

# Contents

[Microsserviços do .NET: Arquitetura de aplicativos .NET em contêineres](#)

[Introdução aos contêineres e ao Docker](#)

[O que é o Docker?](#)

[Terminologia do Docker](#)

[Registros, imagens e contêineres do Docker](#)

[Escolher entre .NET Core e .NET Framework para contêineres do Docker](#)

[Diretrizes gerais](#)

[Quando escolher o .NET Core para os contêineres do Docker](#)

[Quando escolher o .NET Framework para os contêineres do Docker](#)

[Tabela de decisões: estruturas .NET a serem usadas para o Docker](#)

[Para qual sistema operacional direcionar com os contêineres do .NET](#)

[Imagens oficiais do .NET Docker](#)

[Arquitetando aplicativos baseados em contêineres e em microsserviços](#)

[Implantar aplicativos monolíticos em contêineres](#)

[Estado e dados em aplicativos do Docker](#)

[Arquitetura orientada a serviço](#)

[Arquitetura de microsserviços](#)

[Soberania de dados por microsserviço](#)

[Arquitetura lógica versus arquitetura física](#)

[Desafios e soluções do gerenciamento de dados distribuídos](#)

[Identificando os limites de modelo de domínio de cada microsserviço](#)

[Comunicação direta do cliente para microsserviços versus o padrão de Gateway de API](#)

[Comunicação em uma arquitetura de microsserviço](#)

[Comunicação assíncrona baseada em mensagens](#)

[Criando, evoluindo e fazendo o controle de versão de APIs e de contratos de microsserviços](#)

[Capacidade de endereçamento de microsserviços e o Registro do serviço](#)

[Criando interface do usuário de composição baseada em microsserviços, incluindo](#)

- forma e layout visuais da interface do usuário gerados por vários microsserviços
- Resiliência e a alta disponibilidade em microsserviços
- Orquestrar microsserviços e aplicativos de vários contêineres para alta escalabilidade e disponibilidade
- Processo de desenvolvimento de aplicativos baseados no Docker
  - Fluxo de trabalho de desenvolvimento para aplicativos do Docker
  - Projetar e desenvolver aplicativos .NET baseados em vários contêineres e em microsserviços
  - Projetando um aplicativo orientado a microsserviços
  - Criando um microsserviço de CRUD simples controlado por dados
  - Definindo o aplicativo de vários contêineres com o docker-compose.yml
  - Usando um servidor de banco de dados em execução como contêiner
  - Implementando comunicação baseada em evento entre microsserviços (eventos de integração)
  - Implementando um barramento de eventos com o RabbitMQ para o ambiente de desenvolvimento ou de teste
- Assinando eventos
- Testar serviços e aplicativos Web do ASP.NET Core
- Implementar tarefas em segundo plano em microsserviços com IHostedService
- Implementação de Gateways de API com o Ocelot
- Lidando com a complexidade de negócios em um microsserviço com padrões DDD e CQRS
  - Aplicando padrões CQRS e DDD simplificados em um microsserviço
  - Aplicando abordagens CQRS e CQS em um microsserviço DDD em eShopOnContainers
  - Implementando leituras/consultas em um microsserviço CQRS
  - Projetando um microsserviço orientado a DDD
  - Criando um modelo de domínio de microsserviço
  - Implementando um modelo de domínio de microsserviço com o .NET Core
  - Seedwork (classes e interfaces base reutilizáveis para seu modelo de domínio)
  - Implementando objetos de valor
  - Usando classes Enumeration em vez de tipos enum
  - Projetando validações na camada de modelo de domínio
  - Validação do lado do cliente (validação nas camadas de apresentação)

Eventos de domínio: design e implementação

Projetando a camada de persistência da infraestrutura

Implementando a camada de persistência da infraestrutura com o Entity Framework Core

Usando bancos de dados NoSQL como uma infraestrutura de persistência

Projetando a camada de aplicativos de microsserviço e a API Web

Implementando a camada de aplicativos de microsserviço usando a API Web

Implementando aplicativos resilientes

Tratando falha parcial

Estratégias para tratar falhas parciais

Implementando novas tentativas com retirada exponencial

Implementando conexões SQL resilientes do Entity Framework Core

Use o `IHttpClientFactory` para implementar solicitações HTTP resilientes

Implementar repetições de chamadas HTTP com retirada exponencial com a Polly

Implementar o padrão de disjuntor

Monitoramento de integridade

Proteger microsserviços .NET e aplicativos Web

Sobre a autorização em aplicativos Web e em microsserviços .NET

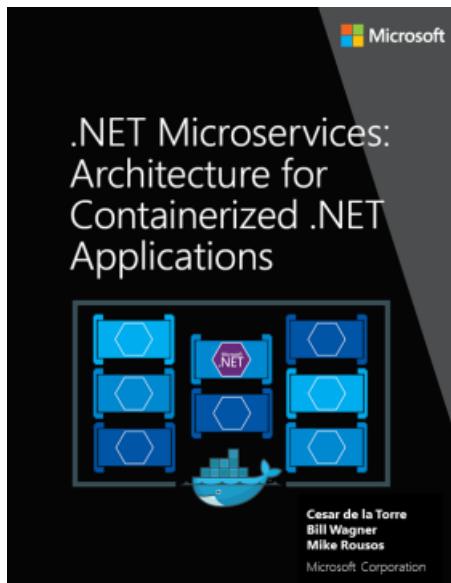
Armazenando segredos de aplicativo com segurança durante o desenvolvimento

Usando o Azure Key Vault para proteger segredos no momento da produção

Principais aspectos a serem lembrados

# Microsserviços do .NET: Arquitetura de aplicativos .NET em contêineres

10/09/2020 • 14 minutes to read • [Edit Online](#)



**Edição v 3.1.2** -atualizado para ASP.NET Core 3,1

Este guia é uma introdução ao desenvolvimento de aplicativos com base em microsserviços e ao gerenciamento deles usando contêineres. Ele discute as abordagens de design de arquitetura e de implementação usando o .NET Core e os contêineres do Docker.

Para facilitar a introdução, o guia concentra-se em um aplicativo de referência em contêineres e baseado em microsserviços que você pode explorar. O aplicativo de referência está disponível no repositório GitHub [eShopOnContainers](#).

## Links de ação

- Este livro eletrônico também está disponível em um formato PDF (somente versão em inglês) [Download](#)
- Clone/Crie fork do aplicativo de referência [eShopOnContainers no GitHub](#)
- Assista ao [vídeo introdutório no Channel 9](#)
- Conheça a [Arquitetura de Microsserviços](#) agora mesmo

## Introdução

As empresas estão cada vez mais percebendo a economia de custo, resolvendo problemas de implantação e melhorando as operações de produção e de DevOps usando os contêineres. A Microsoft tem lançando inovações de contêiner para Windows e Linux com a criação de produtos como o Serviço de Kubernetes do Azure e o Azure Service Fabric e por meio de parcerias com líderes do setor como a Docker, a Mesosphere e a Kubernetes. Esses produtos oferecem soluções de contêiner que ajudam as empresas a criar e implantar aplicativos com a velocidade e a escala da nuvem, seja qual for a escolha de plataformas ou de ferramentas.

O Docker está se tornando o verdadeiro padrão no setor de contêineres, com suporte dos fornecedores mais significativos nos ecossistemas do Windows e do Linux. (A Microsoft é um dos principais fornecedores de nuvem

que suportam o Docker.) No futuro, o Docker provavelmente estará onipresente em qualquer datacenter na nuvem ou no local.

Além disso, a arquitetura de [microsserviços](#) está despontando como uma abordagem importante para aplicativos críticos distribuídos. Em uma arquitetura baseada em microsserviço, o aplicativo é criado em uma coleção de serviços que podem ser desenvolvidos, testados, implantados e ter as versões controladas de forma independente.

## Sobre este guia

Este guia é uma introdução ao desenvolvimento de aplicativos com base em microsserviços e ao gerenciamento deles usando contêineres. Ele discute as abordagens de design de arquitetura e de implementação usando o .NET Core e os contêineres do Docker. Para facilitar a introdução aos contêineres e microsserviços, o guia concentra-se em um aplicativo de referência em contêineres e baseado em microsserviços que você pode explorar. O aplicativo de exemplo está disponível no repositório [eShopOnContainers](#) do GitHub.

Este guia fornece diretrizes básicas de desenvolvimento e de arquitetura principalmente no nível do ambiente de desenvolvimento com foco em duas tecnologias: Docker e .NET Core. Nossa intenção é que você leia este guia, ao pensar sobre o design do aplicativo sem focar a infraestrutura (nuvem ou local) do ambiente de produção. Você tomará decisões sobre a infraestrutura mais tarde, quando criar seus aplicativos prontos para produção. Portanto, este guia tem a intenção de ser independente da infraestrutura e mais centrado no ambiente de desenvolvimento.

Depois de estudar este guia, a próxima etapa será saber mais sobre os microsserviços pronto para produção no Microsoft Azure.

## Versão

Este guia foi revisado para cobrir a versão 3,1 do .NET Core junto com muitas atualizações adicionais relacionadas à mesma "onda" de tecnologias (isto é, Azure e tecnologias de terceiros adicionais) que coincidem no tempo com a versão 3,1 do .NET Core. É por isso que a versão do livro também foi atualizada para a versão 3,1.

## O que este guia não cobre

Este guia não se concentra no ciclo de vida do aplicativo, em DevOps, nos pipelines de CI/CD nem no trabalho da equipe. O guia complementar [Containerized Docker Application Lifecycle with Microsoft Platform and Tools](#) (Ciclo de vida de aplicativo do Docker em contêineres com a plataforma e as ferramentas da Microsoft) trata desse assunto. O guia atual também não fornece detalhes de implementação na infraestrutura do Azure, como informações sobre orquestradores específicos.

### Recursos adicionais

- **Ciclo de vida do aplicativo do Docker em contêineres com a plataforma e as ferramentas da Microsoft** (livro eletrônico para download)  
<https://aka.ms/dockerlifecyclebook>

## Quem deve usar este guia

Escrevemos este guia para desenvolvedores e arquitetos de solução que são novos no desenvolvimento de aplicativos com base no Docker e em arquitetura baseada em microsserviços. Este guia será indicado para você se desejar saber como arquitetar, projetar e implementar aplicativos de prova de conceito com tecnologias de desenvolvimento da Microsoft (com foco especial no .NET Core) e com contêineres do Docker.

Esse guia também será útil se você for um tomador de decisões técnicas, como um arquiteto corporativo que deseja ter uma visão geral da arquitetura e da tecnologia antes de decidir qual abordagem selecionar para aplicativos distribuídos novos e modernos.

### Como usar este guia

A primeira parte deste guia apresenta os contêineres do Docker, discute como escolher entre o .NET Core e o .NET Framework como estrutura de desenvolvimento e fornece uma visão geral sobre microsserviços. Este conteúdo é para arquitetos e tomadores de decisões técnicas que desejam ter uma visão geral, mas que não precisam se concentrar nos detalhes da implementação de código.

A segunda parte do guia de começa com a seção [Processo de desenvolvimento de aplicativos baseados no Docker](#). Ele se concentra nos padrões de desenvolvimento e de microsserviço para implementar aplicativos usando o .NET Core e o Docker. Esta seção será de interesse especial para desenvolvedores e arquitetos que desejam se concentrar no código, nos padrões e nos detalhes de implementação.

## Aplicativo de referência baseado em contêiner e em microsserviço: eShopOnContainers

O aplicativo eShopOnContainers é um aplicativo de referência de software livre para .NET Core e microsserviços criado para ser implantado usando contêineres do Docker. O aplicativo consiste em vários subsistemas, incluindo vários front-ends de interface do usuário de e-Store (um aplicativo Web MVC, um SPA da Web e um aplicativo móvel nativo). Ele também inclui os microsserviços e os contêineres de back-end de todas as operações do servidor necessárias.

A finalidade do aplicativo é demonstrar padrões de arquitetura. **A TI NÃO É UM MODELO PRONTO PARA PRODUÇÃO** para iniciar aplicativos de mundo real. Na verdade, o aplicativo está em um estado beta permanente, pois ele também é usado para testar novas tecnologias potencialmente interessantes conforme elas aparecem.

## Envie-nos seus comentários!

Escrevemos este guia para ajudá-lo a entender a arquitetura de aplicativos em contêineres e de microsserviços no .NET. O guia e o aplicativo de referência relacionado continuarão sendo desenvolvidos, portanto seus comentários são bem-vindos! Se você tiver comentários sobre como esse guia pode ser melhorado, envie comentários em <https://aka.ms/ebookfeedback>.

## Credits

Coautores:

**Cesar de la Torre**, Gerente sênior de produtos , equipe de produto do .NET, Microsoft Corp.

**Bill Wagner**, Desenvolvedor sênior de conteúdo, C+E, Microsoft Corp.

**Mike Rousos**, Engenheiro de Software Principal, equipe DevDiv CAT, Microsoft

Editores:

**Mike Pope**

**Steve Hoag**

Participantes e revisores:

**Jeffrey Richter**, engenheiro de software parceiro, equipe do Azure, Microsoft

**Jimmy Bogard**, Arquiteto Chefe na Headspring

**Udi Dahan**, Fundador e CEO, Particular Software

**Jimmy Nilsson**, Cofundador e CEO da Factor10

**Glenn Condron**, Gerente sênior de programas, equipe do ASP.NET

**Mark Fussell**, PM Líder Principal, equipe do Azure Service Fabric, Microsoft

**Diego Vega**, PM Líder, equipe do Entity Framework, Microsoft

**Barry Dorrans**, gerente de programas de segurança sênior

**Rowan Miller**, Gerente sênior de programas, Microsoft

**Ankit Asthana**, Gerente de PM Principal, equipe do .NET, Microsoft

**Scott Hunter**, PM Diretor de Parceiro, equipe do .NET, Microsoft

**Nish Anil**, Gerente de Programa Sênior, Equipe .NET, Microsoft

**Dylan Reisenberger**, Arquiteto e Desenvolvedor Líder na Polly

**Steve "ardalis" Smith** – Arquiteto de software e instrutor – [Ardalis.com](http://Ardalis.com)

**Cooper Ian**, Arquiteto de Codificação na Brighter

**Unai Zorrilla**, Arquiteto e Desenvolvedor Líder na Plain Concepts

**Eduard Tomas**, Desenvolvedor Líder na Plain Concepts

**Ramon Tomas**, Desenvolvedor na Plain Concepts

**David Sanz**, Desenvolvedor na Plain Concepts

**Javier Valero**, Diretor Executivo de Operações no Grupo Solutio

**Pierre Millet**, Consultor sênior, Microsoft

**Michael Friis**, Gerente de Produto, Docker Inc

**Charles Lowell**, Engenheiro de Software, equipe do VS CAT, Microsoft

**Miguel Veloso**, engenheiro de desenvolvimento de software em conceitos simples

**Pedro Ghosh**, consultor principal da Neudesic

## Direitos autorais

PUBLICADO POR

Divisão de Desenvolvedores Microsoft, equipes dos produtos .NET e Visual Studio

Uma divisão da Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2020 da Microsoft Corporation

Todos os direitos reservados. Nenhuma parte do conteúdo deste guia pode ser reproduzida ou transmitida de nenhuma forma nem por nenhum meio sem a permissão por escrito do publicador.

Este livro é fornecido “no estado em que se encontra” e expressa os pontos de vista e as opiniões do autor. Os pontos de vista, as opiniões e as informações expressos neste guia, incluindo URLs e outras referências a sites da Internet, podem ser alteradas sem aviso prévio.

Alguns exemplos aqui representados são fornecidos somente para fins de ilustração e são fictícios. Nenhuma

associação ou conexão real é intencional ou deve ser inferida.

A Microsoft e as marcas listadas em <https://www.microsoft.com> na página da Web "Marcas" são marcas comerciais do grupo de empresas Microsoft.

Mac e macOS são marcas comerciais da Apple Inc.

O logotipo de redistribuição do Docker é uma marca registrada do Docker, Inc. usada pela permissão.

Todas as outras marcas e logotipos são propriedade de seus respectivos proprietários.

PRÓXIMO

# Introdução aos contêineres e ao Docker

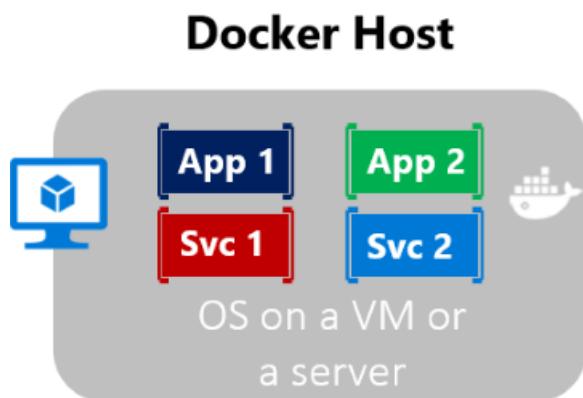
07/04/2020 • 3 minutes to read • [Edit Online](#)

O uso de contêineres é uma abordagem de desenvolvimento de software na qual um aplicativo ou serviço, suas dependências e sua configuração (abstraídos como arquivos de manifesto de implantação) são empacotados juntos como uma imagem de contêiner. O aplicativo em contêineres pode ser testado como uma unidade e implantado como uma instância de imagem de contêiner no SO (sistema operacional) do host.

Assim como os contêineres de remessa permitem que as mercadorias sejam transportadas por navio, trem ou caminhão, independentemente da carga que contenham, os contêineres de implantação de software agem como uma unidade de software padrão que pode conter diferentes dependências e códigos. Inserir o software em contêineres dessa maneira permite que desenvolvedores e profissionais de TI os implantem em diferentes ambientes com pouca ou sem nenhuma modificação.

Os contêineres também isolam os aplicativos uns dos outros em um sistema operacional compartilhado. Os aplicativos em contêineres são executados com base em um host do contêiner que por sua vez é executado no sistema operacional (Linux ou Windows). Os contêineres, portanto, ocupam um espaço significativamente menor do que as imagens de VM (máquina virtual).

Cada contêiner pode executar um aplicativo Web ou um serviço inteiro, conforme é mostrado na Figura 2-1. Neste exemplo, o host do Docker é um host de contêiner e App1, App2, Svc 1 e Svc 2 são aplicativos ou serviços em contêineres.



**Figura 2-1.** Vários contêineres em execução em um host de contêiner

Outro benefício do uso de contêineres é a escalabilidade. Você pode aumentar rapidamente, criando novos contêineres para tarefas de curto prazo. Do ponto de vista do aplicativo, criar uma instância de uma imagem (criar um contêiner) é semelhante a criar uma instância de um processo, como um serviço ou aplicativo Web. No entanto, para assegurar a confiabilidade, ao executar várias instâncias da mesma imagem em vários servidores host, geralmente é melhor que cada contêiner (instância da imagem) seja executado em um servidor host ou em uma VM diferente em domínios de falha diferentes.

Resumindo, os contêineres oferecem os benefícios de portabilidade, agilidade, escalabilidade, controle e isolamento em todo o fluxo de trabalho do ciclo de vida do aplicativo. O benefício mais importante é o isolamento de ambiente fornecido entre Desenvolvimento e Operações.

# O que é o Docker?

18/03/2020 • 11 minutes to read • [Edit Online](#)

O Docker é um [projeto de software livre](#) para automatizar a implantação de aplicativos como contêineres autossuficientes portáteis que podem ser executados na nuvem ou localmente. O Docker é também uma [empresa](#) que promove e aprimora essa tecnologia, trabalhando em colaboração com fornecedores de nuvem, do Linux e do Windows, incluindo a Microsoft.

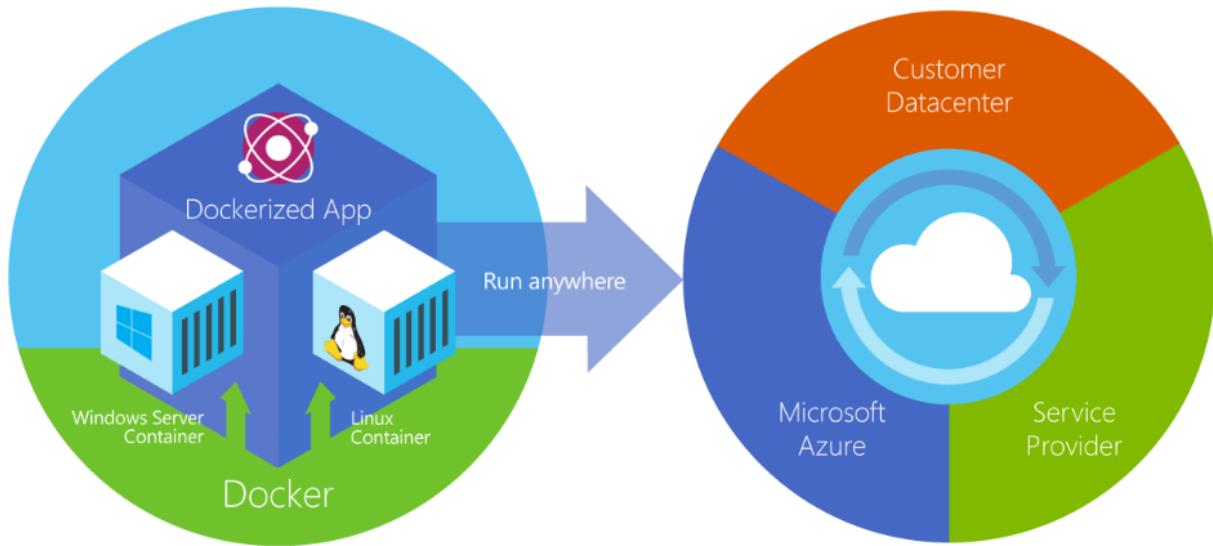


Figura 2-2. Docker implanta contêineres em todas as camadas da nuvem híbrida.

Contêineres do Docker podem executar em qualquer lugar, localmente no datacenter do cliente, em um provedor de serviços externo ou na nuvem, no Azure. Os contêineres de imagem do Docker podem ser executados nativamente no Linux e no Windows. No entanto, imagens do Windows podem executar somente em hosts do Windows e imagens do Linux podem executar em hosts do Linux e hosts do Windows (usando uma VM do Linux do Hyper-V, até o momento), em que o host significa um servidor ou uma VM.

Os desenvolvedores podem usar ambientes de desenvolvimento no Windows, Linux ou macOS. No computador de desenvolvimento, o desenvolvedor executa um host Docker em que as imagens do Docker são implantadas, incluindo o aplicativo e suas dependências. Desenvolvedores que trabalham no Linux ou no macOS usam um host Docker baseado em Linux, e eles podem criar imagens apenas para contêineres Linux. (Os desenvolvedores que trabalham no macOS podem editar código ou executar o Cli Docker a partir do macOS, mas a partir do momento desta escrita, os contêineres não são executados diretamente no macOS.) Desenvolvedores que trabalham no Windows podem criar imagens para O Linux ou Windows Containers.

Para hospedar contêineres em ambientes de desenvolvimento e fornecer ferramentas para desenvolvedores adicionais, o Docker envia o [Docker Community Edition \(CE\)](#) para Windows ou para o macOS. Esses produtos instalam a VM necessária (o host do Docker) para hospedar os contêineres. O Docker também disponibiliza o [Docker Enterprise Edition \(EE\)](#), que foi projetado para desenvolvimento empresarial e é usado por equipes de TI que criam, enviam e executam aplicativos de grande porte críticos para os negócios em produção.

Para executar [contêineres do Windows](#), há dois tipos de runtimes:

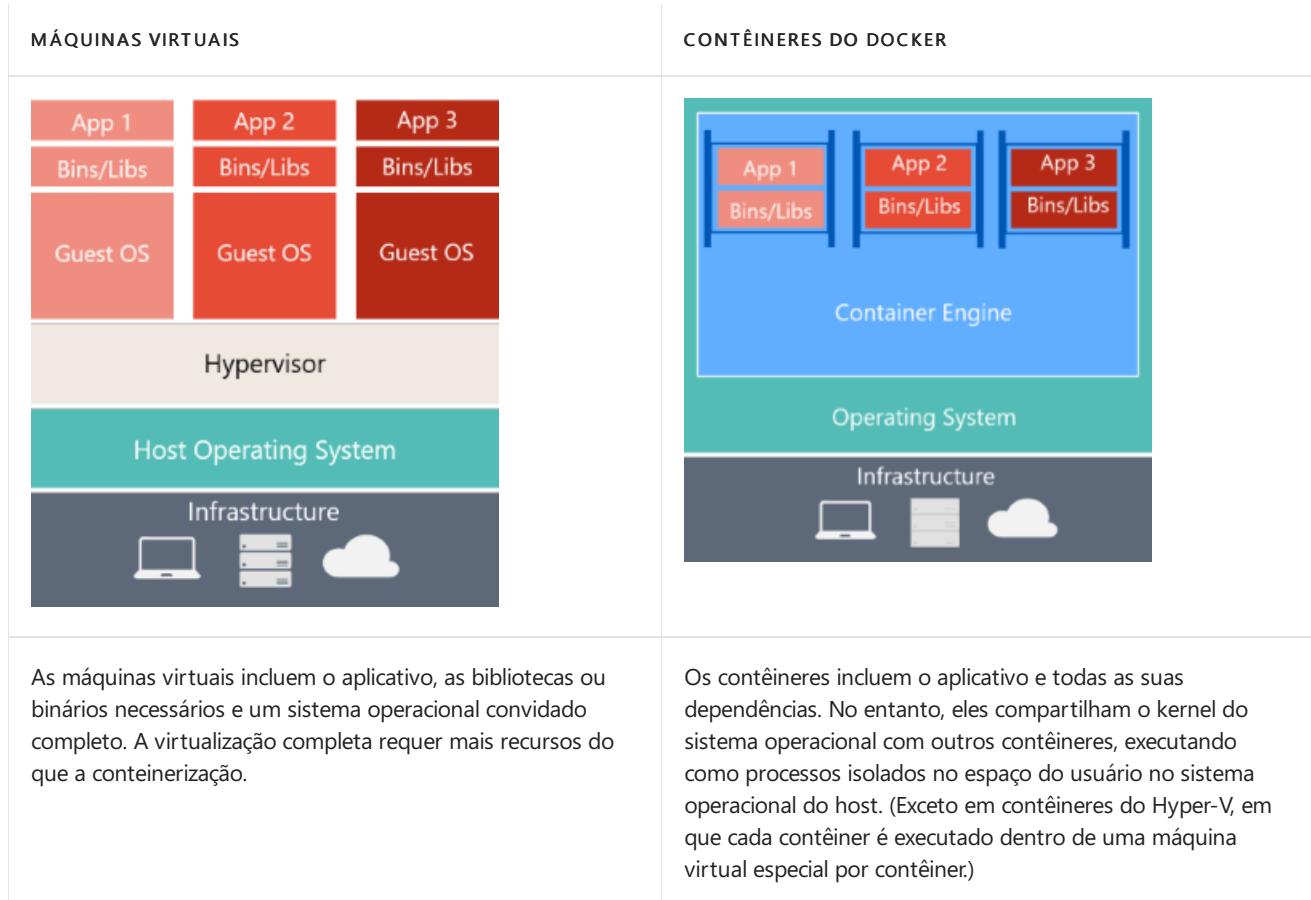
- Os contêineres do Windows Server fornecem isolamento de aplicativos por meio da tecnologia de isolamento de processo e de namespace. Um contêiner do Windows Server compartilha um kernel com o host do contêiner e todos os contêineres em execução no host.

