticación de token de portador en su lugar. Estos tipos de aplicaciones no funcionan con cookies, pero pueden recuperar fácilmente un token de portador e incluirlo en el encabezado de autorización de solicitudes posteriores.

Para habilitar la autenticación de token, ASP.NET Core admite varias opciones para usar [OAuth 2.0](#) y [OpenID Connect](#).

Autenticarse con un proveedor de identidad de OpenID Connect o OAuth 2.0

Si la información del usuario se almacena en Azure Active Directory u otra solución de identidad compatible con OpenID Connect u OAuth 2.0, puede usar el paquete Microsoft.AspNetCore.Authentication.OpenIdConnect para autenticarse mediante el flujo de trabajo de OpenID Connect. Por ejemplo, para autenticarse en el microservicio Identity.Api en eShopOnContainers, una aplicación web ASP.NET Core puede usar el middleware de ese paquete, como se muestra en el siguiente ejemplo simplificado en Startup.cs:

```
// Inicio.cs

public void Configurar (aplicación IApplicationBuilder, entorno IHostingEnvironment) {
    //...
    aplicación.UsarAutenticación(); //...
    app.UseEndpoints(puntos
        finales => {

            //...
        });
}

public void ConfigureServices(servicios IServiceCollection) {
    var IdentityUrl = Configuration.GetValue<string>("IdentityUrl"); var callBackUrl =
    Configuration.GetValue<string>("CallBackUrl"); var sessionCookieLifetime =
    Configuration.GetValue("SessionCookieLifetimeMinutes", 60);

    // Agregar servicios de autenticación

    servicios.AddAuthentication(opciones => {
        opciones.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme ;
        opciones.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme ;
    })
    .AddCookie(configuración => configuración.ExpireTimeSpan = TimeSpan.FromMinutes(sessionCookieLifetime))
    .AddOpenIdConnect(opciones => {

        opciones.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme ; opciones.Autoridad =
        identidadUrl.ToString(); opciones.SignedOutRedirectUri = callBackUrl.ToString() ; opciones.ClientId =
        useLoadTest ? "mvctest" : "mvc"; opciones.ClientSecret = "secreto"; opciones.ResponseType =
        useLoadTest ? "código id_token token" : "código id_token"; opciones.SaveTokens = true;
        opciones.GetClaimsFromUserInfoEndpoint = verdadero; opciones.RequireHttpsMetadata = falso;
        opciones.Scope.Add("openid"); options.Scope.Add("perfil"); options.Scope.Add("pedidos");
        options.Scope.Add("cesta"); opciones.Scope.Add("marketing"); options.Scope.Add("ubicaciones");
        options.Scope.Add("webshoppingagg");
    });
}
```

```

        options.Scope.Add("orders.signalrhub");
    });
}

```

Cuando utiliza este flujo de trabajo, el middleware de ASP.NET Core Identity no es necesario, ya que el servicio de identidad gestiona todo el almacenamiento y la autenticación de la información del usuario.

Emitir tokens de seguridad desde un servicio ASP.NET Core

Si prefiere emitir tokens de seguridad para los usuarios locales de ASP.NET Core Identity en lugar de usar un proveedor de identidad externo, puede aprovechar algunas buenas bibliotecas de terceros.

[IdentityServer4](#) y [OpenIddict](#) son proveedores de OpenID Connect que se integran fácilmente con ASP.NET Core Identity para permitirle emitir tokens de seguridad desde un servicio ASP.NET Core. La [documentación de IdentityServer4](#) tiene instrucciones detalladas para usar la biblioteca. Sin embargo, los pasos básicos para usar IdentityServer4 para emitir tokens son los siguientes.