- Os contêineres do Hyper-V expandem o isolamento fornecido pelos contêineres do Windows Server, executando cada contêiner em uma máquina virtual altamente otimizada. Nessa configuração, o kernel do host do contêiner não é compartilhado com os contêineres do Hyper-V, fornecendo melhor isolamento.

As imagens desses contêineres são criadas e funcionam da mesma maneira. A diferença está em como o contêiner é criado da imagem, executar um contêiner do Hyper-V exige um parâmetro extra. Para obter detalhes, consulte [Contêineres do Hyper-V](#).

## Comparando os contêineres do Docker com máquinas virtuais

A Figura 2-3 mostra uma comparação entre VMs e contêineres do Docker.



**Figura 2-3.** Comparação de máquinas virtuais tradicionais com contêineres do Docker

Para VMs, há três camadas de base no servidor de host, de cima para baixo: infraestrutura, sistema operacional do host e um hipervisor e, acima disso tudo, cada VM tem seu próprio sistema operacional e todas as bibliotecas necessárias. Para o Docker, o servidor de host tem apenas a infraestrutura e o sistema operacional e, acima disso tudo, o mecanismo de contêiner que mantém contêiner isolado mas compartilhando os serviços do sistema operacional base.

Como os contêineres requerem muito menos recursos (por exemplo, eles não precisam de um sistema operacional completo), eles iniciam rapidamente e são fáceis de implantar. Isso permite que você tenha maior densidade, o que significa que permite a você executar mais serviços na mesma unidade de hardware, reduzindo os custos.

Como um efeito colateral da execução no mesmo kernel, você obtém menos isolamento do que em VMs.

A meta principal de uma imagem é que ela torne o ambiente (dependências) o mesmo entre diferentes implantações. Isso significa que é possível depurá-la em seu computador e, em seguida, implantá-la em outro computador com o mesmo ambiente garantido.

Uma imagem de contêiner é uma maneira de empacotar um aplicativo ou serviço e implantá-lo de maneira

confiável e reproduzível. Seria possível dizer que o Docker não é apenas uma tecnologia, mas também uma filosofia e um processo.

Ao usar o Docker, você não escutará os desenvolvedores dizerem "Funciona no meu computador, por que não em produção?" Eles podem simplesmente dizer: "É executado no Docker", porque o aplicativo empacotado do Docker pode ser executado em qualquer ambiente do Docker compatível, sendo executado da maneira pretendida em todos os destinos de implantação (tais como desenvolvimento, garantia de qualidade, preparação e produção).

## Uma analogia simples

Talvez uma analogia simples possa ajudar a obter o entendimento do conceito principal do Docker.

Vamos voltar no tempo à década de 1950 por um momento. Não havia nenhum processador de palavras e as fotocopiadoras eram usadas em toda parte (ou quase).

Imagine que você seja responsável por emitir lotes de cartas rapidamente conforme necessário, enviá-las aos clientes usando papel e envelopes de verdade, a serem entregues fisicamente ao endereço de cada cliente (não havia email naquela época).

Em algum momento, você percebe que as cartas são apenas uma composição de um grande conjunto de parágrafos, que são separados e organizados conforme necessário de acordo com a finalidade da carta, então você elabora um sistema para emitir cartas rapidamente, esperando receber um aumento substancial.

O sistema é simples:

1. Você começa com um maço de folhas transparentes, que contém um parágrafo cada.
2. Para emitir um conjunto de cartas, você escolhe as planilhas com os parágrafos que precisa e, em seguida, empilha-as e alinha-as para que elas possam ser lidas corretamente e fiquem aparência organizada.
3. Por fim, você coloca o conjunto na fotocopiadora e pressiona o botão de início para produzir tantas cartas quantas forem necessárias.

Portanto, simplificando, essa é a ideia central do Docker.

No Docker, cada camada é o conjunto resultante de alterações que ocorrem no sistema de arquivos depois de executar um comando, tal como instalar um programa.

Assim, quando você "olha" para o sistema de arquivos depois que a camada é copiada, você vê todos os arquivos, incluindo a camada quando o programa foi instalado.

Você pode pensar uma imagem como um auxiliar somente leitura disco rígido pronto para ser instalado em um "computador" em que o sistema operacional já está instalado.

Da mesma forma, você pode pensar em um contêiner como o "computador" com o disco rígido de imagem instalado. O contêiner, assim como um computador, pode ser ativado ou desativado.

[PRÓXIMO](#)

[ANTERIOR](#)

# Terminologia do Docker

28/04/2020 • 10 minutes to read • [Edit Online](#)

Esta seção lista os termos e definições que você deve conhecer antes de se aprofundar no Docker. Para obter mais definições, consulte o amplo [glossário](#) fornecido pelo Docker.

**Imagen de contêiner:** um pacote com todas as dependências e informações necessárias para criar um contêiner. Uma imagem inclui todas as dependências (como estruturas), além da configuração de implantação e execução a ser usada por um runtime de contêiner. Geralmente, uma imagem deriva de várias imagens base que são camadas empilhadas umas sobre as outras para formar o sistema de arquivos do contêiner. Uma imagem é imutável depois de ser criada.

**Dockerfile:** um arquivo de texto que contém instruções para criar uma imagem do Docker. É como um script em lotes, a primeira linha declara a imagem base com a qual começar e, em seguida, siga as instruções para instalar os programas necessários, copiar os arquivos e assim por diante, até obter o ambiente de trabalho que precisa.

**Build:** a ação de criar de uma imagem de contêiner com base nas informações e no contexto fornecido pelo Dockerfile, além de arquivos adicionais na pasta em que a imagem é criada. Você pode criar imagens com o seguinte comando do Docker:

```
docker build
```

**Contêiner:** uma instância de uma imagem do Docker. Um contêiner representa a execução de um único aplicativo, processo ou serviço. Consiste no conteúdo de uma imagem do Docker, um ambiente de execução e um conjunto padrão de instruções. Ao dimensionar um serviço, você cria várias instâncias de um contêiner da mesma imagem. Ou um trabalho em lotes pode criar vários contêineres da mesma imagem, passando parâmetros diferentes para cada instância.

**Volumes:** oferecem um sistema de arquivos gravável que o contêiner pode usar. Uma vez que as imagens são somente leitura, mas a maioria dos programas precisam gravar para o sistema de arquivos, os volumes adicionam uma camada gravável sobre a imagem de contêiner, de modo que os programas têm acesso a um sistema de arquivos gravável. O programa não sabe que está acessando um sistema de arquivos em camadas, é apenas o sistema de arquivos como de costume. Os volumes ficam no sistema de host e são gerenciados pelo Docker.

**Marcação:** uma marca ou um rótulo pode ser aplicado a imagens para que imagens ou versões diferentes da mesma imagem (dependendo do número de versão ou do ambiente de destino) possam ser identificadas.

**Build de vários estágios:** é um recurso disponível no Docker 17.05 e posteriores que ajuda a reduzir o tamanho das imagens finais. Em algumas frases, com a compilação de vários estágios, você pode usar, por exemplo, uma grande imagem de base, que contém o SDK, para compilar e publicar o aplicativo e, em seguida, usar a pasta de publicação com uma única imagem base somente de tempo de execução, para produzir uma imagem final muito menor.

**Repositório:** uma coleção de imagens do Docker relacionadas, rotulada com uma marcação que indica a versão da imagem. Alguns repositórios contêm várias variantes de uma imagem específica, como uma imagem que contém SDKs (mais pesado), uma imagem contendo apenas tempos de execução (mais leves), etc. Essas variantes podem ser marcadas com marcas. Um único repositório pode conter variantes de plataforma, como uma imagem do Linux e uma imagem do Windows.

**Registro:** um serviço que dá acesso aos repositórios. O registro padrão para as imagens mais públicas é o [Docker Hub](#) (propriedade da Docker como uma organização). Um registro geralmente contém repositórios de várias equipes. As empresas geralmente têm registros privados para armazenar e gerenciar as imagens que criaram. O

Registro de Contêiner do Azure é outro exemplo.

**Imagen de vários arcos:** para várias arquiteturas, trata-se de um recurso que simplifica a seleção da imagem apropriada, de acordo com a plataforma em que o Docker está em execução. Por exemplo, quando um Dockerfile solicita uma imagem base [do MCR.Microsoft.com/dotnet/Core/SDK:3.1](https://mcr.microsoft.com/dotnet/Core/SDK:3.1) a partir do registro, ele realmente obtém `3.1-SDK-los-1909`, `3.1-sdk-los Server-1809` ou `3.1-SDK-Buster-Slim`, dependendo do sistema operacional e da versão em que o Docker está em execução.

**Docker Hub:** um registro público para carregar imagens e trabalhar com elas. O Docker Hub hospeda imagens do Docker, registros públicos ou privados, cria gatilhos e ganchos da Web e integra-se com o GitHub e o Bitbucket.

**Registro de Contêiner do Azure:** um recurso público para trabalhar com imagens do Docker e seus componentes no Azure. Fornece um registro que está perto de suas implantações no Azure e que permite controlar o acesso, tornando possível usar as permissões e os grupos do Azure Active Directory.

**Docker Trusted Registry (DTR):** serviço de Registro do Docker que pode ser instalado localmente para funcionar no datacenter e na rede da organização. É conveniente para imagens privadas que devem ser gerenciadas dentro da empresa. O Docker Trusted Registry é parte do produto Docker Datacenter. Para saber mais, consulte [Docker Trusted Registry \(DTR\)](#).

**Docker Community Edition (CE):** ferramentas de desenvolvimento para Windows e macOS para build, execução e teste de contêineres localmente. O Docker CE para Windows fornece os ambientes de desenvolvimento para Linux e contêineres do Windows. O host do Docker do Linux no Windows é baseado em uma máquina virtual [Hyper-V](#). O host para contêineres do Windows se baseia diretamente no Windows. Docker CE para Mac baseia-se na estrutura do Apple Hypervisor e o [xhyve hypervisor](#), que fornece uma máquina virtual do host Linux Docker no Mac OS X. O Docker CE para Windows e Mac substitui o Docker Toolbox, que foi baseado no Oracle VirtualBox.

**Docker Enterprise Edition (EE):** uma versão empresarial das ferramentas do Docker para desenvolvimento em Linux e Windows.

**Compose:** uma ferramenta de linha de comando e formato de arquivo YAML com metadados para definir e executar aplicativos de vários contêineres. Você define um único aplicativo com base em várias imagens com um ou mais arquivos `.yml` que podem substituir valores dependendo do ambiente. Depois de criar as definições, você pode implantar todo o aplicativo de vários contêineres com um único comando (`docker-compose up`), que cria um contêiner por imagem no host do Docker.

**Cluster:** uma coleção de hosts do Docker expostos como um único host virtual, para que o aplicativo possa ser dimensionado para várias instâncias dos serviços distribuídos em vários hosts do cluster. Os clusters do Docker podem ser criados com o Kubernetes, o Azure Service Fabric, o Docker Swarm e o Mesosphere DC/OS.

**Orquestrador:** uma ferramenta que simplifica o gerenciamento de clusters e hosts do Docker. Os orquestradores permitem gerenciar imagens, contêineres e hosts por meio de uma CLI (interface de linha de comando) ou uma interface do usuário gráfica. É possível gerenciar a rede de contêiner, configurações, balanceamento de carga, descoberta de serviço, alta disponibilidade, configuração de host do Docker e muito mais. Um orquestrador é responsável por executar, distribuir, dimensionar e reparar de cargas de trabalho em uma coleção de nós. Normalmente, produtos de orquestrador são os mesmos que fornecem infraestrutura de cluster, como Kubernetes e Azure Service Fabric, além de outras ofertas no mercado.

# Registros, imagens e contêineres do Docker

18/03/2020 • 3 minutes to read • [Edit Online](#)

Ao usar o Docker, um desenvolvedor cria um aplicativo ou serviço e empacota a ele e suas dependências em uma imagem de contêiner. Uma imagem é uma representação estática do aplicativo ou do serviço e de sua configuração e dependências.

Para executar o aplicativo ou o serviço, uma instância da imagem do aplicativo é criada para criar um contêiner, que estará em execução no host do Docker. Inicialmente, os contêineres são testados em um ambiente de desenvolvimento ou em um computador.

Os desenvolvedores devem armazenar imagens em um Registro, que funciona como uma biblioteca de imagens e é necessário ao implantar em orquestradores de produção. O Docker mantém um Registro público por meio do [Hub do Docker](#); outros fornecedores fornecem os Registros para diferentes coleções de imagens, incluindo [Registro de Contêiner do Azure](#). Como alternativa, as empresas podem ter um Registro privado local para suas próprias imagens do Docker.

A Figura 2-4 mostra como imagens e Registros no Docker se relacionam com outros componentes. Ela também mostra as várias ofertas de Registro dos fornecedores.

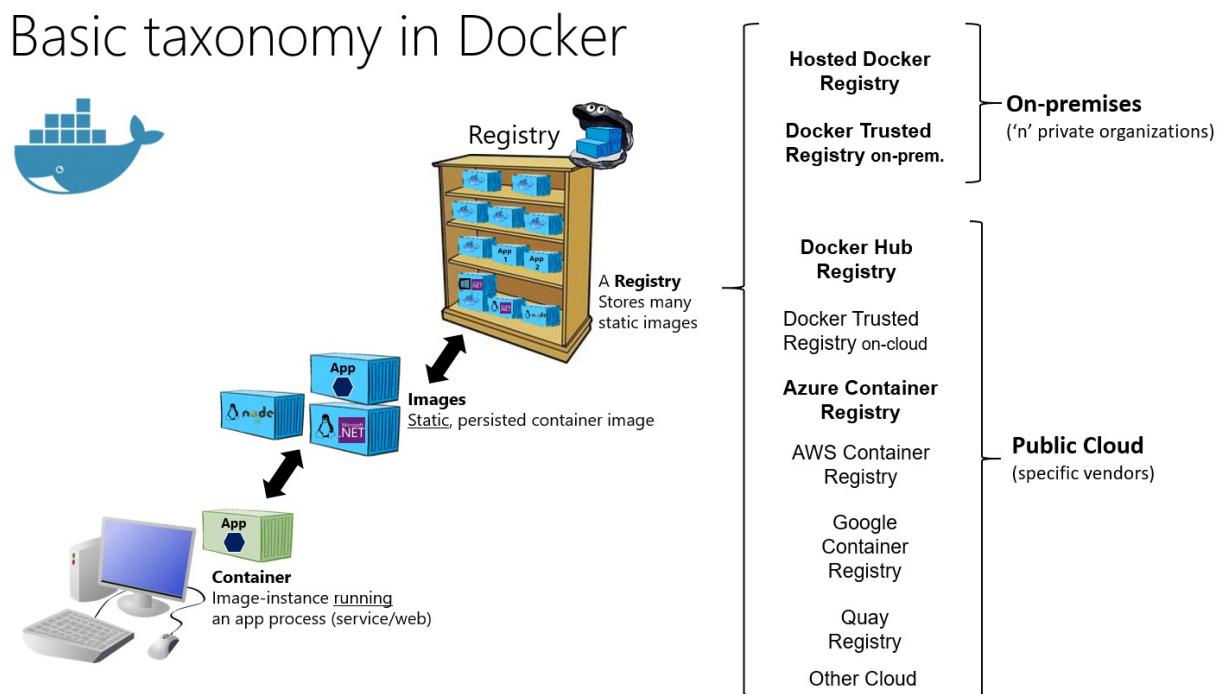


Figura 2-4. Taxonomia de termos e conceitos do Docker

o Registro é como uma estante de livros em que as imagens são armazenadas e ficam disponíveis para serem retiradas para a criação de contêineres para executar os serviços ou aplicativos Web. Há registros do Docker privados locais e na nuvem pública. O Hub do Docker é um registro público mantido pelo Docker, junto com o Registro Confiável do Docker, uma solução empresarial, o Azure oferece ao Registro de Contêiner do Azure. AWS, Google e outros também têm Registros de contêiner.

Colocar imagens em um Registro permite a você armazenar os bits de aplicativo estáticos e imutáveis, incluindo todas as suas dependências em um nível de estrutura. A versão dessas imagens podem ser controladas e elas podem ser implantadas em vários ambientes e, portanto, fornecem uma unidade de implantação consistente.

Registros de imagem privados, hospedados localmente ou na nuvem, são recomendados quando:

- Suas imagens não devem ser compartilhadas publicamente devido à confidencialidade.
- Você deseja ter latência de rede mínima entre suas imagens e o ambiente de implantação escolhido. Por exemplo, se o ambiente de produção for uma nuvem do Azure, você provavelmente desejará armazenar as imagens no [Registro de Contêiner do Azure](#) para que a latência de rede seja mínima. De maneira semelhante, se seu ambiente de produção for local, tenha um Registro Confiável do Docker local disponível na mesma rede local.

[PRÓXIMO](#)

[ANTERIOR](#)

# Escolhendo entre o .NET Core e do .NET Framework para contêineres do Docker

18/03/2020 • 2 minutes to read • [Edit Online](#)

Há suporte para duas estruturas para criar aplicativos do Docker em contêineres no lado do servidor com o .NET: [.NET Framework](#) e [.NET Core](#). Eles compartilham muitos componentes da plataforma .NET e você pode compartilhar o código entre os dois. No entanto, há diferenças fundamentais entre eles e a estrutura a ser usada dependerá do que você deseja realizar. Esta seção fornece orientações sobre quando escolher cada estrutura.

[PRÓXIMO](#)

[ANTERIOR](#)

# Orientação geral

10/09/2020 • 3 minutes to read • [Edit Online](#)

Esta seção oferece um resumo de quando escolher o .NET Core ou o .NET Framework. Fornecemos mais detalhes sobre essas opções nas seções a seguir.

Use o .NET Core, com contêineres do Linux ou do Windows, para seu aplicativo de servidor Docker em contêiner quando:

- Você tiver necessidades de plataforma cruzada. Por exemplo, você desejar usar contêineres Linux e do Windows.
- Sua arquitetura de aplicativo for baseada em microsserviços.
- For necessário iniciar contêineres rapidamente e você desejar ocupar um espaço menor por contêiner para obter melhor densidade ou mais contêineres por unidade de hardware a fim de reduzir seus custos.

Em resumo, quando você cria novos aplicativos .NET em contêineres, você deve considerar o .NET Core como a opção padrão. Ela tem muitos benefícios e se ajusta melhor à filosofia e ao estilo de trabalho dos contêineres.

Um benefício adicional de usar o .NET Core é que você pode executar versões do .NET para aplicativos lado a lado dentro do mesmo computador. Esse benefício é mais importante para servidores ou VMs que não usam contêineres, porque os contêineres isolam as versões do .NET de que o aplicativo precisa. (Desde que sejam compatíveis com o sistema operacional subjacente.)

Use .NET Framework para seu aplicativo de servidor Docker em contêiner quando:

- No momento, seu aplicativo usar o .NET Framework e tem fortes dependências no Windows.
- For necessário usar APIs do Windows não compatíveis com o .NET Core.
- For necessário usar bibliotecas .NET de terceiros ou pacotes NuGet não disponíveis para o .NET Core.

Usar o .NET Framework no Docker pode melhorar suas experiências de implantação minimizando os problemas de implantação. Este [cenário de "lift-and-shift"](#) é importante para colocar aplicativos herdados em contêineres que foram desenvolvidos originalmente com o .NET Framework tradicional, como os serviços ASP.NET WebForms, aplicativos Web MVC ou WCF (Windows Communication Foundation).

## Recursos adicionais

- **Livro eletrônico: modernizar os aplicativos .NET Framework existentes com contêineres do Azure e do Windows**  
<https://aka.ms/liftandshiftwithcontainersebook>
- **Aplicativos de exemplo: modernização de aplicativos Web ASP.NET herdados usando Contêineres do Windows**  
<https://aka.ms/eshopmodernizing>

[ANTERIOR](#)

[AVANÇAR](#)

# Quando escolher o .NET Core para os contêineres do Docker

30/04/2020 • 8 minutes to read • [Edit Online](#)

A modularidade e a natureza leve do .NET Core torna-o perfeito para contêineres. Ao implantar e iniciar um contêiner, sua imagem é muito menor com o .NET Core do que com o .NET Framework. Por outro lado, para usar o .NET Framework para um contêiner, é necessário basear sua imagem na imagem do Windows Server Core, que é muito mais pesada do que as imagens do Windows Nano Server ou Linux que você usa para o .NET Core.

Além disso, o .NET Core é multiplataforma, portanto, é possível implantar aplicativos de servidor com imagens de contêiner do Windows ou Linux. No entanto, se você estiver usando o .NET Framework tradicional, só poderá implantar imagens baseadas no Windows Server Core.

A seguir há uma explicação mais detalhada do porquê escolher o .NET Core.

## Desenvolvendo e implantando uma multiplataforma

Claramente, se sua meta for ter um aplicativo (serviço ou aplicativo Web) que possa ser executado em várias plataformas compatíveis com o Docker (Linux e Windows), a escolha correta será o .NET Core, porque o .NET Framework é compatível somente com o Windows.

O .NET Core também é compatível com o macOS como uma plataforma de desenvolvimento. No entanto, quando você implanta contêineres em um host do Docker, esse host deve (atualmente) ser baseado em Linux ou Windows. Por exemplo, em um ambiente de desenvolvimento, você pode usar uma VM Linux em execução em um Mac.

O [Visual Studio](#) fornece um IDE (ambiente de desenvolvimento integrado) para Windows e é compatível com o desenvolvimento do Docker.

O [Visual Studio para Mac](#) é um IDE, a evolução do Xamarin Studio, que é executada em macOS e dá suporte ao desenvolvimento de aplicativos baseados em Docker. Essa deve ser a opção preferencial para desenvolvedores que trabalham em computadores Mac que querem usar um IDE avançado.

Você também pode usar [Visual Studio Code](#) no MacOS, Linux e Windows. Visual Studio Code oferece suporte total ao .NET Core, incluindo IntelliSense e depuração. Como VS Code é um editor leve, você pode usá-lo para desenvolver aplicativos em contêineres no computador em conjunto com a CLI do Docker e o [CLI do .NET Core](#). Também é possível direcionar o .NET Core com a maioria dos editores de terceiros, como Sublime, Emacs, vi e o projeto OmniSharp de software livre, que também fornece suporte ao IntelliSense.

Além dos IDEs e editores, você pode usar o [CLI do .NET Core](#) para todas as plataformas com suporte.

## Usando contêineres para novos projetos ("campo verde")

Contêineres são comumente usados em conjunto com uma arquitetura de microserviços, embora também possam ser usados para colocar em contêineres aplicativos Web ou serviços que sigam qualquer padrão de arquitetura. É possível usar o .NET Framework em contêineres do Windows, mas a modularidade e a natureza leve do .NET Core torna-o perfeito para contêineres e arquiteturas de microserviços. Quando você cria e implanta um contêiner, sua imagem é muito menor com o .NET Core do que com o .NET Framework.

## Criar e implantar microserviços em contêineres

É possível usar o .NET Framework tradicional para criar aplicativos baseados em microserviços (sem contêineres)

usando processos simples. Dessa forma, como o .NET Framework já está instalado e já é compartilhado entre processos, os processos são leves inicializam rapidamente. No entanto, se você estiver usando contêineres, a imagem do .NET Framework tradicional também será baseada no Windows Server Core e isso a tornará muito pesada para uma abordagem de microsserviços em contêineres. No entanto, as equipes têm busca de oportunidades para melhorar a experiência de .NET Framework usuários também. Recentemente, o tamanho das [imagens de contêiner do Windows Server Core foi reduzido para >40% menor](#).

Por outro lado, o .NET Core é o melhor candidato se você estiver adotando um sistema orientado a microserviços baseado em contêineres, pois o .NET Core é leve. Além disso, suas imagens de contêiner relacionadas, para Linux ou Windows nano Server, são de baixo e pequeno, tornando os contêineres claros e rápidos para começar.

Um microsserviço deve ser o menor possível: ser leve ao iniciar, ocupar um pequeno espaço, ter um Contexto limitado pequeno (confira DDD, [Design Voltado a Domínio](#)), para representar uma pequena área de preocupações e ser capaz de ser iniciado e interrompido rapidamente. Para esses requisitos, você desejará usar imagens de contêiner pequenas e cuja instância é fácil de criar, como a imagem de contêiner do .NET Core.

Uma arquitetura de microsserviços também permite uma combinação de tecnologias em um limite de serviço. Isso permite uma migração gradual para o .NET Core para novos microsserviços que funcionam em conjunto com outros microsserviços ou com serviços desenvolvidos com o Node.js, Python, Java, GoLang ou outras tecnologias.

## Implantando alta densidade em sistemas escalonáveis

Quando seu sistema baseado em contêiner precisar da melhor densidade, granularidade e desempenho possíveis, o .NET Core e o ASP.NET Core são suas melhores opções. O ASP.NET Core é até dez vezes mais rápido do que o ASP.NET no .NET Framework tradicional e lidera outras tecnologias populares do setor para microsserviços, como Java servlets, Go e Node.js.

Isso é especialmente relevante para arquiteturas de microsserviços, em que você poderia ter centenas de microsserviços (contêineres) em execução. Com imagens do ASP.NET Core (baseadas no runtime do .NET Core) no Linux ou Windows Nano, é possível executar seu sistema com um número muito menor de servidores ou VMs, economizando custos em infraestrutura e hospedagem.

[ANTERIOR](#)

[PRÓXIMO](#)

# Quando escolher o .NET Framework para os contêineres do Docker

10/09/2020 • 7 minutes to read • [Edit Online](#)

Embora o .NET Core ofereça benefícios significativos para novos aplicativos e padrões de aplicativo, o .NET Framework continuará sendo uma boa escolha para muitos cenários existentes.

## Migrando aplicativos existentes diretamente para um contêiner do Windows Server

Talvez convenha usar contêineres do Docker apenas para simplificar a implantação, mesmo se você não estiver criando microsserviços. Por exemplo, talvez convenha aprimorar seu fluxo de trabalho DevOps com o Docker — os contêineres podem oferecer ambientes de teste mais bem isolados e também podem eliminar problemas de implantação causados por dependências ausentes quando você muda para um ambiente de produção. Em casos assim, mesmo se você estiver implantando um aplicativo monolítico, faz sentido usar os contêineres do Docker e do Windows para seus aplicativos .NET Framework atuais.

Na maioria dos casos desse cenário, não será necessário migrar seus aplicativos existentes para o .NET Core; é possível usar contêineres do Docker que incluem o .NET Framework tradicional. Contudo, uma abordagem recomendada é usar o .NET Core ao estender um aplicativo existente, por exemplo, para gravar um novo serviço no ASP.NET Core.