1. Llama a `app.UseIdentityServer` en el método `Startup.Configure` para agregar IdentityServer4 a la canalización de procesamiento de solicitudes HTTP de la aplicación. Esto permite que la biblioteca atienda solicitudes a puntos finales de OpenID Connect y OAuth2 como `/connect/token`.
2. Configure IdentityServer4 en `Startup.ConfigureServices` haciendo una llamada a `services.AddIdentityServer`.
3. Configure el servidor de identidad configurando los siguientes datos:
 - Las [credenciales](#) a usar para firmar.
 - Los [recursos de identidad y API a los](#) que los usuarios pueden solicitar acceso:
 - Los recursos de la API representan datos o funcionalidades protegidas a las que un usuario puede acceder con un token de acceso. Un ejemplo de un recurso de API sería una API web (o un conjunto de API) que requiere autorización.
 - Los recursos de identidad representan información (reclamaciones) que se proporciona a un cliente para identificar a un usuario. Las reclamaciones pueden incluir el nombre de usuario, la dirección de correo electrónico, etc.
 - Los [clientes](#) que se estarán conectando para solicitar tokens.
 - El mecanismo de almacenamiento de la información del usuario, como [ASP.NET Core Identity](#) o una alternativa.

Cuando especifica clientes y recursos para que los use IdentityServer4, puede pasar una colección [IEnumerable](#) del tipo apropiado a métodos que toman clientes en memoria o almacenes de recursos. O para escenarios más complejos, puede proporcionar tipos de proveedores de recursos o clientes a través de la inyección de dependencia.

Una configuración de muestra para IdentityServer4 para usar recursos en memoria y clientes proporcionados por un tipo de `IClientStore` personalizado podría parecerse al siguiente ejemplo:

```

public IServiceProvider ConfigureServices(IServiceCollection servicios IServiceCollection) {
    //...
}

```

```

services.AddSingleton<IClientStore, CustomClientStore>(); servicios.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<UsuarioAplicación>(); //...
}

```

Consumir tokens de seguridad

La autenticación contra un punto final de 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 proporcionados por un servicio diferente?

Para ese escenario, el middleware de autenticación que maneja tokens JWT está disponible en el paquete Microsoft.AspNetCore.Authentication.JwtBearer . JWT significa "[token web JSON](#)" y es un [formato de token de seguridad](#) común (definido por RFC 7519) para comunicar reclamos de seguridad. Un ejemplo simplificado de cómo usar middleware para consumir dichos tokens podría verse como este fragmento de código, tomado del microservicio Ordering.Api de eShopOnContainers.

```

// Inicio.cs

public void Configurar (aplicación IApplicationBuilder, entorno IHostingEnvironment) {
    //...
    // Configure la canalización para usar la aplicación de
    autenticación.UseAuthentication (); //... app.UseEndpoints(puntos
    finales => {

        //...
    });
}

public void ConfigureServices(servicios IServiceCollection) {
    var IdentityUrl = Configuration.GetValue<string>("IdentityUrl");

    // Agregar servicios de autenticación

    servicios.AddAuthentication(opciones => {
        opciones.DefaultAuthenticateScheme =
            AspNetCore.Authentication.JwtBearer.JwtBearerDefaults.AuthenticationScheme;
        opciones.DefaultChallengeScheme =
            AspNetCore.Authentication.JwtBearer.JwtBearerDefaults.AuthenticationScheme;

        }).AddJwtBearer(opciones => {

            opciones.Autoridad = identidadUrl;
            opciones.RequireHttpsMetadata = falso;
            opciones.Audiencia = "pedidos";
        });
}

```

Los parámetros en este uso son:

- La audiencia representa el receptor del token entrante o el recurso al que el token otorga acceso. Si el valor especificado en este parámetro no coincide con el parámetro en el token, el token será rechazado.
- La autoridad es la dirección del servidor de autenticación que emite el token. El middleware de autenticación del portador JWT usa este URI para obtener la clave pública que se puede usar para validar la firma del token. El middleware también confirma que el parámetro iss en el token coincide con este URI.

Otro parámetro, RequireHttpsMetadata, es útil para realizar pruebas; establece este parámetro en falso para que pueda probar en entornos donde no tiene certificados. En las implementaciones del mundo real, los tokens de portador JWT siempre deben pasarse solo a través de HTTPS.