## Usando bibliotecas .NET de terceiros ou pacotes NuGet não disponíveis para o .NET Core

As bibliotecas de terceiros estão adotando rapidamente [.net Standard](#), o que permite o compartilhamento de código em todos os tipos .net, incluindo o .NET Core. Com o .NET Standard 2,0 e posterior, a compatibilidade da superfície de API em diferentes estruturas se tornou significativamente maior. Ainda mais, o .NET Core 2.x e os aplicativos mais recentes também podem fazer referência direta a bibliotecas de .NET Framework existentes (Confira [.NET Framework 4.6.1 com suporte a .NET Standard 2,0](#)).

Além disso, o [pacote de compatibilidade do Windows](#) estende a superfície de API disponível para .net Standard 2,0 no Windows. Este pacote permite recompilar a maioria do código existente para o .NET Standard 2.x com pouca ou nenhuma modificação para ser executado no Windows.

No entanto, mesmo com essa progressão excepcional desde o .NET Standard 2,0 e o .NET Core 2,1, pode haver casos em que determinados pacotes NuGet precisam do Windows para serem executados e não dão suporte ao .NET Core. Se esses pacotes forem críticos ao aplicativo, será necessário usar o .NET Framework em contêineres do Windows.

## Usando tecnologias do .NET não disponíveis para .NET Core

Algumas tecnologias .NET Framework não estão disponíveis na versão atual do .NET Core (versão 3,1 no momento da elaboração deste artigo). Alguns deles podem ser disponibilizados em versões posteriores, mas outros não se ajustam aos novos padrões de aplicativo destinados ao .NET Core e podem nunca estar disponíveis.

A lista a seguir mostra a maioria das tecnologias que não estão disponíveis no .NET Core 3,1:

- Web Forms do ASP.NET. Esta tecnologia só está disponível no .NET Framework. Atualmente, não há planos para trazer os Web Forms do ASP.NET para o .NET Core.

- Serviços WCF. Mesmo quando uma [biblioteca de cliente WCF](#) está disponível para consumir serviços WCF do .NET Core, a partir de fev-2020, a implementação do servidor WCF só está disponível no .NET Framework. Esse cenário pode ser considerado para versões futuras do .NET Core. Há, inclusive, algumas APIs consideradas para inclusão no [Pacote de Compatibilidade do Windows](#).
- Serviços relacionados a fluxo de trabalho. O Windows WF (Workflow Foundation), os Serviços de fluxo de trabalho (WCF + WF em um único serviço) e o WCF Data Services (anteriormente conhecido como Serviços de Dados ADO.NET) só estão disponíveis no .NET Framework. No momento, não há planos para colocá-los no .NET Core.

Além das tecnologias listadas no roteiro oficial do [.NET Core](#), outros recursos podem ser portados para o .NET Core ou para a nova [plataforma .net unificada](#). Você pode considerar participar das discussões no GitHub para que sua voz possa ser ouvida. E se você acreditar que algo está faltando, execute um novo problema no repositório do GitHub [dotnet/tempo de execução](#).

## Usando uma plataforma ou API que não dá suporte ao .NET Core

Algumas plataformas da Microsoft e de terceiros não oferecem suporte ao .NET Core. Por exemplo, alguns serviços do Azure fornecem um SDK que ainda não está disponível para consumo no .NET Core. A maioria dos SDK do Azure deve, eventualmente, ser portado para o .NET Core/Standard, mas alguns podem não por vários motivos. Você pode ver os SDKs do Azure disponíveis na página [versões mais recentes do SDK do Azure](#).

Enquanto isso, se qualquer plataforma ou o serviço no Azure ainda não der suporte ao .NET Core com sua API do cliente, será possível usar a API REST equivalente do serviço do Azure ou o SDK do cliente no .NET Framework.

### Recursos adicionais

- Conceitos básicos do .NET  
<https://docs.microsoft.com/dotnet/fundamentals>
- Portando de .NET Framework para o .NET Core  
<https://docs.microsoft.com/dotnet/core/porting/index>
- .NET Core no guia do Docker  
<https://docs.microsoft.com/dotnet/core/docker/introduction>
- Visão geral dos componentes .NET  
<https://docs.microsoft.com/dotnet/standard/components>

[ANTERIOR](#)

[AVANÇAR](#)

# Tabela de decisões: estruturas .NET a serem usadas para o Docker

18/03/2020 • 2 minutes to read • [Edit Online](#)

A tabela de decisão a seguir resume se deve-se usar o .NET Framework ou o .NET Core. Lembre-se de que, para contêineres Linux, são necessários hosts do Docker baseados em Linux (VMs ou servidores) e de que, para contêineres do Windows, são necessários hosts do Docker baseados em Windows Server (VMs ou servidores).

## IMPORTANT

Os computadores de desenvolvimento executarão um host Docker, seja Linux ou Windows. Microsserviços relacionados que você deseja executar e testar em conjunto em uma solução precisarão ser executados na mesma plataforma de contêiner.

ARQUITETURA/TIPO DE APLICATIVO	CONTÊINERES DO LINUX	CONTÊINERES DO WINDOWS
Microsserviços em contêineres	.NET Core	.NET Core
Aplicativo monolítico	.NET Core	.NET Framework .NET Core
Melhores desempenho e escalabilidade da categoria	.NET Core	.NET Core
Migração do aplicativo herdado do Windows Server ("campo-marrom") para contêineres	--	.NET Framework
Novo desenvolvimento baseado em contêiner ("campo-verde")	.NET Core	.NET Core
ASP.NET Core	.NET Core	.NET Core (recomendado) .NET Framework
ASP.NET 4 (MVC 5, API Web 2 e Web Forms)	--	.NET Framework
Serviços SignalR	.NET Core 2.1 ou versão posterior	.NET Framework .NET Core 2.1 ou versão posterior
WCF, WF e outras estruturas herdadas	WCF no .NET Core (somente biblioteca de clientes)	.NET Framework WCF no .NET Core (somente biblioteca de clientes)
Consumo de serviços do Azure	.NET Core (eventualmente, a maioria dos serviços do Azure fornecerá SDKs de clientes para .NET Core)	.NET Framework .NET Core (eventualmente, a maioria dos serviços do Azure fornecerá SDKs de clientes para .NET Core)

[PRÓXIMO](#)

[ANTERIOR](#)

# Para qual sistema operacional direcionar com os contêineres do .NET

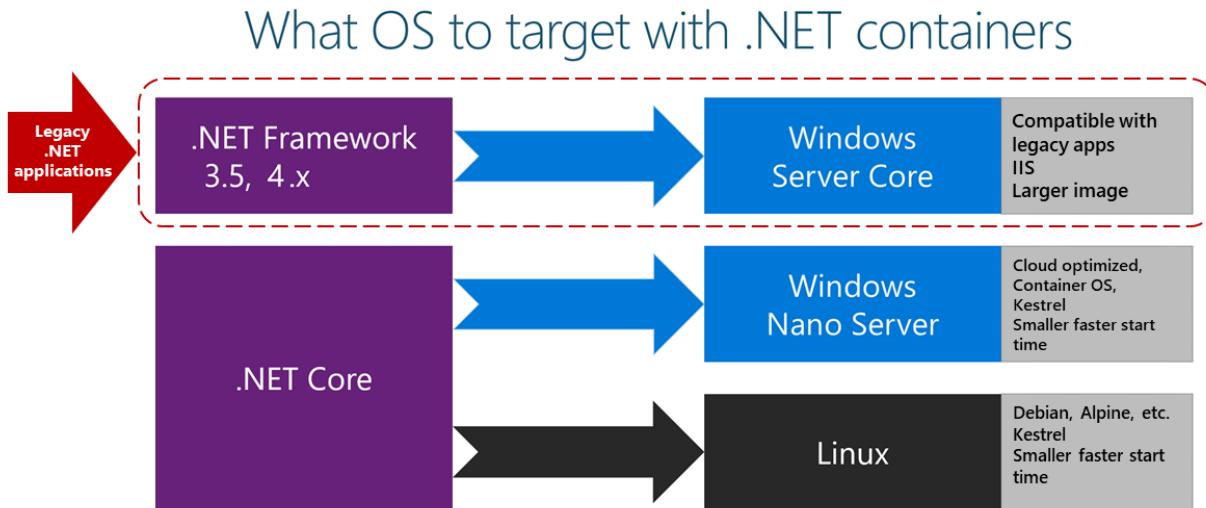
18/03/2020 • 4 minutes to read • [Edit Online](#)

Dada a diversidade de sistemas operacionais compatíveis com o Docker e as diferenças entre o .NET Framework e o .NET Core, você deve direcionar um sistema operacional e versões específicos dependendo da estrutura que você está usando.

Para o Windows, é possível usar o Windows Server Core ou o Windows Nano Server. Essas versões do Windows oferecem características diferentes (IIS no Windows Server Core versus um servidor Web auto-hospedado como Kestrel no Nano Server) que podem ser exigidas pelo .NET Framework ou pelo .NET Core, respectivamente.

Para Linux, várias distribuições estão disponíveis e há compatibilidade com elas nas imagens oficiais do .NET Docker (como Debian).

Na Figure 3-1, é possível ver a possível versão do sistema operacional dependendo do .NET Framework usado.



**Figura 3-1.** Sistemas operacionais a serem direcionados dependendo das versões do .NET Framework

Ao implantar aplicativos legados do .NET Framework, você tem que segmentar o Windows Server Core, compatível com aplicativos legados e IIS, mas ele tem uma imagem maior. Ao implantar aplicativos .NET Core, é possível definir o Windows Nano Server como destino, que é otimizado para nuvem, usa o Kestrel, é menor e inicia mais rapidamente. Também é possível definir Linux, Debian de suporte, Alpine e outros como destino. Também usa Kestrel, é menor, e começo mais rápido.

Também é possível criar sua própria imagem do Docker em casos em que você deseja usar uma distribuição diferente do Linux ou em que você quer uma imagem com versões não fornecidas pela Microsoft. Por exemplo, você pode criar uma imagem com o ASP.NET Core em execução no .NET Framework tradicional e no Windows Server Core, que é um cenário não tão comum para Docker.

#### IMPORTANT

Ao usar imagens do Windows Server Core, você pode descobrir que alguns DLLs estão faltando, quando comparados com imagens completas do Windows. Você pode ser capaz de resolver esse problema criando uma imagem personalizada do Server Core, adicionando os arquivos ausentes no tempo de compilação de imagem, como mencionado neste comentário do [GitHub](#).

Ao adicionar o nome de imagem ao seu arquivo Dockerfile, é possível selecionar o sistema operacional e a versão dependendo da marcação usada, como nos seguintes exemplos:

IMAGEM	COMENTÁRIOS
mcr.microsoft.com/dotnet/core/runtime:3.1	Multiarquitetura .NET Core 3.1: suporta Linux e Windows Nano Server dependendo do host Docker.
mcr.microsoft.com/dotnet/core/aspnet:3.1	ASP.NET multiarquitetura Core 3.1: suporta Linux e Windows Nano Server dependendo do host Docker. A imagem aspnetcore tem algumas otimizações para ASP.NET Core.
mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim	.NET Core 3.1 somente runtime no Linux Debian distro
mcr.microsoft.com/dotnet/core/aspnet:3.1-nanoserver-1809	.NET Core 3.1 somente em tempo de execução no Windows Nano Server (versão 1809 do Windows Server)

## Recursos adicionais

- **BitmapDecoder falha devido à falta do WindowsCodecsExt.dll (problema do GitHub)**  
<https://github.com/microsoft/dotnet-framework-docker/issues/299>

[PRÓXIMO](#)

[ANTERIOR](#)

# Imagens oficiais do .NET Docker

18/03/2020 • 6 minutes to read • [Edit Online](#)

As imagens oficiais do .NET Docker são imagens do Docker criadas e otimizadas pela Microsoft. Eles estão disponíveis publicamente nos repositórios da Microsoft no [Docker Hub](#). Cada repositório pode conter várias imagens, dependendo de versões do .NET e, dependendo do sistema operacional e das versões (Linux Debian, Linux Alpine, Windows Nano Server, Windows Server Core, etc.).

Desde o .NET Core 2.1, todas as imagens do .NET Core, incluindo ASP.NET Core, [https://hub.docker.com/\\_/microsoft-dotnet-core](https://hub.docker.com/_/microsoft-dotnet-core) estão disponíveis no Docker Hub no repositório de imagens .NET Core: .

Desde maio de 2018, as imagens da Microsoft estão sendo [sindicalizadas no Microsoft Container Registry](#). O catálogo oficial ainda está disponível apenas no Docker Hub, e lá você encontrará o endereço atualizado para puxar a imagem.

A maioria dos repositórios de imagem fornece uma marcação extensiva para ajudá-lo a selecionar não apenas uma versão específica da estrutura, mas também para escolher um SISTEMA OPERACIONAL (distribuição Linux ou versão do Windows).

## Otimizações de imagem do .NET Core e do Docker para desenvolvimento versus produção

Ao criar imagens do Docker para desenvolvedores, a Microsoft se concentrava nos seguintes cenários principais:

- Imagens usadas para *desenvolver* e criar aplicativos .NET Core.
- Imagens usadas para *executar* aplicativos .NET Core.

Por que várias imagens? Ao desenvolver, criar e executar aplicativos em contêineres, você geralmente tem prioridades diferentes. Fornecendo diferentes imagens para essas tarefas separadas, a Microsoft ajuda a otimizar os processos separados de desenvolver, criar e implantar aplicativos.

### Durante o desenvolvimento e o build

Durante o desenvolvimento, o que é importante é a velocidade em que é possível iterar alterações e a capacidade de depurá-las. O tamanho da imagem não é tão importante quanto a capacidade de fazer alterações no seu código e ver as alterações rapidamente. Algumas ferramentas e "recipientes de agente de construção", usam o desenvolvimento da imagem .NET Core ([mcr.microsoft.com/dotnet/core/sdk:3.1](https://mcr.microsoft.com/dotnet/core/sdk:3.1)) durante o processo de desenvolvimento e construção. Ao construir dentro de um contêiner Docker, os aspectos importantes são os elementos necessários para compilar seu aplicativo. Isso inclui o compilador e quaisquer outras dependências do .NET.

Por que este tipo de imagem de build é importante? Você não implanta esta imagem na produção. Em vez disso, é uma imagem que você usa para construir o conteúdo que você coloca em uma imagem de produção. Essa imagem seria usada em seu ambiente de integração contínua (CI) ou ambiente de construção ao usar compilações multiestágio sumidas do Docker.

### Em produção

O que é importante na produção é a rapidez com que é possível implantar e iniciar seus contêineres com base em uma imagem do .NET Core de produção. Portanto, a imagem somente em tempo de execução com base em [mcr.microsoft.com/dotnet/core/aspnet:3.1](https://mcr.microsoft.com/dotnet/core/aspnet:3.1) é pequena para que ela possa viajar rapidamente pela rede do seu registro Docker para seus hosts Docker. O conteúdo está pronto para ser executado, habilitando o tempo mais

rápido possível da inicialização do contêiner ao processamento de resultados. No modelo do Docker, não há necessidade de compilação de código C#, como há quando você executa dotnet build ou dotnet publish ao usar o contêiner de build.

Nesta imagem otimizada, você coloca apenas os binários e outros conteúdos necessários para executar o aplicativo. Por exemplo, o conteúdo `dotnet publish` criado por contém apenas os binários.NET compilados, imagens, js e .css. Ao longo do tempo, você verá imagens que contêm pacotes pré-compilados (compilação de IL para nativa que ocorre em runtime).

Embora haja várias versões das imagens do .NET Core e do ASP.NET Core, todas elas compartilham uma ou mais camadas, incluindo a camada base. Portanto, a quantidade de espaço em disco necessário para armazenar uma imagem é pequena, ela é composta apenas pelo delta entre sua imagem personalizada e sua imagem de base. O resultado é que é rápido extrair a imagem do Registro.

Quando você explorar os repositórios de imagem do .NET no Hub do Docker, encontrará várias versões de imagem confidenciais ou marcadas. Essas marcas ajudam a decidir qual usar, dependendo da versão de que você precisa, como as que estão na tabela a seguir:

IMAGEM	COMENTÁRIOS
<code>mcr.microsoft.com/dotnet/core/aspnet:3.1</code>	ASP.NET Core, somente com runtime e otimizações de ASP.NET Core, no Linux e no Windows (várias arquiteturas)
<code>mcr.microsoft.com/dotnet/core/sdk:3.1</code>	.NET Core, com SDKs incluídos, no Linux e no Windows (várias arquiteturas)

[PRÓXIMO](#)

[ANTERIOR](#)

# Como arquitetar aplicativos baseados em contêineres e em microsserviços

18/03/2020 • 4 minutes to read • [Edit Online](#)

*Os microsserviços oferecem grandes benefícios, mas também levantam enormes novos desafios. Padrões de arquitetura de microsserviços são pilares fundamentais ao criar um aplicativo baseado em microsserviços.*

Antes neste guia, você aprendeu os conceitos básicos sobre contêineres e sobre o Docker. Esse é o mínimo de informações necessárias para a introdução aos contêineres. Embora os contêineres habilitem os microsserviços e sejam muito adequados para isso, eles não são obrigatórios para uma arquitetura de microsserviço e muitos conceitos de arquitetura nesta seção de arquitetura também podem ser aplicados sem contêineres. No entanto, este guia concentra-se na interseção de ambos devido a importância dos contêineres que já foi apresentada.

Os aplicativos corporativos podem ser complexos e geralmente são compostos de vários serviços e não em um único serviço. Para esses casos, você precisa entender as abordagens de arquitetura adicionais, como os microsserviços e certos padrões de DDD (design controlado por domínio) além de conceitos de orquestração de contêiner. Observe que este capítulo não descreve apenas microsserviços em contêineres, mas também qualquer aplicativo em contêiner.

## Princípios de design de contêiner

No modelo de contêiner, uma instância de imagem de contêiner representa um único processo. Definindo uma imagem de contêiner como um limite de processo, você pode criar primitivas que podem ser usadas para dimensionar o processo ou criar um lote.

Ao criar uma imagem de contêiner, você verá uma definição [ENTRYPOINT](#) no Dockerfile. Ela define o processo cujo tempo de vida controla o tempo de vida do contêiner. Quando o processo for concluído, o ciclo de vida do contêiner será encerrado. Os contêineres podem representar processos de longa execução como servidores Web, mas também podem representar processos de curta duração, como trabalhos em lotes, que anteriormente talvez eram implementados como Azure [WebJobs](#).

Se o processo falhar, o contêiner é encerrado e o orquestrador assume. Se o orquestrador foi configurado para manter cinco instâncias em execução e uma falhar, ele criará outra instância de contêiner para substituir o processo com falha. Em um trabalho em lotes, o processo é iniciado com parâmetros. Quando o processo é concluído, o trabalho é concluído. Mais tarde, este guia faz drill-down em orquestradores.

Você pode encontrar um cenário em que deseja vários processos em execução em um único contêiner. Para esse cenário, como pode haver apenas um ponto de entrada por contêiner, você poderá executar um script dentro do contêiner para iniciar quantos programas forem necessários. Por exemplo, você pode usar o [Supervisor](#) ou uma ferramenta semelhante para cuidar da inicialização de vários processos dentro de um único contêiner. No entanto, mesmo que seja possível encontrar arquiteturas com vários processos por contêiner, essa abordagem não é muito comum.

[PRÓXIMO](#)

[ANTERIOR](#)

# Implantar aplicativos monolíticos em contêineres

18/03/2020 • 11 minutes to read • [Edit Online](#)

Talvez você queira criar um aplicativo ou serviço Web único e monolítico e implantá-lo como um contêiner. O aplicativo em si pode não ser monolítico internamente, mas estruturado como várias bibliotecas, componentes ou até mesmo camadas (camada de aplicativo, de domínio, de acesso a dados, etc.). No entanto, externamente ele é um contêiner único – um processo único, um aplicativo Web único ou um serviço único.

Para gerenciar esse modelo, implante um contêiner único para representar o aplicativo. Para aumentar a capacidade, você expande, ou seja, apenas adiciona mais cópias com um平衡ador de carga na frente. A simplicidade está em gerenciar uma implantação única em um contêiner ou VM único.

## Monolithic Containerized application

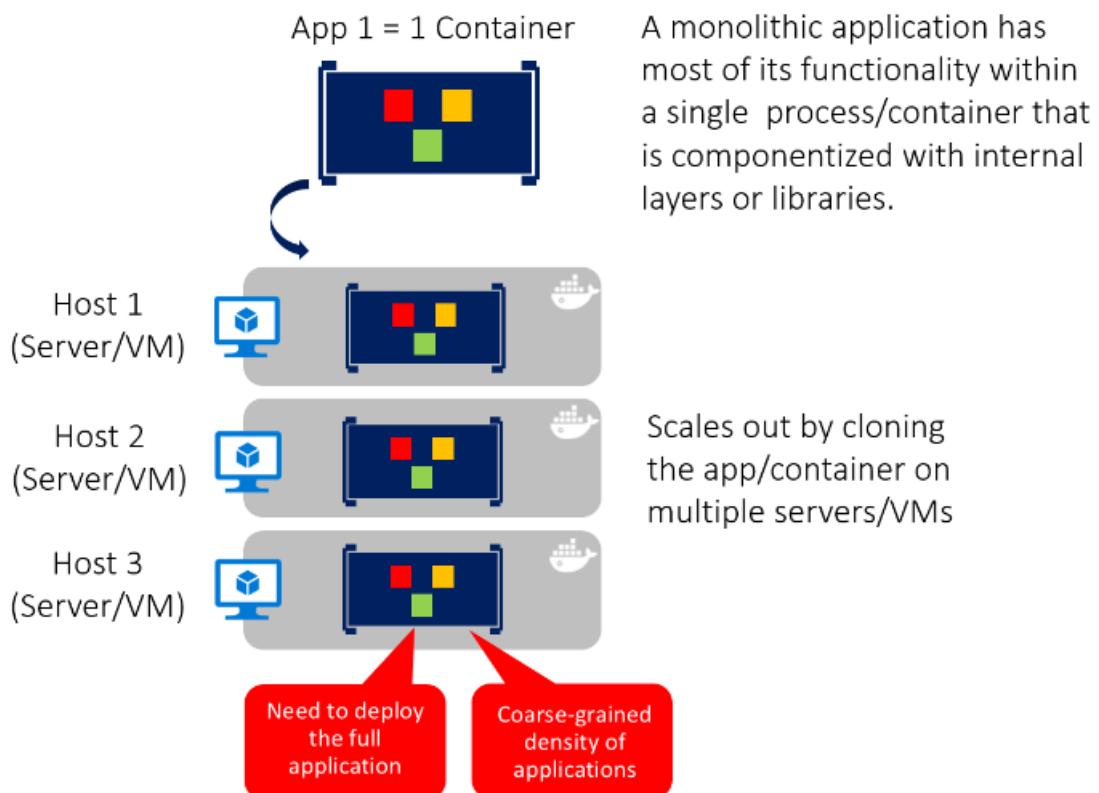


Figura 4-1. Exemplo de arquitetura de um aplicativo monolítico em contêiner

É possível incluir vários componentes, bibliotecas ou camadas internas em cada contêiner, conforme ilustrado na Figura 4-1. Um aplicativo em contêineres monolíticos tem a maior parte de sua funcionalidade dentro de um único contêiner, com camadas internas ou bibliotecas, e se escoa por clonagem do contêiner em vários servidores/VMs. No entanto, esse padrão monolítico pode entrar em conflito com o princípio do contêiner: "um contêiner executa uma ação em um processo". Porém, em alguns casos não haverá problemas.

O ponto negativo dessa abordagem ficará evidente se o aplicativo crescer e for necessário dimensioná-lo. Se o aplicativo inteiro puder ser dimensionado, não será realmente um problema. Entretanto, na maioria dos casos apenas algumas partes do aplicativo são os pontos de redução que exigem escalonamento, enquanto outros componentes são menos utilizados.

Por exemplo, em um aplicativo comum de comércio eletrônico, provavelmente é preciso dimensionar o

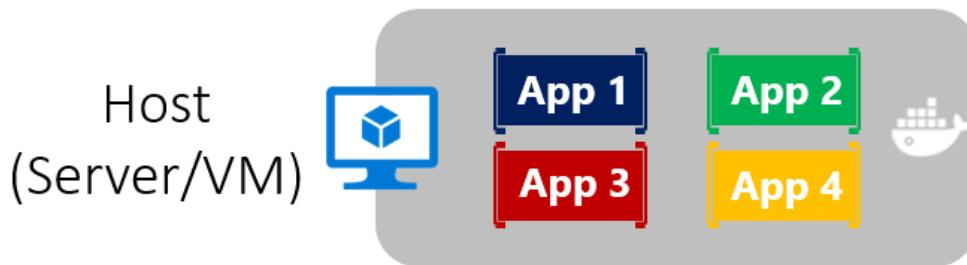
subsistema de informações do produto, pois a procura por produtos é maior do que a compra. Mais clientes usam o carrinho em vez do pipeline de pagamento. Menos clientes fazem comentários ou exibem o histórico de compras. E você pode ter apenas um punhado de funcionários que precisam gerenciar o conteúdo e campanhas de marketing. Ao escalar o design monolítico, todo o código dessas tarefas diferentes é implantado diversas vezes e dimensionado no mesmo grau.

Há várias maneiras de dimensionar um aplicativo: duplicação horizontal, divisão de diferentes áreas do aplicativo e partição de conceitos ou dados empresariais semelhantes. No entanto, além do problema de dimensionamento de todos os componentes, alterar um único componente exige testar novamente todo o aplicativo e reimplantar todas as instâncias.

Porém, a abordagem monolítica é comum, pois o desenvolvimento do aplicativo é inicialmente muito mais fácil do que nas abordagens de microsserviços. Assim, diversas organizações desenvolvem usando essas abordagem de arquitetura. Embora algumas delas tenham obtido bons resultados, outras estão atingindo seus limites. Muitas organizações criaram aplicativos usando esse modelo porque as ferramentas e a infraestrutura dificultaram bastante a criação de SOAs (arquiteturas orientadas a serviços) no passado e elas não viam necessidade disso – até os aplicativos crescerem.

De uma perspectiva de infraestrutura, cada servidor pode executar vários aplicativos no mesmo host e ter um índice de eficiência razoável de uso de recursos, conforme mostrado na Figura 4-2.

## Host running multiple apps/containers



**Figura 4-2.** Abordagem monolítica: host executando vários aplicativos, cada um em execução como um contêiner

Os aplicativos monolíticos no Microsoft Azure podem ser implantados por meio de VMs dedicadas a cada instância. Além disso, usando [conjuntos de dimensionamento de máquinas virtuais do Azure](#), você pode dimensionar VMs facilmente. O [Serviço de Aplicativo do Azure](#) também executa aplicativos monolíticos e dimensiona instâncias facilmente sem necessidade de gerenciamento de VMs. Desde 2016, o Serviços de Aplicativos do Azure também pode executar instâncias únicas de contêineres do Docker, simplificando a implantação.

Como em um ambiente de garantia de qualidade ou de produção limitada, é possível implantar diversas VMs host do Docker e balanceá-las usando o平衡ador do Azure, conforme mostrado na Figura 4-3. Assim, você pode gerenciar o dimensionamento com uma abordagem de alta granularidade, pois o aplicativo inteiro está em um único contêiner.

# Architecture in Docker infrastructure for monolithic applications

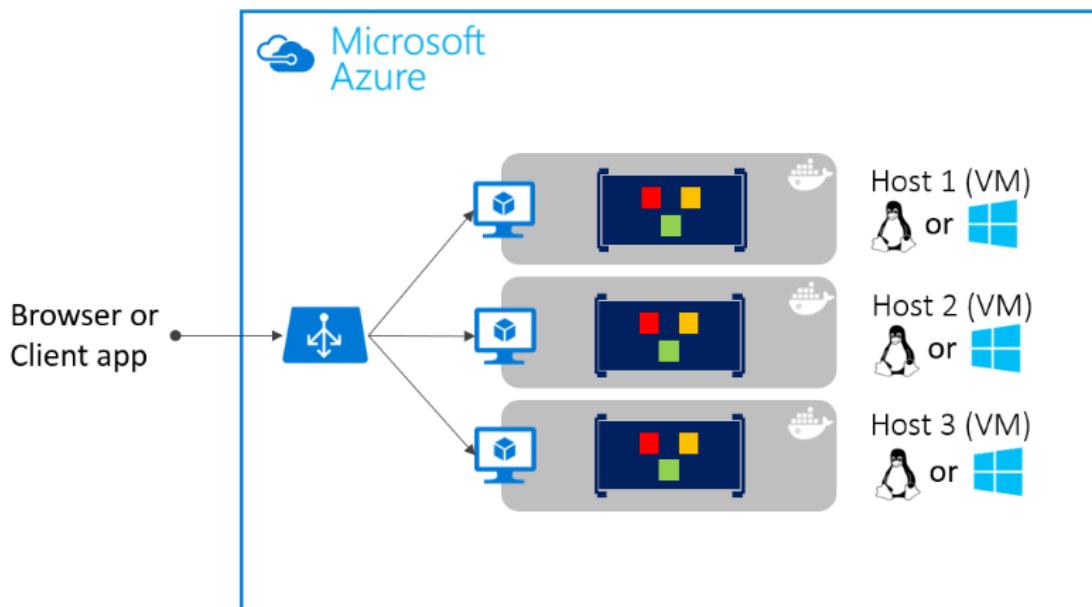


Figura 4-3. Exemplo de vários hosts escalando verticalmente um aplicativo em contêiner único

A implantação em vários hosts pode ser gerenciada com técnicas de implantação tradicionais. Os hosts do Docker podem ser gerenciados manualmente com comandos como `docker run` ou `docker-compose`. Outra opção é a automação, como os pipelines de entrega contínua.

## Implantar um aplicativo monolítico como um contêiner

Há benefícios em usar contêineres para gerenciar implantações de aplicativos monolíticos. O dimensionamento de instâncias de contêiner é muito mais rápido e fácil do que a implantação de VMs adicionais. Mesmo se você usar os conjuntos de dimensionamento de máquinas virtuais, as VMs levarão tempo para iniciar. Quando implantado como instâncias de aplicativo tradicionais em vez de contêineres, a configuração do aplicativo é gerenciada como parte da VM, o que não é o ideal.

Implantar atualizações como imagens do Docker é muito mais rápido e eficiente em termos de rede. As imagens do Docker se inicializam em questão de segundos, o que agiliza a distribuição. Destruir uma instância de imagem do Docker é tão fácil quanto emitir um comando `docker stop` e geralmente leva menos de um segundo.

Como o design dos contêineres é imutável, não é necessário se preocupar com VMs corrompidas. Por outro lado, os scripts de atualização de uma VM podem não levar em conta configurações específicas ou arquivos deixados no disco.

Embora aplicativos monolíticos possam se beneficiar do Docker, estamos mencionando apenas as vantagens. Outros pontos positivos de gerenciar contêineres são decorrentes da implantação com orquestradores de contêiner, que gerenciam as diversas instâncias e o ciclo de vida de cada instância de contêiner. Dividir o aplicativo monolítico em subsistemas que podem ser dimensionados, desenvolvidos e implantados individualmente é o ponto de entrada no universo dos microserviços.

## Publicar um aplicativo baseado em contêiner único no Serviço de Aplicativo do Azure

Seja para validar um contêiner implantado no Azure ou quando um aplicativo é baseado em contêiner único, o Serviço de Aplicativo do Azure é uma ótima maneira de oferecer serviços escalonáveis baseados em contêiner único. Usar o Serviço de Aplicativo do Azure é simples. Ele tem uma ótima integração com o GIT para facilitar a implantação do código criado no Visual Studio diretamente no Azure.

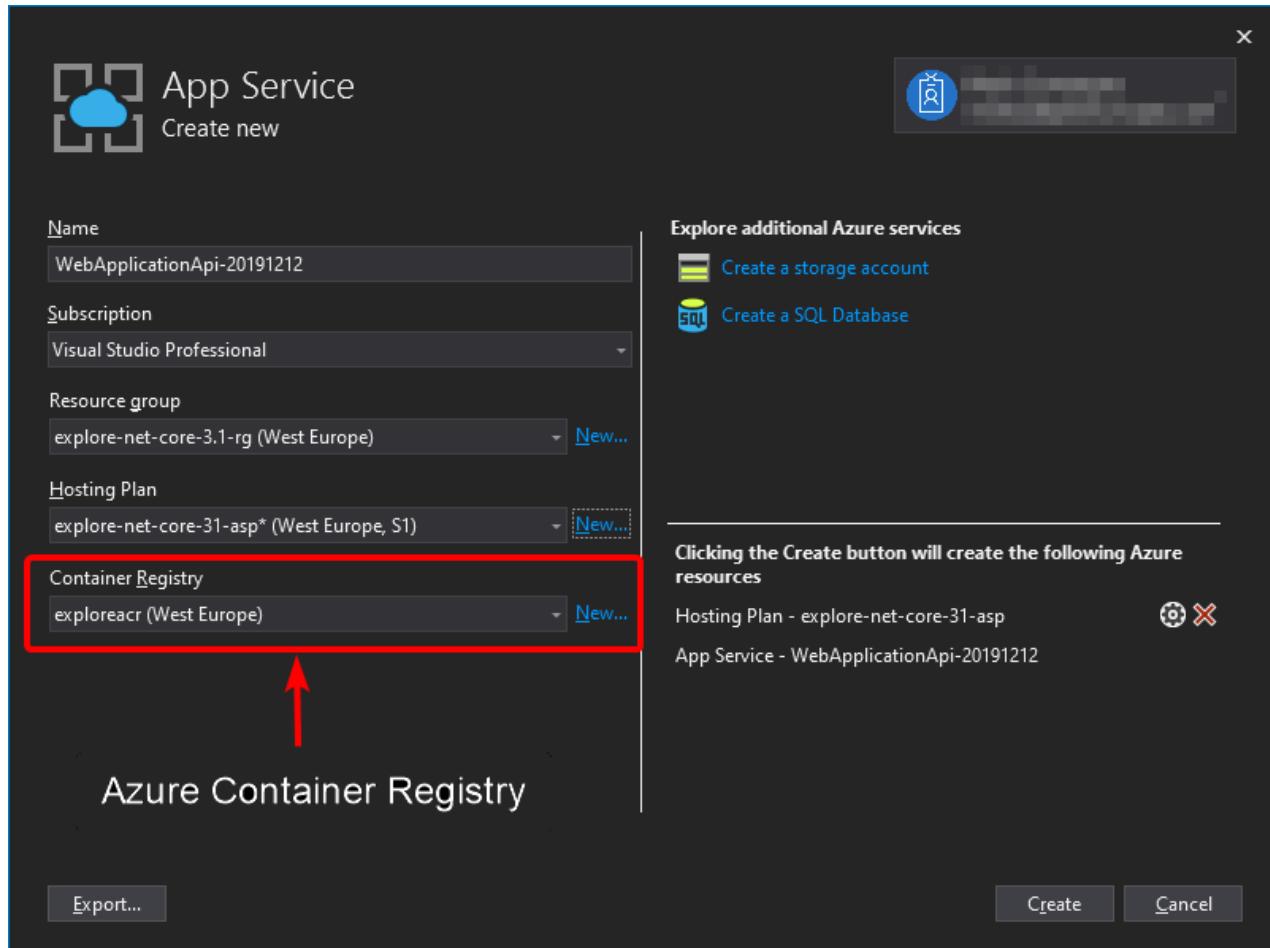


Figura 4-4. Publicando um aplicativo de contêiner único para o Azure App Service do Visual Studio 2019

Sem o Docker, se você precisasse de outros recursos, estruturas ou dependências sem suporte no Serviço de Aplicativo do Azure, seria necessário aguardar até que a equipe do Azure atualizasse essas dependências no Serviço de Aplicativo. Outra opção era mudar para outros serviços, como Serviços de Nuvem do Azure, ou VMs, em que se tinha mais controle e era possível instalar o componente ou a estrutura exigidos pelo aplicativo.

No Visual Studio 2017 e posterior, o suporte para contêineres oferece a capacidade de incluir tudo o que você deseja no ambiente do aplicativo, conforme mostrado na Figura 4-4. Como está sendo executado em um contêiner, se você adicionar uma dependência ao aplicativo, será possível incluí-la em um Dockerfile ou em uma imagem do Docker.

A Figura 4-4 também mostra que o fluxo de publicação envia uma imagem por meio do registro de contêiner. Isso pode ser feito pelo Registro de Contêiner do Azure (um registro próximo às implantações no Azure e protegido por grupos e contas do Azure Active Directory) ou outros registros do Docker, como o Docker Hub ou um registro local.

PRÓXIMO

ANTERIOR

# Estado e dados em aplicativos do Docker

18/03/2020 • 10 minutes to read • [Edit Online](#)

Na maioria dos casos, você pode considerar um contêiner uma instância de um processo. Um processo não mantém o estado persistente. Enquanto um contêiner pode gravar em seu armazenamento local, supondo que uma instância permanecerá indefinidamente seria como presumir que um único local na memória será durável. Você deve presumir que as imagens de contêiner, assim como os processos, têm várias instâncias ou serão eventualmente eliminadas. Se forem gerenciadas com um orquestrador de contêineres, você deverá presumir que podem ser movidas de um nó ou VM para outro.

As seguintes soluções são usadas para gerenciar dados em aplicativos Docker:

Do host do Docker, como [Volumes do Docker](#):

- **Volumes** são armazenados em uma área do sistema de arquivos de host que é gerenciado pelo Docker.
- **Montagens de associação** podem mapear para qualquer pasta no sistema de arquivos de host, portanto, o acesso não pode ser controlado do processo do Docker e pode representar um risco de segurança, já que um contêiner pode acessar pastas confidenciais do sistema operacional.
- **Montagens tmpfs** são como pastas virtuais que só existem na memória do host e nunca são gravadas no sistema de arquivos.

Do armazenamento remoto:

- [Armazenamento do Azure](#), que oferece armazenamento geograficamente distribuído, fornecendo uma boa solução de persistência de longo prazo para contêineres.
- Bancos de dados relacionais remotos, como [Banco de Dados SQL do Azure](#); bancos de dados NoSQL, como o [Azure Cosmos DB](#); ou serviços de cache, como o [Redis](#).

Do contêiner do Docker:

- **Sistema de arquivos sobreposição**. Este recurso DoDocker implementa uma tarefa de cópia na gravação que armazena informações atualizadas no sistema de arquivos raiz do contêiner. Essa informação está "em cima" da imagem original na qual o contêiner está baseado. Se o contêiner for excluído do sistema, essas alterações serão perdidas. Portanto, embora seja possível salvar o estado de um contêiner em seu armazenamento local, criar um sistema com base nisso entraria em conflito com o local do projeto do contêiner, que por padrão é sem estado.

No entanto, o uso do Docker Volumes é agora a maneira preferida de lidar com dados locais no Docker. Se você precisar de mais informações sobre o armazenamento em contêineres, procure em [Drivers de armazenamento do Docker](#) e [Sobre drivers de armazenamento](#).

O exemplo a seguir fornece mais detalhes sobre essas opções:

**Volumes** são diretórios mapeados do sistema operacional do host para diretórios em contêineres. Quando o código no contêiner tem acesso ao diretório, o acesso é, na verdade, a um diretório no sistema operacional host. Esse diretório não está vinculado ao tempo de vida do contêiner em si, sendo que o diretório é gerenciado pelo Docker e isolado em relação à funcionalidade básica do computador host. Assim, os volumes de dados são projetados para persistir dados independentemente do tempo de vida do contêiner. Se você excluir um contêiner ou uma imagem do host do Docker, os dados persistentes no volume de dados não serão excluídos.

Volumes podem ser nomeados ou anônimos (o padrão). Volumes nomeados são a evolução dos **Contêineres de**

**Volume de Dados** e facilitam o compartilhamento de dados entre contêineres. Volumes também são compatíveis com drivers de volume, que permitem armazenar dados em hosts remotos, entre outras opções.

**Montagens de associação** estão disponíveis há muito tempo e permitem o mapeamento de qualquer pasta para um ponto de montagem em um contêiner. Montagens de associação têm mais limitações que os volumes e alguns problemas de segurança importantes, por isso os volumes são a opção recomendada.

**Montagens tmpfs** são basicamente pastas virtuais que só existem na memória do host e nunca são gravadas no sistema de arquivos. Eles são rápidos e seguros, mas usam memória e são destinados apenas para dados temporários e não persistentes.

Conforme mostrado na Figura 4-5, os volumes Docker regulares podem ser armazenados fora dos contêineres de si, mas dentro dos limites físicos do servidor host ou VM. No entanto, contêineres do Docker não podem acessar um volume de um servidor host ou VM para outro. Em outras palavras, com esses volumes, não é possível gerenciar os dados compartilhados entre contêineres executados em hosts diferentes do Docker, embora esses dados possam ser obtidos com um driver de volume que dá suporte a hosts remotos.

## Data Volume and Data Volume Container

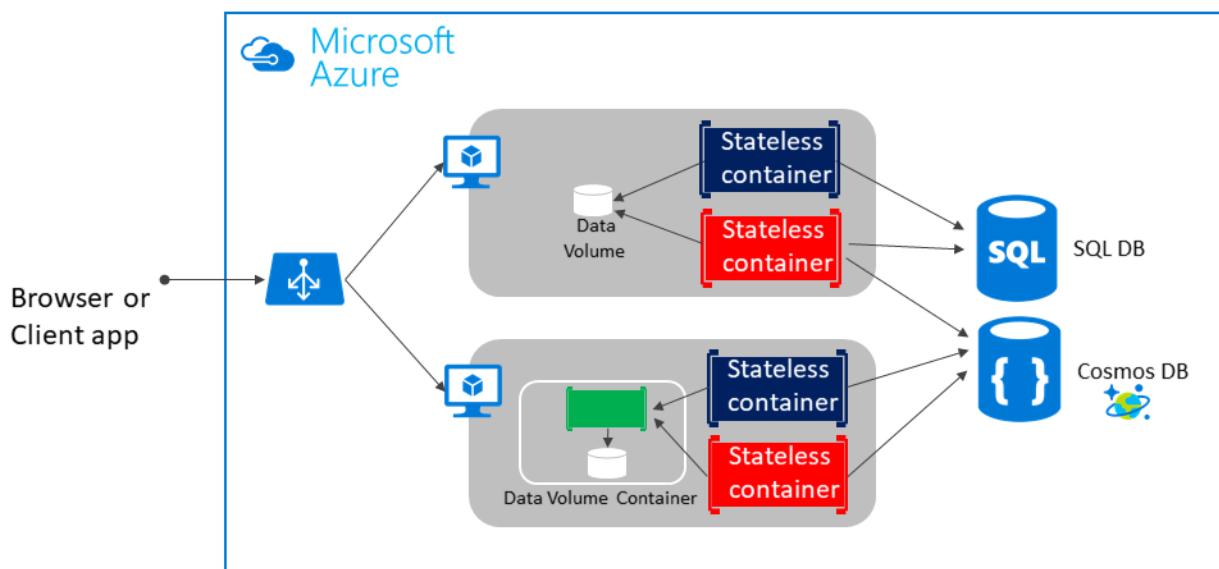


Figura 4-5. Volumes e fontes de dados externas para aplicativos baseados em contêiner

Volumes podem ser compartilhados entre contêineres, mas apenas no mesmo host, a menos que você use um driver remoto que dá suporte a hosts remotos. Além disso, quando gerenciados por um orquestrador, contêineres do Docker podem se "mover" entre os hosts de acordo com as otimizações realizadas pelo cluster. Portanto, não é recomendável usar volumes de dados para dados empresariais. Porém, eles são um bom mecanismo para trabalhar com arquivos de rastreamento, arquivos temporais ou similares que não afetarão a consistência dos dados empresariais.

**Fontes de dados remotas e ferramentas de cache** como o Banco de Dados SQL do Azure, o Azure Cosmos DB ou um cache remoto como o Redis podem ser usados em aplicativos em contêineres da mesma forma que são usados durante o desenvolvimento sem contêineres. Essa é uma forma comprovada de armazenar dados de aplicativo de negócios.

**Armazenamento do Microsoft Azure.** Os dados de negócios geralmente precisarão ser colocados em recursos ou em bancos de dados externos, como o Armazenamento do Azure. O Armazenamento do Azure fornece os seguintes serviços na nuvem:

- O Armazenamento de Blobs armazena dados de objeto não estruturados. Um blob pode ser qualquer tipo de texto ou dados binários, como documentos ou arquivos de mídia (imagens, áudio e vídeo). O Armazenamento de Blobs também é chamado de armazenamento de Objeto.

- O armazenamento de arquivos oferece armazenamento compartilhado para aplicativos herdados que usam protocolo SMB padrão. Os serviços de nuvem e as máquinas virtuais do Azure podem compartilhar dados de arquivos em vários componentes de aplicativo por meio de compartilhamentos montados. Aplicativos locais podem acessar dados de arquivo em um compartilhamento por meio da API REST do serviço de arquivo.
- O Armazenamento de Tabela armazena conjuntos de dados estruturados. O Armazenamento de Tabelas é um armazenamento de dados do atributo-chave NoSQL, que permite o rápido desenvolvimento e acesso a grandes quantidades de dados.

**Bancos de dados relacionais e bancos de dados NoSQL.** Existem muitas opções para bancos de dados externos, desde bancos de dados relacionais como SQL Server, PostgreSQL, Oracle ou NoSQL bancos de dados como Azure Cosmos DB, MongoDB, etc. Essas bases de dados não serão explicadas como parte deste guia, uma vez que estão em um assunto completamente diferente.

[PRÓXIMO](#)

[ANTERIOR](#)

# Arquitetura orientada a serviço

18/03/2020 • 2 minutes to read • [Edit Online](#)

A SOA (arquitetura orientada a serviço) era um termo usado de maneira exagerada e significava diferentes coisas para diferentes pessoas. Mas, como um denominador comum, a SOA significa que você estrutura seu aplicativo o decompondo em vários serviços (mais comumente, como serviços HTTP) que podem ser classificados como diferentes tipos como subsistemas ou camadas.

Agora esses serviços podem ser implantados como contêineres do Docker, que resolve problemas de implantação, porque todas as dependências estão incluídas na imagem de contêiner. No entanto, quando você precisa expandir aplicativos SOA, talvez você tenha desafios de escalabilidade e de disponibilidade se estiver implantando com base em hosts únicos do Docker. É aí que o software de clustering do Docker ou um orquestrador pode ajudar você, conforme explicado nas seções posteriores, em que são descritas as abordagens de implantação para microsserviços.

Os contêineres do Docker são úteis (mas não necessários) para arquiteturas tradicionais orientadas a serviço e para as arquiteturas de microsserviços mais avançadas.

Os microsserviços derivam da SOA, mas a SOA é diferente da arquitetura de microsserviços. Recursos como agentes centrais grandes, orquestradores centrais no nível da organização e o [ESB \(Enterprise Service Bus\)](#) são comuns na SOA. Mas, na maioria dos casos, esses são os antipadrões na comunidade de microsserviços. Na verdade, algumas pessoas argumentam que "A arquitetura de microsserviço é a SOA feita da maneira certa".

Este guia se concentra em microsserviços, porque uma abordagem SOA é menos prescritiva do que os requisitos e as técnicas usados em uma arquitetura de microsserviço. Se você souber como criar um aplicativo baseado em microsserviço, também saberá como criar um aplicativo mais simples orientado a serviços.

[PRÓXIMO](#)

[ANTERIOR](#)

# Arquitetura de microsserviços

18/03/2020 • 6 minutes to read • [Edit Online](#)

Como o nome implica, uma arquitetura de microsserviços é uma abordagem para criar um aplicativo para servidores como um conjunto de serviços pequenos. Isso significa que uma arquitetura de microsserviços é principalmente orientada para o back-end, embora a abordagem também esteja sendo usada para o front-end. Cada serviço é executado em seu próprio processo e comunica-se com outros processos usando protocolos como HTTP/HTTPS, WebSockets ou AMQP. Cada microsserviço implementa um domínio de ponta a ponta específico ou uma capacidade de negócios dentro de um determinado limite de contexto, e cada um deve ser desenvolvido de forma autônoma e ser implantado independentemente. Por fim, cada microsserviço deve ter seu modelo de dados de domínio relacionado e sua lógica de domínio (gerenciamento de dados decentralizados e de soberania) e pode ser baseado em diferentes tecnologias de armazenamento de dados (SQL, NoSQL) e diferentes linguagens de programação.

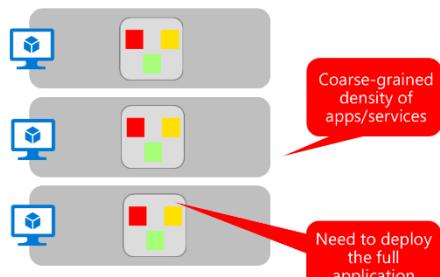
Que tamanho deve ter um microsserviço? Ao desenvolver um microsserviço, o tamanho não deve ser o ponto importante. Em vez disso, o ponto importante deve ser criar serviços acoplados livremente para você ter autonomia de desenvolvimento, implantação e escala, para cada serviço. É claro que, ao identificar e criar microsserviços, você deve tentar torná-los o menor possível, desde que você não tenha muitas dependências diretas com outros microsserviços. Mais importante do que o tamanho do microsserviço é a coesão interna que ele deve ter e sua independência de outros serviços.

Por que uma arquitetura de microsserviços? Em resumo, ela oferece agilidade de longo prazo. Os microsserviços permitem melhor facilidade de manutenção em sistemas complexos, grandes e altamente escalonáveis permitindo a criação de aplicativos com base em muitos serviços implantáveis de maneira independente em que cada um tenha ciclos de vida autônomos e granulares.

Como uma vantagem adicional, os microsserviços podem ser aumentados de forma independente. Em vez de ter um único aplicativo monolítico que você deve aumentar como uma unidade, é possível aumentar microsserviços específicos. Desta forma, é possível escalar apenas a área funcional que precisa de mais potência de processamento ou largura de banda de rede para dar suporte à demanda, em vez de aumentar outras áreas do aplicativo que não precisam ser escaladas. Isso significa economias de custo, porque você precisa de menos hardware.

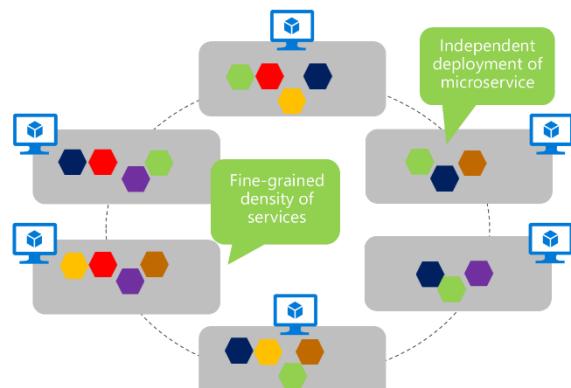
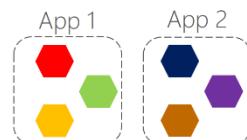
## Monolithic deployment approach

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



## Microservices application approach

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



## Figura 4-6. Implantação monolítica versus a abordagem de microserviços

Como a Figura 4-6 mostra, na abordagem monolítica tradicional, o aplicativo escala clonando todo o aplicativo em vários servidores/VM. Na abordagem de microserviços, a funcionalidade é separada em serviços menores para que cada serviço possa ser dimensionado independentemente. A abordagem de microserviços permite mudanças ágeis e iteração rápida de cada microserviço, pois você pode alterar áreas específicas, pequenas e complexas de aplicações complexas, grandes e escaláveis.

A arquitetura de aplicativos baseados em microserviços refinados permite práticas de integração e de entrega contínua. Ela também acelera a entrega de novas funções no aplicativo. A composição refinada de aplicativos também permite a você executar e testar microserviços em isolamento e evoluí-los de maneira autônoma ao manter contratos claros entre eles. Desde que você não altere a interfaces ou contratos, é possível alterar a implementação interna de qualquer microserviço ou adicionar novas funcionalidades sem interromper outros microserviços.

A seguir estão os aspectos importantes para habilitar o sucesso na entrada em produção com um sistema baseado em microserviços:

- Monitoramento e verificações de integridade dos serviços e da infraestrutura.
- Infraestrutura escalonável dos serviços (ou seja, nuvem e orquestradores).
- Design de segurança e implementação em vários níveis: autenticação, autorização, gerenciamento de segredos, comunicação segura, etc.
- Entrega rápida de aplicativos, geralmente com diferentes equipes focando-se em diferentes microserviços.
- DevOps e práticas e infraestrutura CI/CD.

Dessas, apenas as três primeiras serão abordadas ou introduzidas neste guia. Os dois últimos pontos, que estão relacionados ao ciclo de vida do aplicativo, são abordados no livro eletrônico adicional [Containerized Docker Application Lifecycle with Microsoft Platform and Tools](#) (Ciclo de vida do aplicativo Docker em contêineres com Microsoft Platform e ferramentas).