Con este middleware implementado, los tokens JWT se extraen automáticamente de los encabezados de autorización. Luego se deserializan, se validan (usando los valores en los parámetros Audiencia y Autoridad) y se almacenan como información de usuario para que las acciones de MVC o los filtros de autorización hagan referencia a ellos más adelante.

El middleware de autenticación del portador JWT también puede admitir 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

- Compartir cookies entre aplicaciones <https://docs.microsoft.com/aspnet/core/security/cookie-sharing>
- Introducción a la identidad <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>
- Habilitación de la autenticación mediante Facebook, Google y otros proveedores externos <https://docs.microsoft.com/aspnet/core/security/authentication/social/>
- Michell Anicas. Una introducción a OAuth 2 <https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>
- AspNet.Security.OAuth.Providers (repositorio de GitHub para proveedores de ASP.NET OAuth) <https://github.com/aspnet-contrib/AspNet.Security.OAuth.Providers/tree/dev/src>
- IdentityServer4. Documentación oficial <https://identityserver4.readthedocs.io/en/latest/>

Acerca de la autorización en microservicios .NET y aplicaciones web

Después de la autenticación, las API web de ASP.NET Core deben autorizar el acceso. Este proceso permite que un servicio haga que las API estén disponibles para algunos usuarios autenticados, pero no para todos. [La autorización](#) se puede realizar en función de los roles de los usuarios o en función de una política personalizada, que puede incluir la inspección de notificaciones u otras heurísticas.

Restringir el acceso a una ruta ASP.NET Core MVC es tan fácil como aplicar un atributo Authorize al método de acción (o a la clase del controlador si todas las acciones del controlador requieren autorización), como se muestra en el siguiente ejemplo:

```
Clase pública AccountController: Controlador {
    Inicio de sesión de resultado de acción público
    () {}

    [Authorize]
    cierre de sesión público de ActionResult
    () {}

}
```

De manera predeterminada, agregar un atributo Autorizar sin parámetros limitará el acceso a los usuarios autenticados para ese controlador o acción. Para restringir aún más una API para que esté disponible solo para usuarios específicos, el atributo se puede expandir para especificar roles o políticas requeridas que los usuarios deben cumplir.

Implementar la autorización basada en roles

ASP.NET Core Identity tiene un concepto integrado de funciones. Además de los usuarios, ASP.NET Core Identity almacena información sobre los diferentes roles utilizados por la aplicación y realiza un seguimiento de qué usuarios están asignados a qué roles. Estas asignaciones se pueden cambiar mediante programación con el tipo RoleManager que actualiza los roles en el almacenamiento persistente y el tipo UserManager que puede otorgar o revocar roles de los usuarios.

Si se está autenticando con tokens de portador de JWT, el middleware de autenticación de portador de ASP.NET Core JWT completará los roles de un usuario en función de las notificaciones de roles que se encuentran en el token. Para limitar el acceso a una acción o controlador de MVC a los usuarios en roles específicos, puede incluir un parámetro Roles en la anotación Autorizar (atributo), como se muestra en el siguiente fragmento de código:

```
[Autorizar(Roles = "Administrador, Usuario avanzado")] clase
pública ControlPanelController : Controller {

    Public ActionResult SetTime() {}

    [Autorizar(Roles = "Administrador")] public
    ActionResult ShutDown() {}

}
```

En este ejemplo, solo los usuarios con roles de Administrador o Usuario avanzado pueden acceder a las API en el controlador del Panel de control (como ejecutar la acción Establecer hora). La API ShutDown está aún más restringida para permitir el acceso solo a los usuarios con la función de administrador.

Para exigir que un usuario tenga varias funciones, utilice varios atributos de autorización, como se muestra en el siguiente ejemplo:

```
[Authorize(Roles = "Administrador, Usuario avanzado")]
[Authorize(Roles = "EmpleadoRemoto")]
[Authorize(Policy = "PolíticaPersonalizada")]
público ActionResult API1 () { }
```

En este ejemplo, para llamar a API1, un usuario debe:

- Estar en el rol de administrador o usuario avanzado, y
- Estar en el rol de RemoteEmployee, y
- Satisfaga un controlador personalizado para la autorización de CustomPolicy.

Implementar la autorización basada en políticas

Las reglas de autorización personalizadas también se pueden escribir mediante [políticas de autorización](#). Esta sección proporciona una descripción general. Para obtener más información, consulte el [Taller de autorización de ASP.NET](#).

Las políticas de autorización personalizadas se registran en el método Startup.ConfigureServices mediante el método service.AddAuthorization. Este método toma un delegado que configura un argumento AuthorizationOptions.

```
servicios.AddAuthorization(opciones => {
    opciones.AddPolicy("AdministratorsOnly", política =>
        política.RequireRole("Administrador"));

    opciones.AddPolicy("EmployeesOnly", policy =>
        policy.RequireClaim("EmployeeNumber"));

    opciones.AddPolicy("Mayores de 21", política =>
        política.Requisitos.Add(nuevo requisito de edad mínima(21)));
});
```

Como se muestra en el ejemplo, las políticas se pueden asociar con diferentes tipos de requisitos. Una vez registradas las políticas, se pueden aplicar a una acción o controlador pasando el nombre de la política como argumento de política del atributo Autorizar (por ejemplo, [Authorize(Policy="EmployeesOnly")]). Las políticas pueden tener varios requisitos, no solo uno (como se muestra en estos ejemplos).

En el ejemplo anterior, la primera llamada AddPolicy es solo una forma alternativa de autorizar por rol. Si se aplica [Authorize(Policy="AdministratorsOnly")] a una API, solo los usuarios con la función de administrador podrán acceder a ella.

La segunda llamada `AddPolicy` demuestra una forma fácil de exigir que un reclamo particular esté presente para el usuario. El método `RequireClaim` también toma opcionalmente los valores esperados para la notificación. Si se especifican valores, el requisito se cumple solo si el usuario tiene una notificación del tipo correcto y uno de los valores especificados. Si usa el middleware de autenticación de portador de JWT, todas las propiedades de JWT estarán disponibles como reclamos de usuario.

La política más interesante que se muestra aquí está en el tercer método `AddPolicy`, porque utiliza un requisito de autorización personalizado. Mediante el uso de requisitos de autorización personalizados, puede tener un gran control sobre cómo se realiza la autorización. Para que esto funcione, debe implementar estos tipos:

- Un tipo de requisitos 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 de requisito de edad mínima de muestra .
- Un controlador que implementa `AuthorizationHandler`, donde `T` es el tipo de `IAuthorizationRequirement` que el controlador puede satisfacer. El controlador debe implementar el método `HandleRequirementAsync`, que comprueba si un contexto específico que contiene información sobre el usuario cumple el requisito.

Si el usuario cumple con el requisito, una llamada a `context.Succeed` indicará que el usuario está autorizado.

Si hay varias formas en que un usuario puede satisfacer un requisito de autorización, se pueden crear varios controladores.

Además de registrar requisitos de política personalizados con llamadas `AddPolicy`, también debe registrar controladores de requisitos personalizados a través de Inyección de dependencia (`services.AddTransient<IAuthorizationHandler, MinimalAgeHandler>()`).

En la documentación de autorización de ASP.NET Core se encuentra disponible un ejemplo de un controlador y un requisito de autorización personalizados para comprobar la edad de un usuario (básado en una notificación `DateOfBirth`).

Recursos adicionales

- Autenticación principal de ASP.NET

<https://docs.microsoft.com/aspnet/core/security/authentication/identity>
- Autorización de ASP.NET Core

<https://docs.microsoft.com/aspnet/core/security/authorization/introduction>

 - Autorización basada en funciones

<https://docs.microsoft.com/aspnet/core/security/authorization/roles>
 - Autorización personalizada basada en políticas

<https://docs.microsoft.com/aspnet/core/security/authorization/policies>

Almacene los secretos de la aplicación de forma segura durante el desarrollo

Para conectarse con recursos protegidos y otros servicios, las aplicaciones ASP.NET Core normalmente necesitan usar cadenas de conexión, contraseñas u otras credenciales que contienen información confidencial. Estas piezas sensibles de información se llaman secretos. Es una buena práctica no incluir secretos en la fuente.

código y asegurándose de no almacenar secretos en el control de código fuente. En su lugar, debe usar el modelo de configuración de ASP.NET Core para leer los secretos desde ubicaciones más seguras.