## Recursos adicionais

- **Mark Russinovich.** [Microserviços: Uma revolução de aplicativos alimentada pela nuvem](https://azure.microsoft.com/blog/microservices-an-application-revolution-powered-by-the-cloud/)  
<https://azure.microsoft.com/blog/microservices-an-application-revolution-powered-by-the-cloud/>
- **Martin Fowler.** [Microserviços](https://www.martinfowler.com/articles/microservices.html)  
<https://www.martinfowler.com/articles/microservices.html>
- **Martin Fowler.** [Pré-requisitos de microserviço](https://martinfowler.com/bliki/MicroservicePrerequisites.html)  
<https://martinfowler.com/bliki/MicroservicePrerequisites.html>
- **Jimmy Nilsson.** [Chunk Cloud Computing](https://www.infoq.com/articles/CCC-Jimmy-Nilsson)  
<https://www.infoq.com/articles/CCC-Jimmy-Nilsson>
- **Cesar de la Torre.** [Ciclo de vida do aplicativo Docker containerizado com plataforma e ferramentas microsoft](https://aka.ms/dockerlifecyclebook) (e-book para download)  
<https://aka.ms/dockerlifecyclebook>

# Soberania de dados por microsserviço

18/03/2020 • 10 minutes to read • [Edit Online](#)

Uma regra importante para a arquitetura de microsserviços é que cada um deles precisa ter dados e lógica de domínio. Assim como um aplicativo completo tem sua própria lógica e dados, cada microsserviço precisa tê-los em um ciclo de vida autônomo, com implantação independente por microsserviço.

Isso significa que o modelo conceitual do domínio será diferente entre subsistemas ou microsserviços. Imagine aplicativos empresariais, em que os aplicativos de gerenciamento de relacionamento com o cliente (CRM), subsistemas de compras transacionais e subsistemas de suporte ao cliente exigem atributos e dados únicos de entidade do cliente e aplicam um contexto delimitado diferente.

Esse princípio é semelhante no [DDD \(design controlado por domínio\)](#), em que cada [Contexto Delimitado](#) ou subsistema autônomo ou serviço precisa ter seu próprio modelo de domínio (dados, lógica e comportamento). Cada Contexto Delimitado por DDD se correlaciona a um microsserviço de negócios (um ou vários serviços). Esse ponto do padrão do Contexto Delimitado é expandido na próxima seção.

Por outro lado, a abordagem tradicional (dados monolíticos) usada em diversos aplicativos é ter um banco de dados único centralizado ou poucos bancos de dados. Geralmente, trata-se de um Banco de Dados SQL normalizado utilizado para o aplicativo inteiro e todos seus subsistemas internos, conforme mostrado na Figura 4-7.

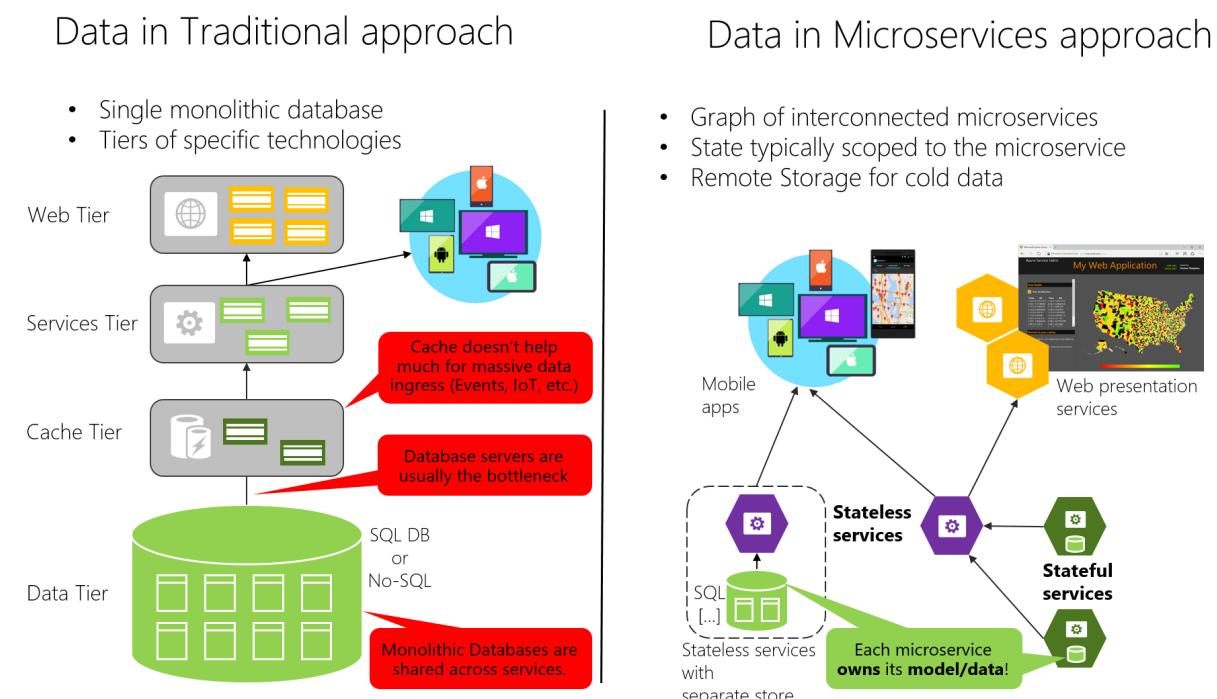


Figura 4-7. Comparação de soberania de dados: banco de dados monolítico x microsserviços

Na abordagem tradicional, há um único banco de dados individual entre todos os serviços, normalmente em uma arquitetura em camadas. Na abordagem de microsserviços, cada microsserviço possui seu modelo/dados. A abordagem de banco de dados centralizado aparenta ser mais simples inicialmente e parece permitir a reutilização de entidades em subsistemas diferentes para tornar tudo consistente. Mas a realidade é que você acaba com uma quantidade enorme de tabelas que servem para muitos subsistemas diferentes e incluem atributos e colunas que não são necessárias na maioria dos casos. É como tentar usar o mesmo mapa físico para uma caminhada em uma trilha pequena, uma viagem de carro de um dia e para aprender geografia.

Um aplicativo monolítico com um banco de dados relacional único conta com duas vantagens: [transações ACID](#) e a linguagem SQL, ambas em funcionamento em todas as tabelas e dados relacionados ao aplicativo. Essa abordagem é uma maneira fácil de gravar consultas que combinam dados de diversas tabelas.

No entanto, o acesso a dados se torna muito mais complicado quando você se muda para uma arquitetura de microsserviços. Mesmo ao usar transações ACID dentro de um microserviço ou Contexto Limitado, é crucial considerar que os dados de cada microserviço são privados para esse microserviço e só devem ser acessados sincronicamente através de seus pontos finais de API (REST, gRPC, SABÃO, etc) ou assíncronamente via mensagens (AMQP ou similar).

Encapsular os dados garante que os microsserviços sejam acoplados de forma flexível e evoluam de modo independente. Se vários serviços acessarem os mesmos dados, as atualizações de esquema exigirão atualizações coordenadas de todos os serviços. Isso interrompe a autonomia do ciclo de vida do microsserviço. Entretanto, as estruturas de dados distribuídos impedem você de realizar transações ACID nos microsserviços. Por sua vez, isso significa que é necessário utilizar consistência eventual quando um processo de negócios abrange vários microsserviços. Isso é muito mais difícil de implementar do que junções SQL, porque você não pode criar restrições de integridade nem usar transações distribuídas entre bancos de dados separados, conforme explicaremos mais adiante. Da mesma forma, muitos outros recursos do banco de dados relacional não estão disponíveis em vários microsserviços.

Além disso, microsserviços diferentes geralmente usam *tipos* diferentes de bancos de dados. Aplicativos modernos armazenam e processam tipos de dados diversificados e um banco de dados relacional nem sempre é a melhor opção. Em alguns casos de uso, um banco de dados NoSQL como o Azure CosmosDB ou o MongoDB podem ter um modelo de dados mais conveniente e oferecer desempenho e escalabilidade melhores que um banco de dados SQL como o SQL Server ou o Banco de Dados SQL do Azure. Em outros casos, um banco de dados relacional ainda é a melhor abordagem. Portanto, aplicativos baseados em microsserviço geralmente usam uma combinação de bancos de dados SQL e NoSQL, chamada às vezes de abordagem de [persistência poliglota](#).

Uma arquitetura partitionada e de persistência poliglota para armazenamento de dados traz muitos benefícios. Entre eles, serviços acoplados de forma flexível, desempenho e escalabilidade melhores e capacidade de gerenciamento. No entanto, essa arquitetura pode apresentar alguns desafios relacionados ao gerenciamento de dados distribuídos, conforme explicado mais adiante neste capítulo, em "[Identificar os limites do modelo de domínio](#)".

## A relação entre os microsserviços e o padrão do Contexto Delimitado

O conceito de microsserviço tem origem no [padrão de BC \(Contexto Delimitado\)](#) na [DDD \(design controlado por domínio\)](#). A DDD aborda modelos grandes dividindo-os em vários BCs e explicitando seus limites. Cada BC precisa ter um modelo e banco de dados próprios. Da mesma forma, cada microsserviço tem seus próprios dados relacionados. Além disso, cada BC geralmente tem uma [linguagem ubíqua](#) própria para ajudar na comunicação entre desenvolvedores de software e especialistas em domínio.

Esses termos (principalmente entidades de domínio) na linguagem ubíqua podem ter nomes diferentes em Contextos Delimitados distintos, mesmo quando entidades de domínio diferentes compartilham a mesma identidade (ou seja, a ID exclusiva utilizada para ler a entidade do armazenamento). Por exemplo, em um Contexto Delimitado de perfil de usuário, a entidade de domínio Usuário pode compartilhar a identidade com a entidade de domínio Comprador no Contexto Delimitado de pedido.

Desse modo, um microsserviço é como um Contexto Delimitado, mas também especifica seu caráter de serviço distribuído. Ele é criado como um processo separado para cada Contexto Delimitado e precisa usar os protocolos distribuídos mencionados anteriormente, como HTTP/HTTPS, WebSockets ou [AMQP](#). Entretanto, o padrão do Contexto Delimitado não especifica se é um serviço distribuído ou simplesmente um limite lógico (como um subsistema genérico) em um aplicativo de implantação monolítica.

É importante salientar que definir um serviço para cada Contexto Delimitado é uma boa maneira de começar.

Porém, não é preciso restringir seu design a ele. Às vezes, é necessário criar um Contexto Delimitado ou microsserviço de negócios composto de vários serviços físicos. Mas, por fim, ambos os padrões — Contexto Delimitado e microsserviço — estão intimamente relacionados.

A DDD se beneficia dos microsserviços ao obter limites reais na forma de microsserviços distribuídos. Contudo, ideias como o não compartilhamento do modelo entre os microsserviços também são desejáveis em um Contexto Delimitado.

### Recursos adicionais

- **Chris Richardson. Padrão: Banco de dados por serviço**  
<https://microservices.io/patterns/data/database-per-service.html>
- **Martin Fowler. Contexto limitado**  
<https://martinfowler.com/bliki/BoundedContext.html>
- **Martin Fowler. PoliglotaPersistência**  
<https://martinfowler.com/bliki/PolyglotPersistence.html>
- **Alberto Brandolini. Design estratégico orientado a domínios com mapeamento de contexto**  
<https://www.infoq.com/articles/ddd-contextmapping>

[PRÓXIMO](#)

[ANTERIOR](#)

# Arquitetura lógica versus arquitetura física

18/03/2020 • 4 minutes to read • [Edit Online](#)

Neste momento, é útil parar e discutir a diferença entre as arquiteturas lógica e física, além de como isso se aplica ao design de aplicativos baseados em microsserviços.

Para começar, a criação de microsserviços não exige o uso de nenhuma tecnologia específica. Por exemplo, os contêineres do Docker não são obrigatórios para criar uma arquitetura baseada em microsserviço. Essas microsserviços também podem ser executados como processos simples. Os microsserviços são uma arquitetura lógica.

Além disso, mesmo quando um microsserviço puder ser implementado fisicamente como um único serviço, processo ou contêiner (para simplificar, essa é a abordagem usada na versão inicial de [eShopOnContainers](#)), essa paridade entre microsserviço empresarial e serviço ou contêiner físico não será necessariamente obrigatória em todos os casos quando você criar um aplicativo grande e complexo composto por muitas dezenas ou até mesmo centenas de serviços.

É aqui em que há uma diferença entre a arquitetura lógica e a arquitetura física de um aplicativo. A arquitetura lógica e os limites de lógicos de um sistema não necessariamente mapeiam individualmente para a arquitetura física ou de implantação. Isso pode acontecer, mas geralmente não acontece.

Embora você possa ter identificado determinados microsserviços empresariais ou Contextos Limitados, isso não significa que a melhor maneira de implementá-los seja sempre criando um único serviço (como uma ASP.NET Web API) ou um único contêiner do Docker para cada microsserviço empresarial. Ter uma regra que diz que cada microsserviço de negócios deve ser implementado usando um único serviço ou contêiner é rígido demais.

Portanto, um microsserviço de negócios ou um Contexto limitado é uma arquitetura lógica que pode coincidir (ou não) com a arquitetura física. O ponto importante é que um microsserviço de negócios ou Contexto limitado deve ser autônomo, permitindo que o controle de versão, a implantação e a colocação em escala do código e do estado sejam feitas de maneira independente.

Como mostra a Figura 4-8, o microsserviço de negócios de catálogo pode ser composto por vários serviços ou processos. Eles podem ser vários serviços do ASP.NET Web API ou qualquer outro tipo de serviços que usam HTTP ou qualquer outro protocolo. Mais importante, os serviços podem compartilhar os mesmos dados, desde que esses serviços sejam coesos em relação ao mesmo domínio de negócios.

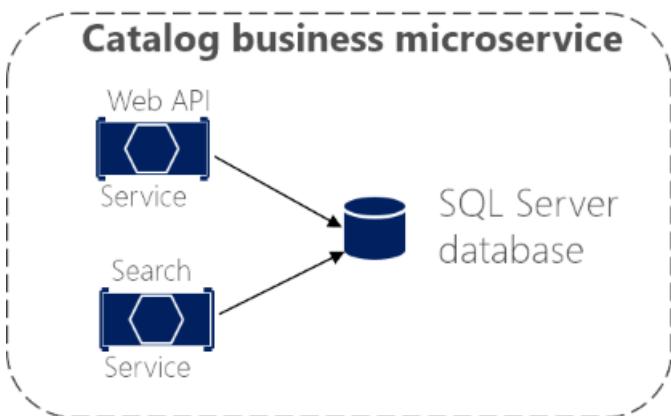


Figura 4-8. Microsserviço de negócios com vários serviços físicos

Os serviços no exemplo compartilham o mesmo modelo de dados, porque o serviço de API Web tem como alvo os mesmos dados que o Serviço de Pesquisa. Portanto, na implementação física do microsserviço empresarial, você está dividindo essa funcionalidade para poder escalar ou reduzir verticalmente cada um desses serviços

internos, conforme necessário. Talvez o serviço de API Web geralmente precise de mais instâncias do que o Serviço de Pesquisa ou vice-versa.

Em resumo, a arquitetura lógica de microsserviços nem sempre precisa coincidir com a arquitetura de implantação física. Neste guia, sempre que mencionamos um microsserviço, queremos dizer um microsserviço lógico ou empresarial que pode ser mapeado para um ou mais serviços (físicos). Na maioria dos casos, esse será um único serviço, mas pode ser mais.

[PRÓXIMO](#)

[ANTERIOR](#)

# Desafios e soluções do gerenciamento de dados distribuídos

10/09/2020 • 22 minutes to read • [Edit Online](#)

## Desafio #1: Como definir os limites de cada microsserviço

Definir os limites dos microsserviços provavelmente é o primeiro desafio que qualquer pessoa encontra. Cada microsserviço deve ser uma parte do aplicativo e cada microsserviço deve ser autônomo com todos os benefícios e desafios que ele abrange. Mas como identificar esses limites?

Primeiro, você precisa se concentrar nos modelos de domínio lógicos do aplicativo e nos dados relacionados. Tente identificar ilhas de dados desacopladas e contextos diferentes dentro do mesmo aplicativo. Cada contexto pode ter uma linguagem de negócios diferente (termos de negócios diferentes). Os contextos devem ser definidos e gerenciados de forma independente. Os termos e as entidades usados nesses contextos diferentes podem parecer semelhantes, mas você descobrirá que, em um contexto específico, um conceito empresarial é usado para uma finalidade e em outro contexto para outra finalidade, podendo até mesmo ter um nome diferente. Por exemplo, um usuário pode ser mencionado como um usuário no contexto de identidade ou de associação, como um cliente em um contexto de CRM, como um comprador em um contexto de pedidos e assim por diante.

Sua forma de identificar os limites entre vários contextos de aplicativo com um domínio diferente para cada contexto é exatamente a forma que você pode usar para identificar os limites de cada microsserviço de negócios, além de seu modelo de domínio e seus dados relacionados. Você sempre deve tentar minimizar o acoplamento entre esses microsserviços. Mais adiante, este guia apresentará mais detalhes sobre essa identificação e esse design de modelo de domínio na seção [Identificando limites de modelo de domínio para cada microsserviço](#).

## Desafio #2: Como criar consultas que recuperam dados de vários microsserviços

Um segundo desafio é como implementar consultas que recuperam dados de vários microsserviços, evitando a comunicação intensa dos aplicativos clientes remotos com os microsserviços. Um exemplo pode ser uma única tela de um aplicativo móvel que precisa mostrar microsserviços de informações do usuário que pertencem ao carrinho de compras, de catálogo e de identidade do usuário. Outro exemplo seria um relatório complexo envolvendo diversas tabelas localizadas em vários microsserviços. A solução certa depende da complexidade das consultas. Mas em qualquer caso, será necessária uma maneira de agregar informações se você desejar melhorar a eficiência nas comunicações do sistema. As soluções mais comuns são as seguintes.

**Gateway de API.** Para a agregação de dados simples de vários microsserviços que têm bancos de dados diferentes, a abordagem recomendada é um microsserviço de agregação, conhecido como Gateway de API. No entanto, você precisa ter cuidado ao implementar esse padrão, porque ele pode ser um ponto de redução no sistema e pode violar o princípio de autonomia dos microsserviços. Para atenuar essas possibilidades, você pode ter vários Gateways de API refinados, cada um concentrado em uma "fazenda" vertical ou em uma área de negócios do sistema. O padrão do Gateway de API será explicado mais detalhadamente na seção [Gateway de API](#) mais adiante.

**CQRS com tabelas de consulta/leituras.** Outra solução para agregar dados de vários microsserviços é o [Padrão de exibição materializada](#). Nessa abordagem, você gera com antecedência (prepara os dados desnormalizados antes que as consultas reais ocorram) uma tabela somente leitura com os dados que pertencem a vários microsserviços. A tabela tem um formato adequado às necessidades do aplicativo cliente.

Considere algo como tela de um aplicativo móvel. Se houver um banco de dados individual, você poderá reunir os

dados para essa tela usando uma consulta SQL que execute uma junção complexa envolvendo várias tabelas. No entanto, quando houver vários bancos de dados, e cada banco de dados pertencer a um microsserviço diferente, você não poderá consultar esses bancos de dados e criar uma junção SQL. Sua consulta complexa se tornará um desafio. Você pode atender ao requisito usando uma abordagem de CQRS, ou seja, criar uma tabela desnormalizada em outro banco de dados que é usado apenas para consultas. A tabela pode ser desenvolvida especificamente para os dados necessários para essa consulta complexa, com uma relação um-para-um entre os campos necessários para a tela do aplicativo e as colunas na tabela de consulta. Ela também pode funcionar para a geração de relatórios.

Além de resolver o problema original (como fazer a consulta e a junção entre os microsserviços), essa abordagem também melhora o desempenho consideravelmente em comparação com uma junção complexa, porque os dados que o aplicativo precisa já estão na tabela de consulta. É claro que, o uso de CQRS (segregação de responsabilidade de comando e consulta) com tabelas de consulta/leituras significa um trabalho de desenvolvimento adicional e a necessidade de abranger uma coerência eventual. No entanto, o CQRS com vários bancos de dados deve ser aplicado para atender requisitos de desempenho e de alta escalabilidade em [cenários de colaboração](#) (ou cenários de concorrência, dependendo do ponto de vista).

**"Dados frios" em bancos de dados centrais.** Para relatórios e consultas complexos que talvez não exijam dados em tempo real, uma abordagem comum é exportar os "dados quentes" (dados transacionais dos microsserviços) como "dados frios" para bancos de dados grandes usados somente para relatórios. Esse sistema de banco de dados central pode ser um sistema baseado em Big Data, como o Hadoop, um data warehouse como um baseado no SQL Data Warehouse do Azure ou até mesmo um Banco de Dados SQL individual usado apenas para relatórios (se o tamanho não for problema).

Tenha em mente que esse banco de dados centralizado deve ser usado somente para consultas e relatórios que não precisam de dados em tempo real. As atualizações e transações originais, ou seja, a fonte confiável, precisam estar nos dados dos microsserviços. A maneira de sincronizar os dados seria usar a comunicação controlada por evento (abordada nas próximas seções) ou usar outras ferramentas de importação/exportação da infraestrutura do banco de dados. Se você usar a comunicação controlada por evento, esse processo de integração será semelhante à maneira de propagar dados já descrita para tabelas de consulta de CQRS.

No entanto, se o design do aplicativo envolver a agregação constante de informações de vários microsserviços para consultas complexas, esse poderá ser um sintoma de design ruim, pois um microsserviço deve estar o mais isolado possível dos outros microsserviços. (Isso exclui relatórios/análises que sempre devem usar bancos de dados do Cold Data Central.) Ter esse problema muitas vezes pode ser um motivo para mesclar os microserviços. Você precisa equilibrar a autonomia de evolução e a implantação de cada microsserviço com dependências fortes, coesão e agregação de dados.

## Desafio #3: Como obter consistência entre vários microsserviços

Como mencionado anteriormente, os dados pertencentes a cada microsserviço são privados desse microsserviço e só podem ser acessados usando a API desse microsserviço. Portanto, um desafio apresentado é como implementar processos de negócios de ponta a ponta, mantendo a consistência entre vários microsserviços.

Para analisar o problema, vamos examinar um exemplo do [Aplicativo de referência eShopOnContainers](#). O microsserviço de catálogo mantém informações sobre todos os produtos, incluindo o preço do produto. O microsserviço Carrinho de compras gerencia dados temporais sobre itens de produto que os usuários estão adicionando aos seus carrinhos de compras, o que inclui o preço dos itens no momento em que eles foram adicionados ao carrinho de compras. Quando o preço de um produto é atualizado no catálogo, esse preço também deve ser atualizado nos carrinhos de compras ativos que contêm o mesmo produto e, além disso, o sistema provavelmente deve avisar o usuário, informando que o preço de um determinado item foi alterado desde que o usuário o adicionou ao seu carrinho de compras.

Em uma versão monolítica hipotética deste aplicativo, quando o preço é alterado na tabela de produtos, o subsistema de catálogo pode simplesmente usar uma transação ACID para atualizar o preço atual na tabela de

Carrinho de compras.

No entanto, em um aplicativo baseado em microsserviços, as tabelas de Produto e Carrinho de compras pertencem aos seus respectivos microsserviços. Um microsserviço nunca deve incluir tabelas/armazenamento pertencentes a outro microsserviço em suas próprias transações, nem mesmo em consultas diretas, conforme é mostrado na Figura 4-9.

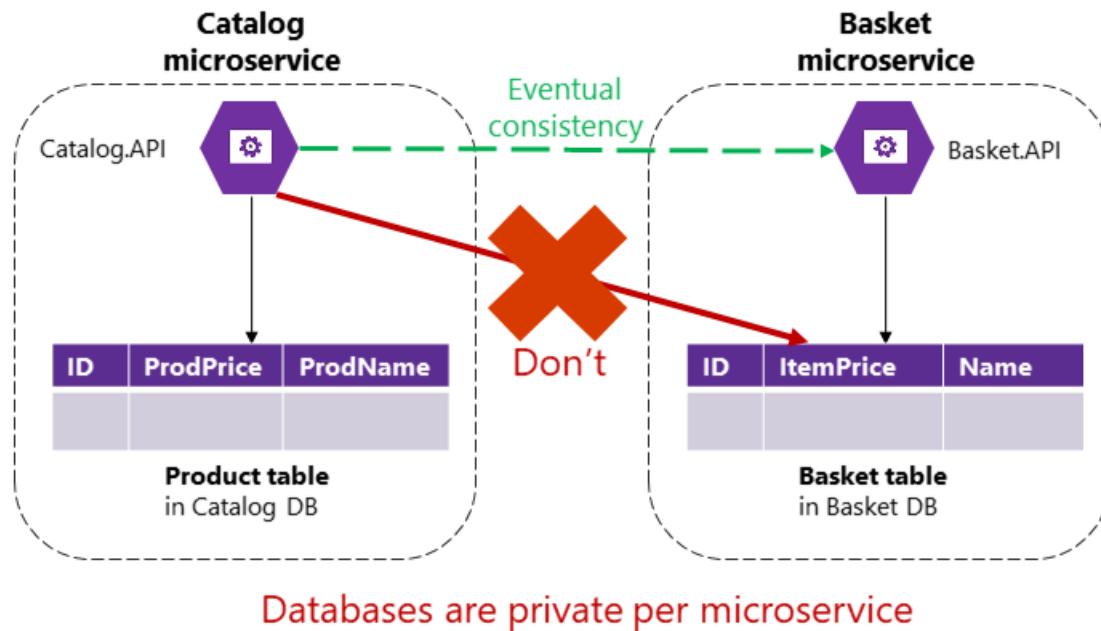


Figura 4-9. Um microsserviço não pode acessar diretamente uma tabela em outro microsserviço

O microsserviço de Catálogo não deve atualizar a tabela de Carrinho de compras diretamente, porque a tabela de Carrinho de compras é de propriedade do microsserviço de Carrinho de compras. Para fazer uma atualização no microsserviço de Carrinho de compras, o microsserviço de Catálogo deve usar a consistência eventual provavelmente com base comunicação assíncrona, assim como em eventos de integração (comunicação baseada em mensagem e em evento). É assim que o aplicativo de referência [eShopOnContainers](#) executa esse tipo de consistência entre microsserviços.

Conforme indicado pelo [Teorema CAP](#), é necessário escolher entre a disponibilidade e a coerência forte de ACID. A maioria dos cenários baseados em microsserviço exigem disponibilidade e alta escalabilidade em vez de coerência forte. Os aplicativos críticos precisam permanecer em funcionamento e os desenvolvedores podem contornar a coerência forte usando técnicas para trabalhar com consistência eventual ou fraca. Essa é a abordagem usada pela maioria das arquiteturas baseadas em microsserviço.

Além de o estilo ACID ou as transações de confirmação de duas fases serem contra os princípios dos microsserviços, a maioria dos bancos de dados NoSQL (como o Azure Cosmos DB, o MongoDB, etc.) não dão suporte às transações de confirmação de duas fases, típicas em cenários de bancos de dados distribuídos. No entanto, manter a consistência dos dados entre os serviços e os bancos de dados é fundamental. Esse desafio também está relacionado à questão de como propagar alterações entre vários microsserviços quando determinados dados precisam ser redundante. Por exemplo, quando o nome ou a descrição do produto precisa estar no microsserviço de catálogo e também no microsserviço de Carrinho de compras.