Debe separar los secretos para acceder a los recursos de desarrollo y preparación de los que se utilizan para acceder a los recursos de producción, porque diferentes personas necesitarán acceso a esos diferentes conjuntos de secretos. Para almacenar secretos usados durante el desarrollo, los enfoques comunes son almacenar secretos en variables de entorno o usar la herramienta ASP.NET Core Secret Manager. Para un almacenamiento más seguro en entornos de producción, los microservicios pueden almacenar secretos en Azure Key Vault.

Almacenar secretos en variables de entorno

Una forma de mantener los secretos fuera del código fuente es que los desarrolladores establezcan secretos basados en cadenas como variables de entorno en sus máquinas de desarrollo. Cuando utiliza variables de entorno para almacenar secretos con nombres jerárquicos, como los anidados en las secciones de configuración, debe nombrar las variables para incluir la jerarquía completa de sus secciones, delimitadas con dos puntos (:).

Por ejemplo, establecer una variable de entorno Logging:LogLevel:Default en el valor Debug sería equivalente a un valor de configuración del siguiente archivo JSON:

```
{
  "Registro": {
    "Nivel de registro": {
      "Predeterminado": "Depurar"
    }
  }
}
```

Para acceder a estos valores desde las variables de entorno, la aplicación solo necesita llamar a AddEnvironmentVariables en su ConfigurationBuilder al construir un objeto IConfigurationRoot .

Nota

Las variables de entorno se almacenan comúnmente como texto sin formato, por lo que si la máquina o el proceso con las variables de entorno se ven comprometidos, los valores de las variables de entorno serán visibles.

Almacene secretos con ASP.NET Core Secret Manager

La herramienta ASP.NET Core [Secret Manager](#) proporciona otro método para mantener los secretos fuera del código fuente durante el desarrollo. Para usar la herramienta Secret Manager, instale el paquete Microsoft.Extensions.Configuration.SecretManager en su archivo de proyecto. Una vez que esa dependencia está presente y se ha restaurado, se puede usar el comando dotnet user-secrets para establecer el valor de los secretos desde la línea de comando. Estos secretos se almacenarán en un archivo JSON en el directorio del perfil del usuario (los detalles varían según el sistema operativo), lejos del código fuente.

Los secretos establecidos por la herramienta Administrador de secretos están organizados por la propiedad UserSecretsId del proyecto que usa los secretos. Por lo tanto, debe asegurarse de establecer la propiedad UserSecretsId en su archivo de proyecto, como se muestra en el fragmento a continuación. El valor predeterminado es un GUID asignado por Visual Studio, pero la cadena real no es importante siempre que sea única en su computadora.

```
<Grupo de propiedades>
<UserSecretsId>Cadena de identificación única</UserSecretsId>
</Grupo de propiedades>
```

El uso de secretos almacenados con Secret Manager en una aplicación se logra llamando a AddUserSecrets<T> en la instancia de ConfigurationBuilder para incluir secretos para la aplicación en su configuración. El parámetro genérico T debe ser un tipo del ensamblado al que se aplicó UserSecretId. Por lo general, usar AddUserSecrets<Startup> está bien.

AddUserSecrets <Startup>() se incluye en las opciones predeterminadas para el entorno de desarrollo cuando se usa el método CreateDefaultBuilder en Program.cs .

Use Azure Key Vault para proteger los secretos en el momento de la producción

Los secretos almacenados como variables de entorno o almacenados por la herramienta Secret Manager todavía se almacenan localmente y sin cifrar en la máquina. Una opción más segura para almacenar secretos es [Azure Key Vault](#), que proporciona una ubicación central segura para almacenar claves y secretos.

El paquete `Azure.Extensions.AspNetCore.Configuration.Secrets` permite que una aplicación ASP.NET Core lea la información de configuración de Azure Key Vault. Para comenzar a usar secretos de Azure Key Vault, siga estos pasos:

1. Registre su aplicación como una aplicación de Azure AD. (Azure AD administra el acceso a los almacenes de claves). Esto se puede hacer a través del portal de administración de Azure.

Como alternativa, si desea que su aplicación se autentique mediante un certificado en lugar de una contraseña o un secreto de cliente, puede usar el cmdlet de PowerShell [New-AzADApplication](#) . El certificado que registra con Azure Key Vault solo necesita su clave pública. Su aplicación utilizará la clave privada.

2. Proporcione a la aplicación registrada acceso al almacén de claves mediante la creación de una nueva entidad de servicio. Tú puedes hacer esto usando los siguientes comandos de PowerShell:

```
$sp = New-AzADServicePrincipal -ApplicationId "<guid de ID de aplicación>"
Set-AzKeyVaultAccessPolicy -VaultName "<VaultName>" -ServicePrincipalName $sp.ServicePrincipalNames[0]
-PermissionsToSecrets all -ResourceGroupName "<KeyVault Resource Group>"
```

3. Incluya el almacén de claves como fuente de configuración en su aplicación llamando al método de extensión `AzureKeyVaultConfigurationExtensions.AddAzureKeyVault` cuando cree una instancia de `IConfigurationRoot` .

Tenga en cuenta que llamar a `AddAzureKeyVault` requiere el identificador de la aplicación que se registró y se le dio acceso al almacén de claves en los pasos anteriores. O puede ejecutar primero el comando de la CLI de Azure: az login, luego usar una sobrecarga de `AddAzureKeyVault` que toma una `DefaultAzureCredential` en lugar del cliente.

Importante

Le recomendamos que registre Azure Key Vault como el último proveedor de configuración, de modo que pueda anular los valores de configuración de los proveedores anteriores.

Recursos adicionales

- Uso de Azure Key Vault para proteger los secretos de las aplicaciones <https://docs.microsoft.com/azure/architecture/multitenant-identity>
- Almacenamiento seguro de secretos de aplicaciones durante el desarrollo <https://docs.microsoft.com/aspnet/core/security/app-secrets>
- Configuración de la protección de datos <https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/overview>
- Gestión de claves de protección de datos y vida útil en ASP.NET Core <https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/default-settings>

Microservicios .NET

Conclusiones clave de la arquitectura

Como resumen y conclusiones clave, las siguientes son las conclusiones más importantes de esta guía.

Beneficios de usar contenedores. Las soluciones basadas en contenedores brindan importantes ahorros de costos porque ayudan a reducir los problemas de implementación causados por fallas en las dependencias en los entornos de producción. Los contenedores mejoran significativamente DevOps y las operaciones de producción.

Los contenedores serán omnipresentes. Los contenedores basados en Docker se están convirtiendo en el estándar de facto de la industria, respaldados por proveedores clave en los ecosistemas de Windows y Linux, como Microsoft, Amazon AWS, Google e IBM. Es probable que Docker pronto sea omnipresente tanto en la nube como en los centros de datos locales.

Los contenedores como unidad de despliegue. Un contenedor Docker se está convirtiendo en la unidad estándar de implementación para cualquier aplicación o servicio basado en servidor.

Microservicios. La arquitectura de microservicios se está convirtiendo en el enfoque preferido para aplicaciones de misión crítica distribuidas y grandes o complejas basadas en muchos subsistemas independientes en forma de servicios autónomos. En una arquitectura basada en microservicios, la aplicación se crea como una colección de servicios que se desarrollan, prueban, versionan, implementan y escalan de forma independiente. Cada servicio puede incluir cualquier base de datos autónoma relacionada.

Diseño dirigido por dominio y SOA. Los patrones de arquitectura de microservicios se derivan de la arquitectura orientada a servicios (SOA) y el diseño controlado por dominio (DDD). Cuando diseña y desarrolla microservicios para entornos con necesidades y reglas comerciales en constante evolución, es importante considerar los enfoques y patrones de DDD.