Uma boa solução para esse problema é usar a consistência eventual entre os microsserviços, articulada pela comunicação controlada por evento e por um sistema de publicação e assinatura. Esses tópicos serão abordados na seção [Comunicação controlada por evento assíncrono](#) mais adiante neste guia.

**Desafio #4:** Como projetar a comunicação entre os limites dos microsserviços

A comunicação entre os limites dos microsserviços é realmente um grande desafio. Nesse contexto, a comunicação não se refere a qual protocolo você deve usar (HTTP e REST, AMQP, mensagens e assim por diante). Nesse caso, ela aborda qual estilo de comunicação você deve usar e, principalmente, que nível de acoplamento os microsserviços devem ter. Dependendo do nível de acoplamento, quando ocorrer uma falha, o impacto dessa falha no sistema poderá variar significativamente.

Em um sistema distribuído, como um aplicativo baseado em microsserviços, com um número muito grande de artefatos em movimentação, com serviços distribuídos em vários servidores ou hosts, eventualmente, os componentes falharão. Falhas parciais e interrupções ainda maiores certamente ocorrerão, portanto, você precisa projetar seus microsserviços e a comunicação entre eles levando em conta os riscos comuns nesse tipo de sistema distribuído.

Uma abordagem comum é implementar microsserviços baseados em HTTP (REST), devido à simplicidade. Uma abordagem baseada em HTTP é perfeitamente aceitável e, nesse caso, o problema está relacionado à maneira de usá-la. Se você usar solicitações e respostas HTTP apenas para que os aplicativos cliente ou os gateways de API interajam com os microsserviços, tudo bem. Mas, se você criar longas cadeias de chamadas HTTP síncronas entre os microsserviços, comunicando-se entre seus limites como se os microsserviços fossem objetos em um aplicativo monolítico, seu aplicativo acabará tendo problemas.

Por exemplo, imagine que o aplicativo cliente faça uma chamada HTTP à API para um microsserviço individual, como o microsserviço de pedidos. Se o microsserviço de pedidos, por sua vez, chamar microsserviços adicionais usando HTTP no mesmo ciclo de solicitação/resposta, você estará criando uma cadeia de chamadas HTTP. Parece razoável em um primeiro momento. No entanto, existem pontos importantes a serem considerados ao adotar esse caminho:

- Bloqueio e baixo desempenho. Devido à natureza síncrona de HTTP, a solicitação original não obtém uma resposta até que todas as chamadas HTTP internas sejam concluídas. Imagine se o número dessas chamadas aumentar significativamente e ao mesmo tempo uma das chamadas HTTP intermediárias para um microsserviço for bloqueada. O resultado é que o desempenho será afetado e a escalabilidade geral será extremamente afetada com o aumento de solicitações HTTP adicionais.
- Acoplando microsserviços com HTTP. Os microsserviços empresariais não devem ser acoplados com outros microsserviços empresariais. O ideal é que eles não "saibam" da existência de outros microsserviços. Se seu aplicativo depender do acoplamento de microsserviços como no exemplo, será praticamente impossível alcançar a autonomia de cada microsserviço.
- Falha em um dos microsserviços. Se você implementar uma cadeia de microsserviços vinculados por chamadas HTTP, quando um dos microsserviços falhar (e, com certeza, isso vai ocorrer) toda a cadeia de microsserviços falhará. Um sistema baseado em microsserviço deve ser criado para continuar a trabalhar da melhor maneira possível durante falhas parciais. Mesmo se você implementar uma lógica do cliente que use novas tentativas com retirada exponencial ou com mecanismos de disjuntor, quanto mais complexas as cadeias de chamadas HTTP forem, mais complexa será a implementação de uma estratégia de falha baseada em HTTP.

Na verdade, se os microsserviços internos estiverem se comunicando por meio da criação de cadeias de solicitações HTTP, conforme descrito, será possível argumentar que você tem um aplicativo monolítico, mas baseado em HTTP entre processos e não em mecanismos de comunicação entre processos.

Portanto, para impor a autonomia do microsserviço e melhorar a resiliência, você deve minimizar o uso de cadeias de comunicação de solicitação/resposta entre os microsserviços. É recomendável usar somente a interação assíncrona para a comunicação entre os microsserviços, seja usando a comunicação assíncrona baseada em evento e em mensagem ou usando a sondagem de HTTP (assíncrona), independentemente do ciclo de solicitação/resposta HTTP original.

O uso da comunicação assíncrona será explicado em detalhes mais adiante neste guia nas seções [A integração assíncrona dos microsserviços impõe a autonomia do microsserviço](#) e [Comunicação assíncrona baseada em](#)

mensagem.

## Recursos adicionais

- Teorema CAP  
[https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem)
- Consistência eventual  
[https://en.wikipedia.org/wiki/Eventual\\_consistency](https://en.wikipedia.org/wiki/Eventual_consistency)
- Primer de consistência de dados  
[/previous-versions/msp-n-p/dn589800\(v=pandp.10\)](https://previous-versions/msp-n-p/dn589800(v=pandp.10))
- Martin Fowler. CQRS (Separação das Operações de Comando e de Consulta)  
<https://martinfowler.com/bliki/CQRS.html>
- Exibição materializada  
[/azure/architecture/patterns/materialized-view](https://azure/architecture/patterns/materialized-view)
- Carlos linha. ACID versus BASE: a pH de mudança do processamento de transações de banco de dados  
<https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/>
- Transação de compensação  
[/azure/architecture/patterns/compensating-transaction](https://azure/architecture/patterns/compensating-transaction)
- Udi Dahan. Composição orientada por serviço  
<https://udidahan.com/2014/07/30/service-oriented-composition-with-video/>

[ANTERIOR](#)

[AVANÇAR](#)

# Identificar os limites de modelo de domínio de cada microsserviço

18/03/2020 • 13 minutes to read • [Edit Online](#)

A meta de identificar os limites do modelo e o tamanho de cada microsserviço não é obter a separação mais granular possível, embora você deva tender a usar microsserviços pequenos se possível. Em vez disso, sua meta deve ser obter a separação mais significativa orientada pelo seu conhecimento do domínio. A ênfase não está no tamanho, mas em funcionalidades empresariais. Além disso, se for necessária uma clara coesão para uma determinada área do aplicativo com base em um grande número de dependências, isso será a indicação da necessidade de um único microsserviço também. A coesão é uma maneira de identificar como separar ou agrupar microsserviços. Por fim, enquanto você obtém mais conhecimento sobre o domínio, deve adaptar o tamanho do seu microsserviço iterativamente. Localizar o tamanho correto não é um processo único.

**SAM Newman**, um conhecido promotor de microsserviços e autor do livro [Building microservices](#) (Criando microsserviços), realça o que você deve projetar seus microsserviços com base no padrão de BC (Contexto Limitado) (parte do controlado por domínio design), conforme apresentado anteriormente. Às vezes, um BC poderia ser composto por vários serviços físicos, mas não vice-versa.

Um modelo de domínio com entidades de domínio específicas dentro de um BC ou microsserviço concreto. Um BC delimita a aplicabilidade de um modelo de domínio e dá aos membros da equipe do desenvolvedor uma compreensão clara e compartilhada sobre o que deve ser coeso e o que pode ser desenvolvido de modo independente. Essas são as mesmas metas de microsserviços.

Outra ferramenta que informa a sua escolha de design é a [lei de Conway](#), que diz que um aplicativo refletirá os limites sociais da organização que o produziu. Porém, às vezes, o oposto é verdadeiro: a organização da empresa é formada pelo software. Talvez você precise reverter a lei de Conway e criar os limites de compilação da maneira que deseja que a empresa seja organizada, inclinada em direção à consultoria do processo empresarial.

Para identificar contextos limitados, você pode usar um padrão DDD chamado de [padrão de mapeamento de contexto](#). Com Mapeamento de Contexto, você identifica os vários contextos no aplicativo e seus limites. É comum haver um contexto e um limite diferentes para cada subsistema pequeno, por exemplo. O Mapa de Contexto é uma maneira de definir e explicitar esses limites entre domínios. Um BC é autônomo e inclui os detalhes de um único domínio (detalhes como as entidades de domínio) e define os contratos de integração com outros BCs. Isso é semelhante à definição de um microsserviço: é autônomo, implementa determinadas funcionalidades de domínio e deve fornecer interfaces. É por isso que o Mapeamento de Contexto e o padrão de Contexto Limitado são boas abordagens para identificar os limites de modelo de domínio dos seus microsserviços.

Ao criar um aplicativo grande, você verá como seu modelo de domínio pode ser fragmentado; um especialista de domínio do domínio de catálogo nomeará as entidades de maneira diferente nos domínios de catálogo e de inventário que um especialista em domínio de envio, por exemplo. Ou a entidade de usuário de domínio pode ser diferente em termos de tamanho e número de atributos ao lidar com um especialista do CRM que queira armazenar todos os detalhes sobre o cliente do que ao lidar com um especialista de domínio de ordenação que precisa apenas de dados parciais sobre o cliente. É muito difícil resolver a ambiguidade de todos os termos do domínio em todos os domínios relacionados a um aplicativo grande. Mas o mais importante é que você não deve tentar unificar os termos. Em vez disso, aceite as diferenças e riqueza fornecida pelos domínios individuais. Se você tentar obter um banco de dados unificado para o aplicativo inteiro, tentativas de um vocabulário unificado serão inadequadas e não soarão bem a nenhum dos vários especialistas de domínio. Portanto, BCs (implementados como microsserviços) ajudarão a esclarecer em que locais você pode usar determinados termos do domínio e em que locais precisará dividir o sistema e criar BCs adicionais com domínios diferentes.

Você saberá que obteve os limites e tamanhos certos de cada BC e modelo de domínio se tiver alguma relação forte entre modelos de domínio e normalmente não precisará mesclar informações de vários modelos de domínio ao executar um operações de aplicativo típicas.

Talvez a melhor resposta à pergunta de quanto grande um modelo de domínio para cada microsserviço deve ser é a seguinte: ele deve ter um BC autônomo, o mais isolado possível, que lhe permita trabalhar sem precisar mudar constantemente para outros contextos (outros modelos do microsserviço). Na Figura 4-10, você pode ver como cada um dos vários microsserviços (vários BCs) tem seus próprios modelos e como suas entidades podem ser definidas, dependendo dos requisitos específicos para cada um dos domínios identificados em seu aplicativo.

## Identifying a Domain Model per Microservice or Bounded Context

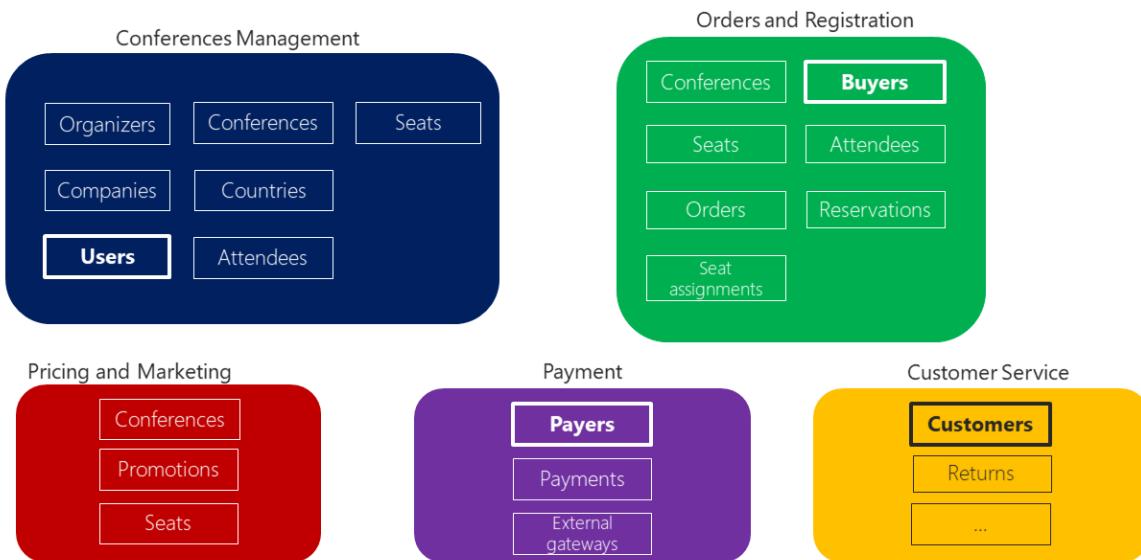


Figura 4-10. Identificação de entidades e limites de modelo de microsserviço

A Figura 4-10 ilustra um cenário de exemplo relacionado a um sistema de gerenciamento de conferência online. A mesma entidade aparece como "Usuários", "Compradores", "Pagadores" e "Clientes" dependendo do contexto limitado. Você identificou vários BCs que podem ser implementados como microsserviços, com base em domínios que especialistas de domínio definiram para você. Como você pode ver, há entidades que estão presentes apenas em um único modelo de microsserviço, como Pagamentos no microsserviço de Pagamento. Esses serão fáceis de implementar.

No entanto, você também pode ter entidades com diferentes formas, mas que compartilhem de uma mesma identidade em vários modelos de domínio de vários microsserviços. Por exemplo, a entidade de Usuário é identificada no microsserviço Gerenciamento de Conferências. Esse mesmo usuário, com a mesma identidade, é o chamado Compradores no microsserviço de Ordenação ou o chamado Pagante no microsserviço de Pagamento e, até mesmo, o chamado Cliente no microsserviço de Atendimento ao Cliente. Isso ocorre porque, dependendo da [linguagem ubíqua](#) que cada especialista em domínio está usando, um usuário pode ter uma perspectiva diferente mesmo com atributos diferentes. A entidade de usuário no modelo de microsserviço denominado Gerenciamento de Conferências pode ter a maioria dos seus atributos de dados pessoais. No entanto, esse mesmo usuário na forma Pagante no microsserviço Pagamento ou na forma de Cliente no microsserviço Atendimento ao Cliente talvez não precise da mesma lista de atributos.

Uma abordagem similar é ilustrada na Figura 4-11.

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

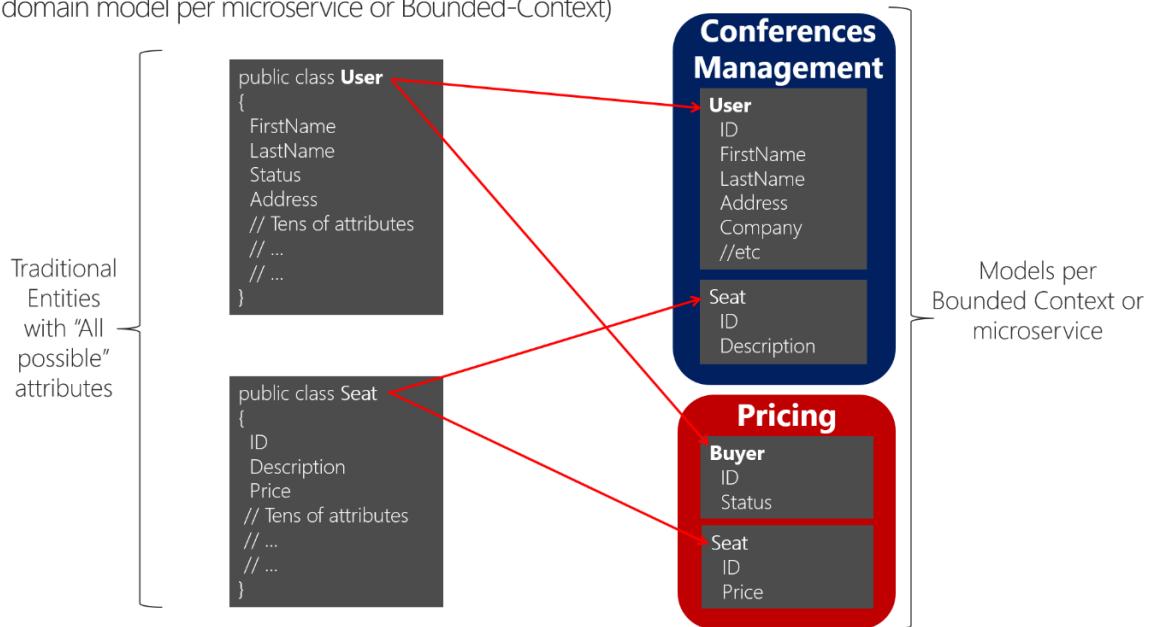


Figura 4-11. Deco<sup>r</sup>po<sup>s</sup>o de modelos de dados tradicionais em v<sup>á</sup>rios modelos de dom<sup>í</sup>nio

Ao decompor um modelo de dados tradicional entre contextos limitados, você pode ter entidades diferentes que compartilham a mesma identidade (um comprador também é um usuário) com atributos diferentes em cada contexto limitado. Você pode ver como o usuário está presente no modelo de microsserviço de Gerenciamento de Conferências como a entidade de Usuário e também está presente na forma de entidade de Comprador no microsserviço Preços, com atributos ou detalhes alternativos sobre o usuário quando ele é de fato um Comprador. Cada microsserviço ou BC pode não precisar de todos os dados relacionados a uma entidade de Usuário, apenas parte deles, dependendo do problema ou do contexto a resolver. Por exemplo, no modelo de microsserviço de Preços, não é necessário o endereço nem o nome do usuário, apenas a ID (como identidade) e o Status, que afetarão os descontos ao definir os preços das estações por comprador.

A entidade Seat tem o mesmo nome, mas diferentes atributos em cada modelo de domínio. No entanto, Seat compartilha identidade com base na mesma ID, conforme acontece com Usuário e Comprador.

Basicamente, há um conceito compartilhado de um usuário que existe em vários serviços (domínios), que compartilham todos da identidade do usuário. Mas, em cada modelo de domínio, pode haver detalhes adicionais ou diferentes sobre a entidade de usuário. Portanto, precisa haver uma maneira de mapear uma entidade de usuário de um domínio (microsserviço) para outro.

Existem vários benefícios em não compartilhar a mesma entidade de usuário com o mesmo número de atributos entre domínios. Um benefício é reduzir a duplicação para que os modelos de microsserviço não tenham nenhum dado de que não precisem. Outro benefício é ter um microsserviço mestre que detém um determinado tipo de dados por entidade de modo que atualizações e consultas para esse tipo de dados sejam controladas apenas por esse microsserviço.

PRÓXIMO

ANTERIOR

# Padrão de gateway de API versus comunicação direta de cliente com microsserviço

10/09/2020 • 30 minutes to read • [Edit Online](#)

Em uma arquitetura de microsserviços, cada microsserviço expõe um conjunto de pontos de extremidade (tipicamente) refinados. Esse fato pode afetar a comunicação de cliente com microsserviço, conforme explicado nesta seção.

## Comunicação direta de cliente com microsserviço

Uma abordagem possível é usar uma arquitetura de comunicação direta de cliente com microsserviço. Nessa abordagem, um aplicativo cliente pode fazer solicitações diretamente para alguns dos microsserviços, conforme mostra a Figura 4-12.

### Direct Client-To-Microservice communication Architecture

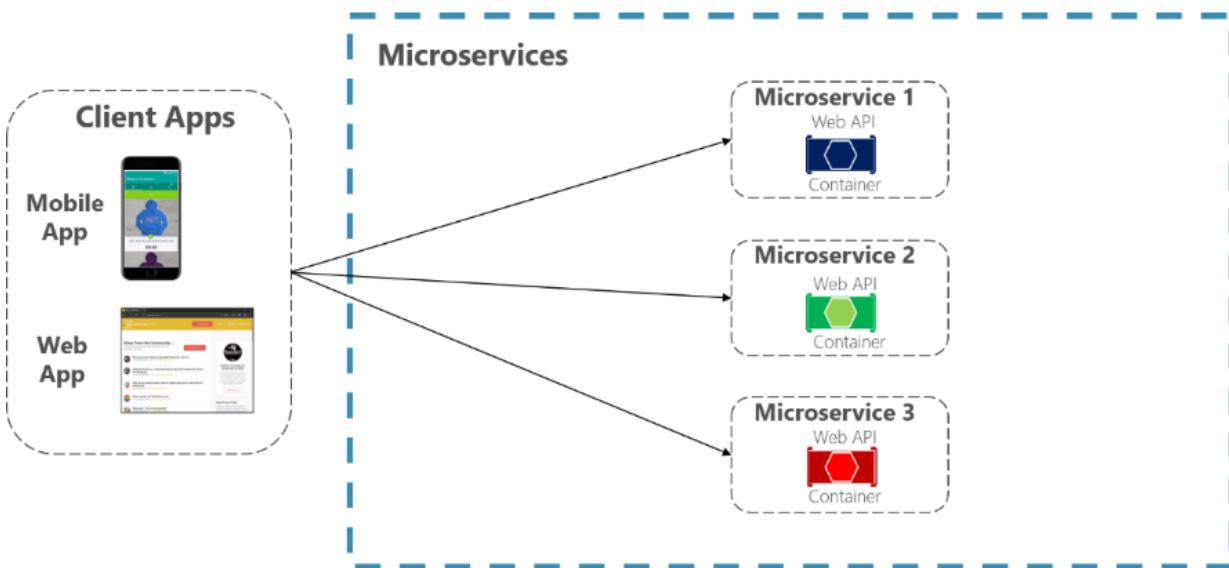


Figura 4-12. Usando uma arquitetura de comunicação direta de cliente com microsserviço

Nessa abordagem, cada microsserviço tem um ponto de extremidade público, às vezes, com uma porta TCP diferente para cada microsserviço. Um exemplo de uma URL para um serviço específico pode ser a URL a seguir no Azure:

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

Em um ambiente de produção com base em um cluster, essa URL mapearia para o balanceador de carga usado no cluster, que, por sua vez, distribui as solicitações entre os microsserviços. Em ambientes de produção, você pode ter um ADC (Controlador de Entrega de Aplicativo) como o [Gateway de Aplicativo do Azure](#) entre os microsserviços e a Internet. Isso funciona como uma camada transparente que não executa balanceamento de carga, mas protege seus serviços ao oferecer terminação SSL. Isso aumenta a carga de seus hosts descarregando terminação SSL com uso intensivo de CPU e outras tarefas de roteamento para o Gateway de Aplicativo do Azure. Em qualquer caso, um balanceador de carga e ADC são transparentes de um ponto de vista da arquitetura do aplicativo lógico.

Uma arquitetura de comunicação direta de cliente com microsserviço pode ser boa o bastante para um aplicativo baseado em microsserviço pequeno, especialmente se o aplicativo cliente for um aplicativo Web do lado do servidor como um aplicativo ASP.NET MVC. No entanto, quando você cria aplicativos grandes e complexos baseados em microsserviço (por exemplo, ao lidar com dezenas de tipos de microsserviço), e especialmente quando os aplicativos cliente são aplicativos móveis remotos ou aplicativos Web do SPA, essa abordagem enfrenta alguns problemas.

Considere as seguintes questões ao desenvolver um aplicativo grande com base em microsserviços:

- *Como os aplicativos cliente podem minimizar o número de solicitações para o back-end e reduzir a comunicação excessiva com vários microsserviços?*

Interagir com vários microsserviços para criar uma única tela de interface do usuário aumenta o número de viagens de ida e volta pela Internet. Isso aumenta a latência e a complexidade do lado da interface do usuário. O ideal é que as respostas sejam agregadas com eficiência no lado do servidor. Isso reduz a latência, já que várias partes de dados voltam em paralelo e algumas interfaces do usuário podem mostrar dados assim que eles estão prontos.

- *Como você pode lidar com preocupações abrangentes, como autorização, transformações de dados e despacho de solicitação dinâmica?*

Implementar preocupações relativas a segurança e abrangentes, como segurança e autorização em cada microsserviço, pode exigir um esforço significativo de desenvolvimento. Uma abordagem possível é ter esses serviços no host do Docker ou no cluster interno para restringir o acesso direto a eles do lado de fora e implementar esses interesses paralelos em um local centralizado, como um Gateway de API.

- *Como os aplicativos cliente podem se comunicar com serviços que usam protocolos não amigáveis para a Internet?*

Protocolos usados no lado do servidor (como AMQP ou protocolos binários) geralmente não são compatíveis com aplicativos cliente. Portanto, as solicitações devem ser executadas por meio de protocolos como HTTP/HTTPS e convertidas em outros protocolos posteriormente. Uma abordagem *man-in-the-middle* pode ajudar nessa situação.

- *Como é possível moldar uma fachada feita especialmente para aplicativos móveis?*

A API de vários microsserviços podem não ser bem projetadas para as necessidades de diferentes aplicativos cliente. Por exemplo, as necessidades de um aplicativo móvel podem ser diferentes das necessidades de um aplicativo Web. Para aplicativos móveis, convém otimizar ainda mais para que as respostas de dados possam ser mais eficientes. Você pode fazer isso agregando dados de vários microsserviços e retornando um único conjunto de dados e, às vezes, eliminando os dados na resposta que não são necessários para o aplicativo móvel. E, claro, você pode compactar dados. Novamente, uma fachada ou uma API entre o aplicativo móvel e os microsserviços pode ser conveniente para esse cenário.

## Por que considerar o uso de Gateways de API em vez de comunicação direta de cliente com microsserviço

Em uma arquitetura de microsserviços, os aplicativos cliente normalmente precisam consumir a funcionalidade de mais de um microsserviço. Se esse consumo for executado diretamente, o cliente precisará manipular várias chamadas para terminais de microsserviço. O que acontece quando o aplicativo evolui e novos microsserviços são introduzidos ou os microsserviços existentes são atualizados? Se o seu aplicativo tiver muitos microsserviços, lidar com tantos pontos de extremidade dos aplicativos cliente pode ser um pesadelo. Como o aplicativo cliente seria acoplado a esses pontos de extremidade internos, a evolução dos microsserviços no futuro pode causar alto impacto aos aplicativos cliente.

Portanto, ter um nível intermediário ou indireto (Gateway) pode ser muito conveniente para aplicativos baseados

em microsserviço. Se você não tiver Gateways de API, os aplicativos do cliente deverão enviar solicitações diretamente aos microsserviços, o que causará problemas como os seguintes:

- **Acoplamento:** sem o padrão de Gateway de API, os aplicativos cliente são acoplados aos microsserviços internos. Os aplicativos clientes precisam saber como as várias áreas do aplicativo são decompostas em microsserviços. Ao evoluir e refatorar os microserviços internos, essas ações impactam a manutenção porque causam alterações significativas nos aplicativos cliente devido à referência direta aos microsserviços internos dos aplicativos cliente. Os aplicativos clientes precisam ser atualizados com frequência, o que dificulta a evolução da solução.
- **Muitas viagens de ida e volta:** uma única página/tela no aplicativo cliente pode exigir diversas chamadas para vários serviços. Isso pode resultar em várias viagem de ida e volta na rede entre o cliente e o servidor, adicionando latência significativa. A agregação manipulada em um nível intermediário pode melhorar o desempenho e a experiência do usuário para o aplicativo cliente.
- **Problemas de segurança:** sem um gateway, todos os microsserviços precisam ser expostos ao "mundo externo", tornando a superfície de ataque maior do que se você ocultar microsserviços internos que não são usados diretamente pelos aplicativos cliente. Quanto menor a superfície de ataque, mais segura sua aplicação pode ser.
- **Preocupações abrangentes:** cada microserviço publicado publicamente deve lidar com questões como autorização e SSL. Em muitas situações, esses interesses podem ser tratados em uma única camada para que os microsserviços internos sejam simplificados.