Desafíos de los microservicios. Los microservicios ofrecen muchas capacidades poderosas, como implementación independiente, fuertes límites de subsistemas y diversidad tecnológica. Sin embargo, también plantean muchos desafíos nuevos relacionados con el desarrollo de aplicaciones distribuidas, como modelos de datos fragmentados e independientes, comunicación resistente entre microservicios, consistencia eventual y complejidad operativa que resulta de agregar información de registro y monitoreo de múltiples microservicios. Estos aspectos introducen un nivel de complejidad mucho mayor que una aplicación monolítica tradicional. Como resultado, solo los escenarios específicos son adecuados para las aplicaciones basadas en microservicios. Estos incluyen aplicaciones grandes y complejas con múltiples subsistemas en evolución. En estos casos, vale la pena invertir en una arquitectura de software más compleja, porque proporcionará una mejor agilidad y mantenimiento de aplicaciones a largo plazo.

Contenedores para cualquier aplicación. Los contenedores son convenientes para los microservicios, pero también pueden ser útiles para aplicaciones monolíticas basadas en .NET Framework tradicional, cuando se usan contenedores de Windows. Los beneficios de usar Docker, como resolver muchos problemas de implementación a producción y proporcionar entornos de desarrollo y prueba de última generación, se aplican a muchos tipos diferentes de aplicaciones.

CLI frente a IDE. Con las herramientas de Microsoft, puede desarrollar aplicaciones .NET en contenedores utilizando su enfoque preferido. Puede desarrollar con una CLI y un entorno basado en un editor mediante la CLI de Docker y Visual Studio Code. O puede usar un enfoque centrado en IDE con Visual Studio y sus características únicas para Docker, como la depuración de varios contenedores.

Aplicaciones en la nube resistentes. En los sistemas basados en la nube y los sistemas distribuidos en general, siempre existe el riesgo de falla parcial. Dado que los clientes y los servicios son procesos separados (contenedores), es posible que un servicio no pueda responder de manera oportuna a la solicitud de un cliente. Por ejemplo, un servicio puede estar inactivo debido a una falla parcial o por mantenimiento; el servicio puede estar sobrecargado y responder lentamente a las solicitudes; o puede que no sea accesible por un corto tiempo debido a problemas de red. Por lo tanto, una aplicación basada en la nube debe aceptar esas fallas y tener una estrategia para responder a esas fallas. Estas estrategias pueden incluir políticas de reintento (reenvío de mensajes o reintento de solicitudes) e implementación de patrones de disyuntores para evitar la carga exponencial de solicitudes repetidas. Básicamente, las aplicaciones basadas en la nube deben tener mecanismos resistentes, ya sea basados en la infraestructura de la nube o personalizados, como los de alto nivel proporcionados por orquestadores o buses de servicio.

Seguridad. Nuestro mundo moderno de contenedores y microservicios puede exponer nuevas vulnerabilidades. Existen varias formas de implementar la seguridad básica de las aplicaciones, en función de la autenticación y la autorización.

Sin embargo, la seguridad de los contenedores debe considerar componentes clave adicionales que den como resultado aplicaciones inherentemente más seguras. Un elemento fundamental 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 similares, comúnmente denominados secretos de aplicaciones. Cualquier solución segura debe seguir las mejores prácticas de seguridad, como cifrar secretos mientras están en tránsito y en reposo, y evitar que los secretos se filtrencen cuando los consuma la aplicación final. Esos secretos deben almacenarse y mantenerse de forma segura, como cuando se usa Azure Key Vault.

Orquestadores. Los orquestadores basados en contenedores, como Azure Kubernetes Service y Azure Service Fabric, son una parte clave de cualquier microservicio importante y aplicación basada en contenedores. Estas aplicaciones llevan necesidades de alta complejidad, escalabilidad y están en constante evolución. Esta guía ha presentado orquestadores y su rol en soluciones basadas en microservicios y contenedores. Si las necesidades de su aplicación lo llevan hacia aplicaciones complejas en contenedores, le resultará útil buscar recursos adicionales para obtener más información sobre orquestadores.