## Por que o padrão de Gateway de API?

Quando você projeta e cria aplicativos grandes ou complexos baseado em microsserviço com vários aplicativos cliente, uma boa abordagem a considerar pode ser um [Gateway de API](#). Esse é um serviço que fornece um único ponto de entrada para determinados grupos de microsserviços. É semelhante ao [padrão fachada](#) do design orientado a objeto, mas, nesse caso, faz parte de um sistema distribuído. O padrão de Gateway de API às vezes também é conhecido como [BFF](#) ("back-end para front-end"), porque ele é criado pensando nas necessidades do aplicativo cliente.

Portanto, o gateway da API fica entre os aplicativos cliente e os microsserviços. Ele atua como um proxy reverso, encaminhando as solicitações de clientes para serviços. Ele também pode fornecer outros recursos paralelos, como autenticação, terminação SSL e cache.

A Figura 4-13 mostra como um Gateway de API personalizado pode se encaixar em uma arquitetura simplificada baseada em microsserviço com apenas alguns microsserviços.

# Using a single custom API Gateway service

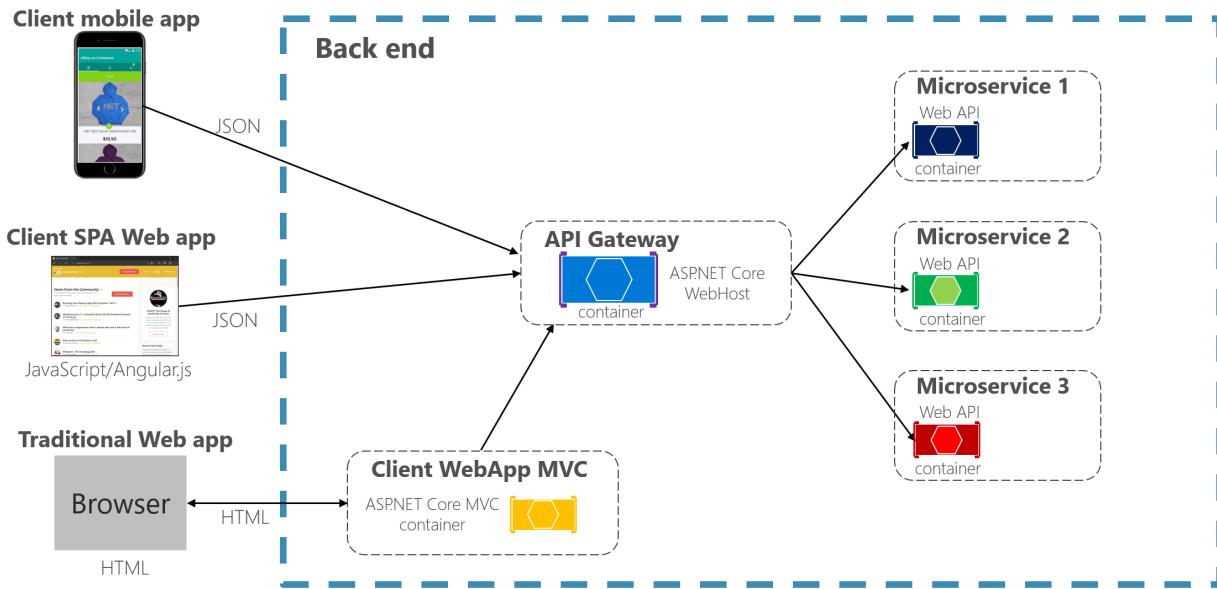


Figura 4-13. Uso de um Gateway de API implementado como um serviço personalizado

Os aplicativos se conectam a um único ponto de extremidade, o gateway de API, que é configurado para encaminhar solicitações a microserviços individuais. Neste exemplo, o Gateway de API deve ser implementado como um serviço de WebHost ASPNET Core personalizado em execução como um contêiner.

É importante destacar que, neste diagrama, você usa um único serviço personalizado de Gateway de API, voltado para vários aplicativos cliente diferentes. Esse fato pode ser um risco importante, pois seu serviço de Gateway de API estará crescendo e em evolução com base em vários requisitos diferentes dos aplicativos cliente. Por fim, ele será inflado por causa dessas diferentes necessidades e efetivamente pode ser semelhante a um aplicativo monolítico ou serviço monolítico. É por isso que é muito recomendável dividir o Gateway de API em vários serviços ou vários Gateways de API menores, um por tipo de fator forma de aplicativo cliente, por exemplo.

É necessário ter cuidado ao implementar o padrão de Gateway de API. Geralmente, não é recomendável ter um único Gateway de API para agregar todos os microserviços internos do seu aplicativo. Em caso afirmativo, ele atua como um agregador ou orquestrador monolítico e viola a autonomia de microserviço acoplando todos os microserviços.

Portanto, os Gateways de API devem ser segregados com base nos limites de negócios e nos aplicativos cliente, e não agir como um agregador único para todos os microserviços internos.

Ao dividir a camada do Gateway de API em vários Gateways de API, se o aplicativo tiver vários aplicativos cliente, isso poderá ser um fator fundamental ao identificar os vários tipos de Gateways de API, de modo que você possa ter uma fachada distinta para as necessidades de cada aplicativo cliente. Esse caso é um padrão chamado "back-end para front-end" (BFF), em que cada gateway de API pode fornecer uma API diferente adaptada para cada tipo de aplicativo cliente, possivelmente mesmo com base no fator forma de cliente implementando um código de adaptador específico que, sob a chamada de vários microserviços internos, conforme mostrado na imagem a seguir:

# Using multiple API Gateways / BFF

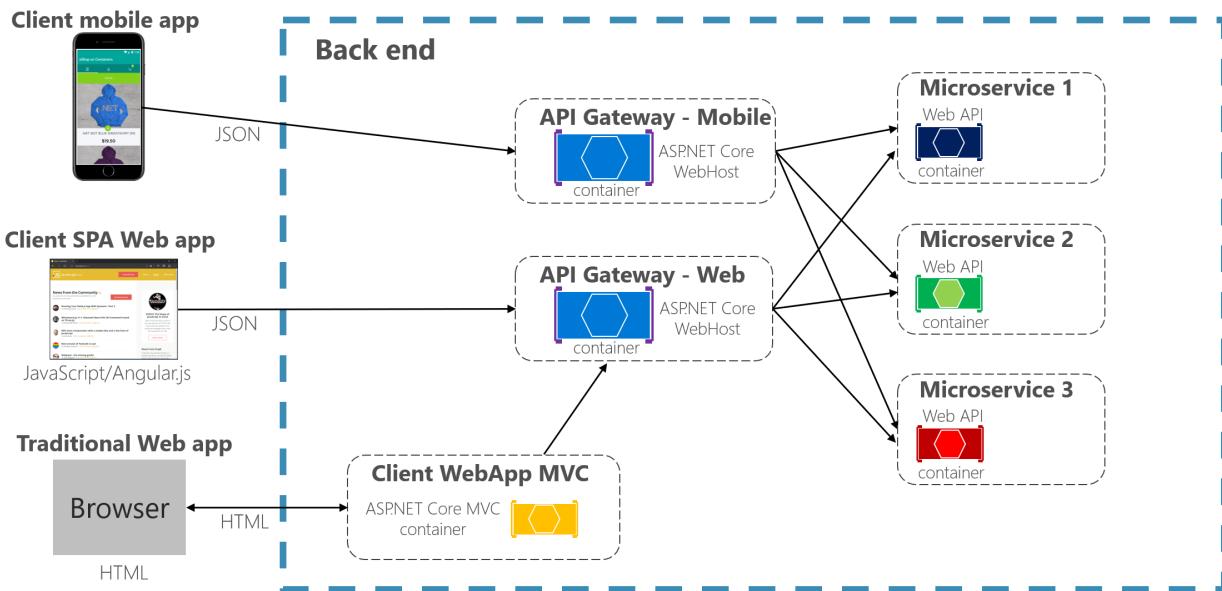


Figura 4-13.1. Uso de mostrando vários Gateways de API personalizados

Figura 4-13.1 mostra os gateways de API que são separados por tipo de cliente; um para clientes móveis e outro para clientes Web. Um aplicativo Web tradicional se conecta a um microsserviço MVC que usa o Gateway de API Web. O exemplo ilustra uma arquitetura simplificada com vários gateways de API refinados. Nesse caso, os limites identificados para cada Gateway de API são baseados puramente no padrão BFF ("Back-end para Front-end"), portanto, baseados apenas na API necessária por aplicativo cliente. Porém, em aplicativos maiores, você deve ir além e criar Gateways de API adicionais com base nos limites de negócios como um segundo fator de design.

## Principais recursos do padrão de Gateway de API

Um Gateway de API pode oferecer vários recursos. Dependendo do produto, ele pode oferecer recursos mais avançados ou mais simples, no entanto, os recursos mais importantes e fundamentais para qualquer Gateway de API são os seguintes padrões de design:

**Proxy reverso ou roteamento de gateway.** O Gateway de API oferece um proxy reverso para redirecionar ou encaminhar solicitações (roteamento da camada 7, geralmente solicitações HTTP) para os pontos de extremidade dos microsserviços internos. O gateway fornece um único ponto de extremidade ou URL para os aplicativos clientes e, em seguida, mapeia internamente as solicitações para um grupo de microsserviços internos. Esse recurso de roteamento ajuda a desacoplar os aplicativos cliente dos microsserviços, mas também é conveniente ao modernizar uma API monolítica posicionando o gateway de API entre a API monolítica e os aplicativos cliente, então você pode adicionar novas APIs como novos microsserviços enquanto ainda usa a API monolítica herdada até que ela seja dividida em muitos microsserviços no futuro. Devido ao Gateway de API, os aplicativos cliente não notarão se as APIs em uso forem implementadas como microsserviços internos ou uma API monolítica. O mais importante é o fato que, ao evoluir e refatorar a API monolítica em microsserviços, graças ao roteamento do Gateway de API, os aplicativos cliente não serão afetados por nenhuma alteração de URI.

Para saber mais, confira [Padrão de roteamento do gateway](#).

**Solicitações de agregação.** Como parte do padrão de gateway, é possível agrregar várias solicitações de cliente (geralmente solicitações HTTP) direcionadas a vários microsserviços internos em uma única solicitação de cliente. Esse padrão é especialmente conveniente quando uma página/tela do cliente precisa de informações de vários microsserviços. Com essa abordagem, o aplicativo cliente envia uma única solicitação ao Gateway da API, que envia várias solicitações aos microsserviços internos e, em seguida, agrupa os resultados e envia tudo de volta ao aplicativo cliente. O principal benefício e o objetivo desse padrão de design é reduzir a informação entre os

aplicativos cliente e a API de back-end, que é especialmente importante para aplicativos remotos do datacenter em que os microserviços residem, como aplicativos móveis ou solicitações provenientes de aplicativos de SPA provenientes do JavaScript em navegadores remotos do cliente. Para aplicativos Web comuns que executam as solicitações no ambiente do servidor (como um aplicativo Web ASP.NET Core MVC), esse padrão não é tão importante, já que a latência é muito menor do que em aplicativos cliente remotos.

Dependendo do produto do Gateway de API usado, ele poderá executar essa agregação. No entanto, em muitos casos, é mais flexível criar microsserviços de agregação no escopo do Gateway de API, de modo que você defina a agregação no código (ou seja, o código C#):

Para saber mais, confira o [Padrão de agregação de Gateway](#).

**Interesses paralelos ou descarregamento de gateway.** Dependendo dos recursos oferecidos por cada produto do Gateway de API, você pode descarregar a funcionalidade de microsserviços individuais para o gateway, o que simplifica a implementação de cada microsserviço ao consolidar interesses paralelos em uma camada. Isso é especialmente conveniente para recursos especializados que podem ser complexos de implementar adequadamente em cada microsserviço interno, como as seguintes funcionalidades:

- Autenticação e autorização
- Integração de serviços de descoberta
- Cache de resposta
- Políticas de repetição, disjuntor e QoS
- Limitação de taxa
- Balanceamento de carga
- Registrando em log, rastreamento, correlação
- Cabeçalhos, cadeias de caracteres de consulta e transformação de declarações
- Lista de permissões de IP

Para saber mais, confira o [Padrão de descarregamento de Gateway](#).

## Uso de produtos com recursos do Gateway de API

Pode haver muitos outros interesses paralelos oferecidos pelos produtos de Gateways de API, dependendo de cada implementação. Exploraremos aqui:

- [Gerenciamento de API do Azure](#)
- [Ocelot](#)

### Gerenciamento de API do Azure

O [Gerenciamento de API do Azure](#) (conforme mostrado na Figura 4-14) não apenas soluciona as necessidades do Gateway de API, mas também fornece funcionalidades como coleta de insights das APIs. Se você estiver usando uma solução de gerenciamento de API, um Gateway de API será apenas um componente nessa solução completa de gerenciamento de API.

# API Gateway with Azure API Management

## Architecture

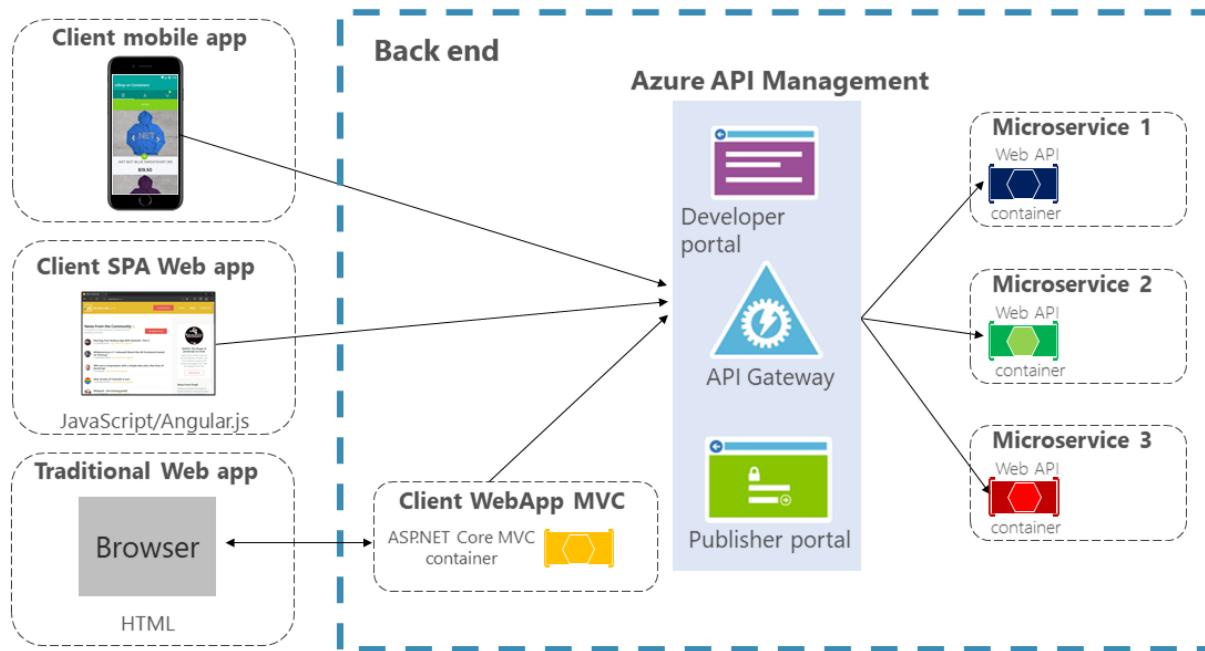


Figura 4-14. Usando o Gerenciamento de API do Azure para o Gateway de API

O gerenciamento de API do Azure resolve o gateway de API e as necessidades de gerenciamento, como log, segurança, medição, etc. Nesse caso, ao usar um produto como o gerenciamento de API do Azure, o fato de que você pode ter um único gateway de API não é tão arriscado, pois esses tipos de gateways de API são "mais finos", o que significa que você não implementa código C# personalizado que poderia evoluir em direção a um componente monolítico.

Os produtos de Gateway de API costumam atuar como um proxy reverso para comunicação de entrada, em que você também pode filtrar as APIs dos microsserviços internos e aplicar a autorização para as APIs publicadas nessa camada única.

As informações disponíveis do sistema Gerenciamento de API ajudam a entender como as suas APIs estão sendo usadas e como elas são executadas. Isso ocorre porque você pode exibir relatórios de análise quase em tempo real e identificar tendências que podem afetar seus negócios. Além disso, você pode ter logs sobre a atividade de solicitação e resposta para mais análises online e offline.

Com o Gerenciamento de API do Azure, você pode proteger suas APIs usando uma chave, um token e filtragem de IP. Esses recursos permitem que você imponha cotas flexíveis e refinadas e limites de taxa, modifique a forma e o comportamento de suas APIs usando políticas e melhore o desempenho com o cache de resposta.

Neste guia e no aplicativo de exemplo de referência (eShopOnContainers), a arquitetura é limitada a uma arquitetura em contêiner mais simples e personalizada para focar em contêineres simples sem o uso de produtos PaaS como Gerenciamento de API do Azure. Mas, para grandes aplicativos baseados em microsserviço implantados no Microsoft Azure, recomendamos que você avaliar o Gerenciamento de API do Azure como base para seus Gateways de API em produção.

### Ocelot

O [Ocelot](#) é um Gateway de API leve, recomendado para abordagens mais simples. O Ocelot é um gateway de API baseado em .NET Core de software livre, especialmente feito para arquiteturas de microsserviços que precisam de pontos unificados de entrada em seus sistemas. É leve, rápido e escalonável e fornece roteamento e autenticação entre muitos outros recursos.

O principal motivo para escolher Ocelot para o [aplicativo de referência eShopOnContainers](#) é porque o Ocelot é

um gateway de API leve do .NET Core que pode ser implantado no mesmo ambiente de implantação de aplicativos em que você está implantando seus microserviços/contêineres, como um host do Docker, kubernetes, etc. E, como ele é baseado no .NET Core, ele é multiplataforma que permite que você implante no Linux ou no Windows.

Os diagramas anteriores que mostram os Gateways de API personalizados em execução nos contêineres correspondem precisamente ao modo como você também pode executar o Ocelot em um contêiner e em um aplicativo baseado em microsserviço.

Além disso, existem muitos outros produtos no mercado que oferecem recursos de Gateways de API, como Apigee, Kong, MuleSoft, WSO2 e outros produtos como Linkerd e Istio para recursos de controlador de entrada de malha de serviço.

Após as seções iniciais de explicação sobre arquitetura e padrões, as próximas seções explicam como implementar Gateways de API com [Ocelot](#).

## Desvantagens do padrão de Gateway de API

- A desvantagem mais importante é que, quando você implementa um Gateway de API, está acoplando essa camada aos microsserviços internos. Um acoplamento como este pode introduzir problemas sérios no seu aplicativo. Clemens Vaster, arquiteto na equipe do Barramento de Serviço do Azure, refere-se a essa possível dificuldade como "o novo ESB" na sessão "[Mensagens e microsserviços](#)" na GOTO 2016.
- Usar um Gateway de API de microsserviços cria um possível ponto único de falha adicional.
- Um Gateway de API pode apresentar um maior tempo de resposta devido à chamada de rede adicional. No entanto, essa chamada extra normalmente causa um menor impacto do que ter uma interface do cliente com excesso de chamadas diretas aos microsserviços internos.
- Se não for dimensionado corretamente, o Gateway de API poderá se tornar um gargalo.
- Um Gateway de API requer custos adicionais de desenvolvimento e manutenção futura se incluir lógica personalizada e agregação de dados. Os desenvolvedores precisam atualizar o Gateway de API para expor os pontos de extremidade de cada microsserviço. Além disso, as alterações de implementação nos microsserviços internos podem causar alterações de código no nível do Gateway da API. No entanto, se o Gateway de API estiver apenas aplicando segurança, logon e controle de versão (como ao usar o Gerenciamento de API do Azure), esse custo de desenvolvimento adicional poderá não se aplicar.
- Se o Gateway de API for desenvolvido por uma única equipe, poderá haver um gargalo de desenvolvimento. Esse é outro motivo pelo qual uma abordagem melhor é ter vários Gateways de API refinados que respondam às diferentes necessidades do cliente. Você também pode separar o Gateway de API internamente em várias áreas ou camadas de propriedade de diferentes equipes trabalhando em microsserviços internos.

## Recursos adicionais

- Chris Richardson. **Padrão: gateway de API/backend para front-end**  
<https://microservices.io/patterns/apigateway.html>
- **Padrão de gateway de API**  
[/azure/architecture/microservices/gateway](https://azure/architecture/microservices/gateway)
- **Padrão de agregação e composição**  
<https://microservices.io/patterns/data/api-composition.html>
- **Gerenciamento de API do Azure**  
<https://azure.microsoft.com/services/api-management/>
- Udi Dahan. **Composição orientada por serviço**

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

- **Grandes Clemenss.** Mensagens e microserviços em GOTO 2016 (vídeo)  
<https://www.youtube.com/watch?v=rXi5CLjIQ9k>
- **Gateway de API em resumo** (ASP.NET Core série de tutoriais de gateway de API)  
<https://www.pogsdotnet.com/2018/08/api-gateway-in-nutshell.html>

[ANTERIOR](#)

[AVANÇAR](#)

# Comunicação em uma arquitetura de microsserviço

18/03/2020 • 21 minutes to read • [Edit Online](#)

Em um aplicativo monolítico em execução em um único processo, os componentes invocam-se usando um método no nível da linguagem ou chamadas de função. Eles poderão ser fortemente acoplados se você estiver criando objetos com código (por exemplo, `new ClassName()`) ou poderão ser invocados de forma desacoplada, se você estiver usando injeção de dependência referenciando abstrações em vez de instâncias concretas de objeto. De qualquer forma, os objetos são executados no mesmo processo. O maior desafio ao passar de um aplicativo monolítico para um aplicativo baseado em microsserviços é alterar o mecanismo de comunicação. Uma conversão direta das chamadas de método internas em processo em chamadas RPC para os serviços causará uma comunicação muito intensa e ineficiente que não terá um bom desempenho em ambientes distribuídos. Os desafios da criação adequada de um sistema distribuído são tão conhecidos que existe um código conhecido como [Fallacies of distributed computing](#) (Enganos da computação distribuída) que lista as suposições que os desenvolvedores geralmente fazem ao passar de designs monolíticos para designs distribuídos.

Não há apenas uma solução, mas várias. Uma das soluções envolve isolar os microsserviços de negócios o máximo possível. Em seguida, usar a comunicação assíncrona entre os microsserviços internos e substituir a comunicação refinada típica na comunicação entre processos dos objetos por uma comunicação menos refinada. Você pode fazer isso agrupando chamadas e retornando para o cliente dados que agregam os resultados de várias chamadas internas.

Um aplicativo baseado em microsserviços é um sistema distribuído em execução em vários processos ou serviços, geralmente, até mesmo em vários servidores ou hosts. Cada instância de serviço geralmente é um processo. Portanto, os serviços devem interagir usando um protocolo de comunicação entre processos, como HTTP, AMQP ou um protocolo binário, como TCP, dependendo da natureza de cada serviço.

A comunidade de microsserviços promove a filosofia de "pontos de extremidade inteligentes e pipes tolos". Este slogan incentiva um design o mais desacoplado possível entre os microsserviços e o mais coeso possível dentro de um único microsserviço. Conforme explicado anteriormente, cada microsserviço tem seus próprios dados e sua própria lógica do domínio. Mas os microsserviços que compõem um aplicativo de ponta a ponta geralmente são coreografados simplesmente usando comunicações REST em vez de protocolos complexos, como WS-\*, e comunicações flexíveis controladas por evento, em vez de orquestradores de negócios e processos centralizados.

Os dois protocolos mais usados são solicitação/resposta HTTP com APIs de recurso (principalmente em consultas) e mensagens assíncronas leves, ao comunicar atualizações entre vários microsserviços. Isso será mais bem explicado nas seções a seguir.

## Tipos de comunicação

O cliente e os serviços podem se comunicar por vários tipos de comunicação diferentes, cada um direcionando a um cenário diferente e a metas diferentes. Inicialmente, esses tipos de comunicação podem ser classificados em dois eixos.

O primeiro eixo define se o protocolo é síncrono ou assíncrono:

- Protocolo síncrono. HTTP é um protocolo síncrono. O cliente envia uma solicitação e espera uma resposta do serviço. Isso é independente da execução do código do cliente que pode ser síncrona (o thread é bloqueado) ou assíncrona (o thread não é bloqueado e a resposta acabará alcançando um retorno de chamada). O ponto importante aqui é que o protocolo (HTTP/HTTPS) é síncrono e o código do cliente somente poderá continuar sua tarefa quando receber a resposta do servidor HTTP.
- Protocolo assíncrono. Outros protocolos, como o AMQP (um protocolo compatível com vários sistemas

operacionais e ambientes de nuvem), usam mensagens assíncronas. O código do cliente ou o remetente da mensagem geralmente não espera uma resposta. Ele apenas envia a mensagem como ao enviar uma mensagem para uma fila do RabbitMQ ou para qualquer outro agente de mensagens.

O segundo eixo define se a comunicação tem um único destinatário ou vários destinatários:

- Único destinatário. Cada solicitação deve ser processada por exatamente um destinatário ou serviço. Um exemplo dessa comunicação é o [Padrão de comando](#).
- Vários destinatários. Cada solicitação pode ser processada por nenhum destinatário, por um ou por vários destinatários. Esse tipo de comunicação precisa ser assíncrono. Um exemplo é o mecanismo [de publicação/assinatura](#) usado em padrões como a [Arquitetura orientada por eventos](#). Ele se baseia em um agente de mensagem ou em uma interface de barramento de evento ao propagar atualizações de dados entre vários microsserviços por meio de eventos. Ele geralmente é implementado por meio de um barramento de serviço ou artefato semelhante, como o [Barramento de Serviço do Azure](#), usando [tópicos e assinaturas](#).

Um aplicativo baseado em microsserviço geralmente usará uma combinação desses estilos de comunicação. O tipo mais comum é a comunicação de único destinatário com um protocolo síncrono como HTTP/HTTPS ao invocar um serviço HTTP de API Web regular. Geralmente os microsserviços também usam protocolos de mensagens para a comunicação assíncrona entre eles.

É bom conhecer esses eixos para esclarecer melhor os mecanismos de comunicação possíveis, mas eles não são as questões mais importantes ao criar microsserviços. Não é a natureza assíncrona da execução de thread de cliente nem a natureza assíncrona do protocolo selecionado os pontos mais importantes ao integrar microsserviços. O que é importante é ser capaz de integrar os microsserviços de forma assíncrona, mantendo a independência dos microsserviços, conforme será explicado na seção a seguir.

## A integração assíncrona dos microsserviços impõe a autonomia do microsserviço

Conforme mencionado, o ponto importante ao criar um aplicativo baseado em microsserviços é a maneira de integrar os microsserviços. O ideal é tentar minimizar a comunicação entre os microsserviços internos. Quanto menos comunicações entre os microsserviços, melhor. Mas, em muitos casos, será necessário integrar os microsserviços de alguma forma. Quando isso for necessário, a regra crítica será que a comunicação entre os microsserviços deve ser assíncrona. Isso não significa que você precisa usar um protocolo específico (por exemplo, um sistema de mensagens assíncrono em vez de HTTP síncrono). Isso apenas significa que a comunicação entre os microsserviços deve ser feita somente pela propagação de dados de forma assíncrona, mas tente não depender de outros microsserviços internos como parte da operação inicial de solicitação/resposta HTTP do serviço.

Se possível, nunca dependa da comunicação síncrona (solicitação/resposta) entre vários microsserviços, nem mesmo para consultas. O objetivo de cada microsserviço é ser autônomo e estar disponível para o consumidor cliente, mesmo se outros serviços que fazem parte do aplicativo de ponta a ponta estiverem inativos ou não íntegros. Se você acha que precisa fazer uma chamada de um microsserviço para outros microsserviços (como executar uma solicitação HTTP para uma consulta de dados) para poder fornecer uma resposta a um aplicativo cliente, você tem uma arquitetura que não será resiliente quando alguns microsserviços falharem.

Além disso, as dependências de HTTP entre os microsserviços, como ao criar longos ciclos de solicitação/resposta com cadeias de solicitação HTTP, como foi mostrado na primeira parte da Figura 4-15, não permitem que os microsserviços sejam autônomos e também afetam o desempenho dos microsserviços quando um dos serviços nessa cadeia não é executado corretamente.

Quanto mais você adicionar dependências síncronas entre os microsserviços, como solicitações de consulta, pior será o tempo de resposta geral para os aplicativos clientes.

## Synchronous vs. async communication across microservices

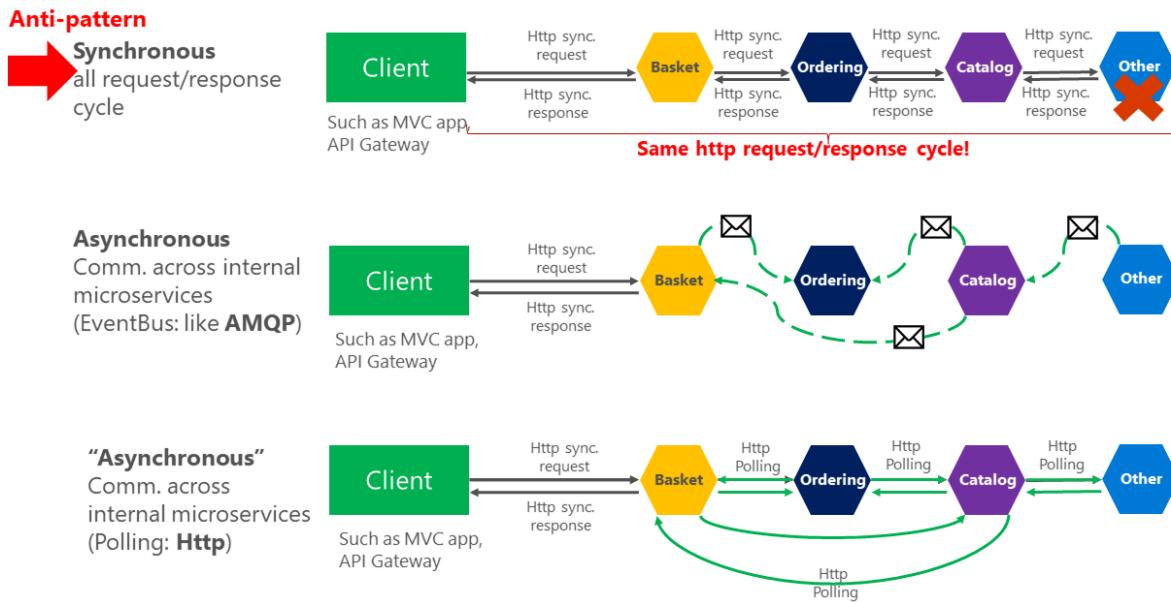


Figura 4-15. Antipadrões e padrões na comunicação entre microsserviços

Como mostrado no diagrama acima, na comunicação síncrona uma "cadeia" de solicitações é criada entre microserviços enquanto atende a solicitação do cliente. Isso é um antipadrão. Em microsserviços de comunicação assíncrona, use mensagens assíncronas ou a sondagem http para se comunicar com outros microsserviços, mas a solicitação do cliente é atendida imediatamente.

Se seu microsserviço precisar gerar uma ação adicional em outro microsserviço, se possível, não execute essa ação de forma síncrona e como parte da operação original de solicitação e resposta do microsserviço. Nesse caso, faça isso de forma assíncrona (usando o serviço de mensagens assíncrono ou eventos de integração, filas, etc.). No entanto, tanto quanto possível, não invoque a ação de forma síncrona como parte da operação de solicitação e resposta síncrona original.

E, finalmente, (e este é o momento em que maioria dos problemas surgem ao criar microsserviços), se o microsserviço inicial precisar de dados que sejam originalmente pertencentes a outros microsserviços, não confie em fazer solicitações síncronas para esses dados. Nesse caso, replique ou propague esses dados (somente os atributos necessários) para o banco de dados do serviço inicial usando consistência eventual (normalmente com eventos de integração, conforme será explicado nas próximas seções).

Como observado anteriormente nos limites do modelo de domínio de identificação para cada seção [de microsserviço](#), duplicar alguns dados em vários microsserviços não é um design incorreto — pelo contrário, ao fazer isso, você pode traduzir os dados para o idioma específico ou termos desse domínio adicional ou contexto limitado. Por exemplo, no [aplicativo eShopOnContainers](#) você `identity-api` tem um microsserviço chamado responsável pela maioria dos `User` dados do usuário com uma entidade chamada `.`  No entanto, quando você precisa armazenar `Ordering` dados sobre o usuário dentro `Buyer` do microsserviço, você armazená-lo como uma entidade diferente chamada `.`  A `Buyer` entidade compartilha a mesma `User` identidade com a entidade original, mas `ordering` pode ter apenas os poucos atributos necessários pelo domínio, e não todo o perfil de usuário.

Você pode usar qualquer protocolo para comunicar e propagar dados de forma assíncrona entre os microsserviços para garantir a consistência eventual. Conforme mencionado, você pode usar eventos de integração com um barramento de evento, com um agente de mensagem ou até mesmo com sondagem HTTP dos outros serviços. Não importa. A regra importante é não criar dependências síncronas entre os microsserviços.

As seções a seguir explicam os vários estilos de comunicação que você pode considerar para um aplicativo baseado em microsserviço.

# Estilos de comunicação

Existem vários protocolos e opções que você pode usar para comunicação, dependendo do tipo de comunicação que deseja usar. Se você estiver usando um mecanismo de comunicação baseado em solicitação/resposta síncrona, protocolos como HTTP e REST serão os mais comuns, principalmente se você estiver publicando serviços fora do host do Docker ou do cluster de microsserviços. Se você estiver fazendo a comunicação entre serviços internamente (no host do Docker ou no cluster de microsserviços), também será possível usar os mecanismos de comunicação de formato binário (como o WCF usando TCP e o formato binário). Como alternativa, você pode usar mecanismos de comunicação assíncrona baseada em mensagem, como o AMQP.

Também há vários formatos de mensagem como JSON ou XML, ou até mesmo formatos binários, que podem ser mais eficientes. Se o formato binário escolhido não for um padrão, provavelmente, não será uma boa ideia publicar os serviços publicamente usando esse formato. É possível usar um formato não padrão para a comunicação interna entre os microsserviços. Você pode fazer isso na comunicação entre os microsserviços dentro do host do Docker ou do cluster de microsserviços (por exemplo, orquestradores do Docker) ou para aplicativos cliente do proprietário que se comunicam com os microsserviços.

## Comunicação de solicitação/resposta com HTTP e REST

Quando um cliente usa a comunicação de solicitação/resposta, ele envia uma solicitação para um serviço e, em seguida, o serviço processa a solicitação e retorna uma resposta. A comunicação de solicitação/resposta é muito adequada principalmente para consultar dados de uma interface do usuário em tempo real (uma interface do usuário em tempo real) dos aplicativos clientes. Portanto, em uma arquitetura de microsserviço provavelmente você usará esse mecanismo de comunicação para a maioria das consultas, conforme mostrado na Figura 4-16.

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

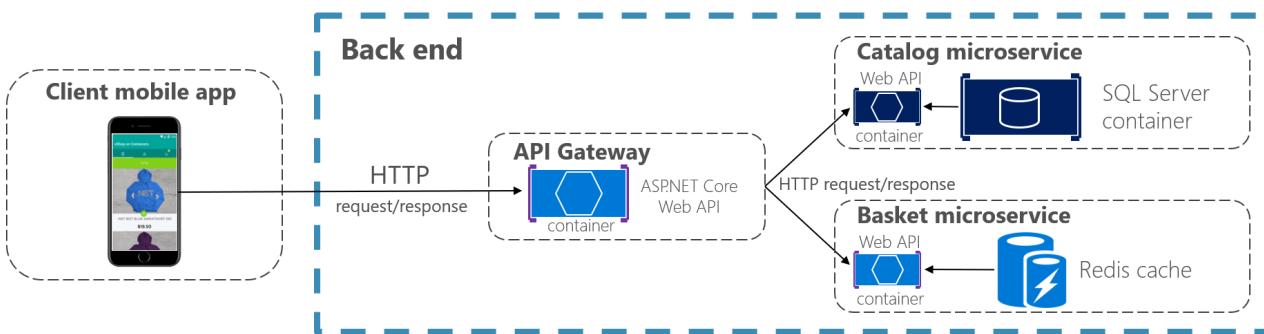


Figura 4-16. Usando a comunicação de solicitação/resposta HTTP (síncrona ou assíncrona)

Quando um cliente usa a comunicação de solicitação/resposta, ele considera que a resposta chegará muito em breve, geralmente, em menos de um segundo ou, no máximo, em alguns segundos. Para respostas em atraso, você precisa implementar a comunicação assíncrona baseada em [padrões de sistema de mensagens](#) e em [tecnologias de sistema de mensagens](#), que é uma abordagem diferente que explicaremos na próxima seção.

Um estilo popular de arquitetura para comunicação de solicitação/resposta é o [REST](#) (Transferência de Estado Representacional). Essa abordagem é baseada e fortemente vinculada ao protocolo [HTTP](#), adotando verbos HTTP como GET, POST e PUT. O REST é a abordagem de comunicação de arquitetura mais comum usada na criação de serviços. Você pode implementar serviços REST ao desenvolver serviços de API Web ASP.NET Core.

Há um valor adicional ao usar serviços REST HTTP como sua linguagem IDL. Por exemplo, ao usar [metadados do Swagger](#) para descrever sua API de serviço, você poderá usar ferramentas que geram stubs de cliente para descobrir e consumir seus serviços diretamente.

## Recursos adicionais

- O Martin Fowler. Richardson Maturity Model Uma descrição do modelo REST.

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

- **Swagger** O site oficial.

<https://swagger.io/>

#### Comunicação por push e em tempo real baseada em HTTP

Outra possibilidade (geralmente para finalidades diferentes do REST) é uma comunicação em tempo real e de um-para-muitos com estruturas de nível mais alto, como [ASP.NET SignalR](#) e protocolos como [WebSockets](#).

Como mostra a Figura 4-17, a comunicação HTTP em tempo real significa que o código do servidor pode enviar conteúdo por push para os clientes conectados à medida que os dados ficam disponíveis, ou seja, o servidor não precisa esperar que um cliente solicite novos dados.

## Push and real-time communication based on HTTP

One-to-many communication

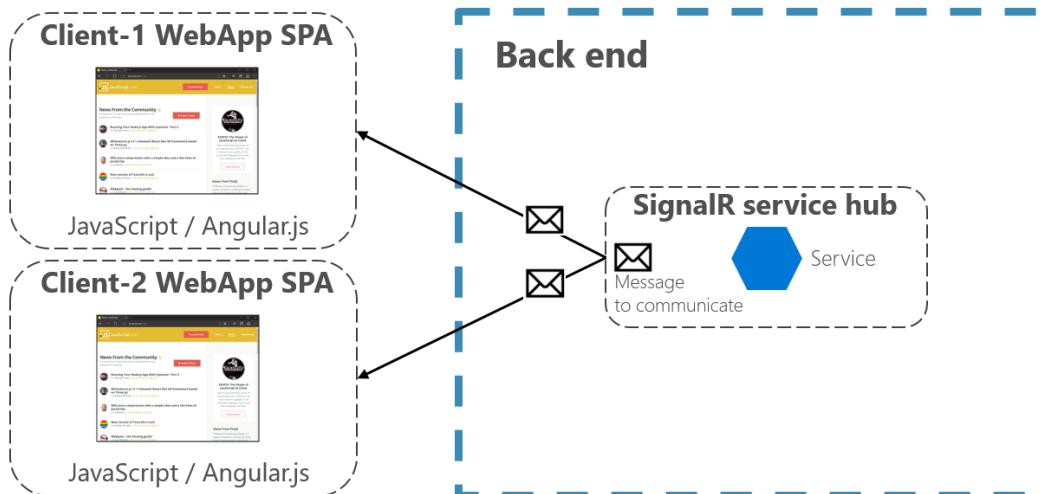


Figura 4-17. Comunicação de mensagem assíncrona de um-para-um em tempo real

O SignalR é uma boa maneira de atingir a comunicação em tempo real para enviar por push o conteúdo para os clientes de um servidor de back-end. Como a comunicação ocorre em tempo real, os aplicativos clientes mostram as alterações quase instantaneamente. Geralmente, isso é tratado por um protocolo como WebSockets, usando várias conexões de WebSocket (uma por cliente). Um exemplo típico é quando um serviço comunica uma alteração na pontuação de um jogo de esportes para vários aplicativos Web clientes simultaneamente.

[PRÓXIMO](#)

[ANTERIOR](#)

# Comunicação assíncrona baseada em mensagens

10/09/2020 • 15 minutes to read • [Edit Online](#)

Mensagens assíncronas e comunicação controlada por evento são críticos ao propagar alterações entre vários microsserviços e seus modelos de domínio relacionados. Conforme mencionado anteriormente na discussão, microsserviços e BCs (Contextos Limitados), modelos (Usuário, Cliente, Produto, Conta etc.) podem ter diferentes significados para diferentes microsserviços ou BCs. Isso significa que, quando ocorrem alterações, você precisa de algum modo de reconciliar essas alterações entre diferentes modelos. Uma solução é consistência eventual e comunicação controlada por evento com base em mensagens assíncronas.

Ao usar mensagens, processos se comunicam trocando mensagens de maneira assíncrona. Um cliente faz um comando ou uma solicitação a um serviço ao enviar uma mensagem a ele. Se o serviço precisar responder, ele enviará uma mensagem diferente de volta ao cliente. Como é uma comunicação baseada em mensagens, o cliente presume que a resposta não será recebida imediatamente e que poderá não haver resposta alguma.

Uma mensagem é composta por um cabeçalho (metadados como informações de identificação ou de segurança) e um corpo. As mensagens geralmente são enviadas por meio de protocolos assíncronos, como AMQP.

A infraestrutura preferencial para esse tipo de comunicação na comunidade de microsserviços é um agente de mensagem leve, que é diferente de agentes e orquestradores grandes usados em SOA. Em um agente de mensagem leve, a infraestrutura é normalmente "inútil," atuando somente como um agente de mensagens, com implantações simples como RabbitMQ ou um barramento de serviço escalonável na nuvem como o Barramento de Serviço do Azure. Nesse cenário, a maioria do pensamento "inteligente" ainda está nos pontos de extremidade que estão produzindo e consumindo mensagens; ou seja, nos microsserviços.

Outra regra que você deve tentar seguir o máximo possível é usar apenas mensagens assíncronas entre os serviços internos e usar comunicação síncrona (como HTTP) apenas dos aplicativos cliente para os serviços de front-end (gateways de API mais o primeiro nível de microsserviços).

Há dois tipos de comunicação assíncrona de mensagens: comunicação baseada em mensagens do receptor único e comunicação baseada em mensagens de vários receptores. As seções a seguir fornecem detalhes sobre eles.

## Comunicação baseada em mensagem de destinatário único

Comunicação assíncrona baseada em mensagem com um único destinatário significa que há comunicação ponto a ponto que entrega uma mensagem a exatamente um dos consumidores que está lendo do canal e que essa mensagem é processada apenas uma vez. No entanto, existem situações especiais. Por exemplo, em um sistema de nuvem que tenta se recuperar automaticamente de falhas, a mesma mensagem pode ser enviada várias vezes. Devido à rede ou outras falhas, o cliente deve ser capaz de tentar novamente enviar as mensagens, e o servidor precisa implementar uma operação para ser idempotente para processar uma mensagem específica apenas uma vez.

A comunicação baseada em mensagem destinatário único é especialmente adequada para enviar comandos assíncronos de um microsserviço para outro, conforme mostrado na Figura 4-18 que ilustra essa abordagem.

Depois de começar a enviar comunicação baseada em mensagens (seja com comandos ou eventos), você deve evitar misturar comunicação baseada em mensagem com comunicação HTTP síncrona.

## Single receiver message-based communication

(i.e. Message-based Commands)

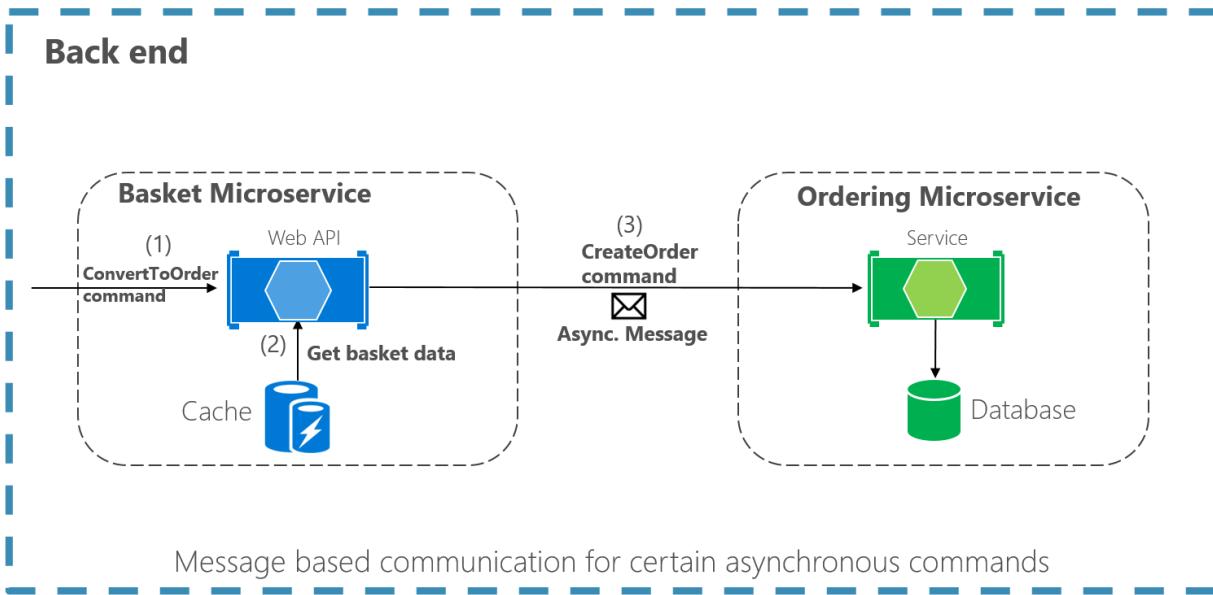


Figura 4-18. Um único microsserviço recebendo uma mensagem assíncrona

Quando os comandos são provenientes de aplicativos cliente, eles podem ser implementados como comandos síncronos HTTP. Use comandos baseados em mensagem quando precisar de maior escalabilidade ou quando já estiver em um processo de negócios baseado em mensagem.

## Comunicação baseada em mensagens de vários destinatários

Como uma abordagem mais flexível, você também poderá usar um mecanismo de publicação/assinatura para que a sua comunicação do remetente esteja disponível a microsserviços de assinante adicionais ou para aplicativos externos. Portanto, ele o ajuda a seguir o [princípio de abrir/fechar](#) no serviço de envio. Dessa forma, mais assinantes podem ser adicionados no futuro sem necessidade de modificar o serviço do remetente.

Ao usar uma comunicação de publicação/assinatura, talvez você esteja usando uma interface de barramento de eventos para publicar eventos a qualquer assinante.

## Comunicação controlada por evento assíncrono

Ao usar comunicação controlada por evento assíncrono, um microsserviço publica um evento de integração quando acontece algo em seu domínio e outro microsserviço precisa estar ciente disso, como uma alteração de preço em um microsserviço do catálogo de produtos. Microsserviços adicionais assinam os eventos para que possam recebê-los de maneira assíncrona. Quando isso acontece, os receptores podem atualizar as próprias entidades de domínio, o que podem causar a publicação de mais eventos de integração. Este sistema de publicação/assinatura normalmente é executado por meio de uma implementação de um barramento de evento. O barramento de evento pode ser projetado como uma abstração ou interface, com a API que é necessária para assinar ou cancelar a assinatura de eventos e para publicar eventos. O barramento de evento também pode ter uma ou mais implementações com base em qualquer agente de mensagens e entre processos, como uma fila de mensagens ou barramento de serviço que seja compatível com a comunicação assíncrona e um modelo de publicação/assinatura.

Se um sistema usa consistência eventual orientada por eventos de integração, é recomendável que essa abordagem fique completamente clara para o usuário final. O sistema não deve usar uma abordagem que simule eventos de integração, como sistemas de sondagem ou SignalR do cliente. O usuário final e o proprietário da empresa precisam adotar explicitamente consistência eventual no sistema e observar que, em muitos casos, os negócios não têm nenhum problema com essa abordagem, desde que ela seja explícita. Isso é importante porque

os usuários podem esperar ver alguns resultados imediatamente e isso pode não acontecer com a consistência eventual.

Conforme observado anteriormente na seção [Desafios e soluções para o gerenciamento de dados distribuídos](#), você pode usar eventos de integração para implementar as tarefas de negócios que abrangem vários microsserviços. Assim, você terá consistência eventual entre esses serviços. Uma transação eventualmente consistente é composta por uma coleção de ações distribuídas. Em cada ação, o microsserviço relacionado atualiza uma entidade de domínio e publica outro evento de integração que gera a próxima ação dentro da mesma tarefa comercial de ponta a ponta.

Um ponto importante é que você pode querer comunicar-se com vários microsserviços inscritos para o mesmo evento. Para fazer isso, você pode usar mensagens de publicação/assinatura com base em comunicação controlada por evento, conforme mostra a Figura 4-19. Esse mecanismo de publicação/assinatura não é exclusivo da arquitetura de microsserviço. Ele é semelhante à maneira como [Contextos Limitados](#) no DDD devem se comunicar ou à maneira como você propaga atualizações do banco de dados de gravação para o banco de dados de leitura no padrão de arquitetura de [CQRS \(Segregação de Responsabilidade de Comando e Consulta\)](#). A meta é ter consistência eventual entre várias fontes de dados em seu sistema distribuído.

## Asynchronous event-driven communication

### Multiple receivers

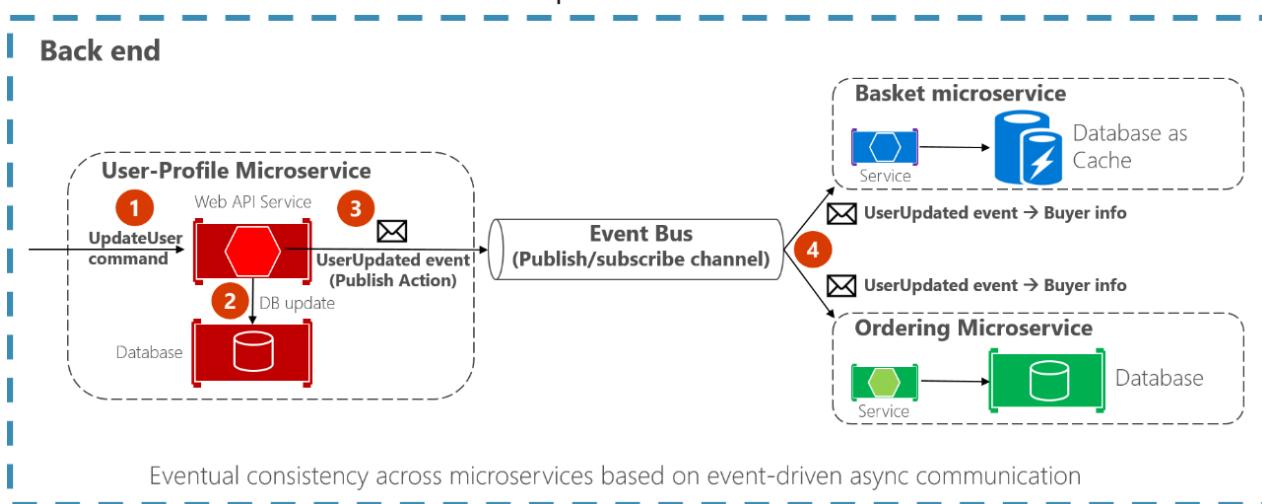


Figura 4-19. Comunicação de mensagem controlada por evento assíncrono

Na comunicação assíncrona orientada por evento, um microserviço publica eventos em um barramento de evento e muitos microserviços podem assiná-lo, para ser notificado e agir sobre ele. Sua implementação determinará o protocolo a ser usado para comunicações controladas por eventos e baseadas em mensagem. O [AMQP](#) pode ajudar a obter uma comunicação confiável na fila.

Ao usar um barramento de evento, talvez você queira usar um nível de abstração (como uma interface de barramento de evento) com base em uma implementação relacionada em classes com o código usando a API de um agente de mensagens como [RabbitMQ](#) ou um barramento de serviço como [Barramento de Serviço do Azure com Tópicos](#). Como alternativa, você talvez queira usar um barramento de serviço de nível superior, como NServiceBus, MassTransit ou Brighter para articular seu barramento de evento e sistema de publicação/assinatura.

## Uma observação sobre tecnologias para mensagens para sistemas de produção

As tecnologias de mensagens disponíveis para implementar seu barramento do evento abstrato estão em diferentes níveis. Por exemplo, produtos, como RabbitMQ (um transporte do agente de mensagens) e o

Barramento de Serviço do Azure ficam em um nível inferior a outros produtos, como NServiceBus, MassTransit ou Brighter, que podem funcionar sobre RabbitMQ e o Barramento de Serviço do Azure. Sua escolha depende de quantos recursos avançados no nível do aplicativo e escalabilidade imediata você precisa para seu aplicativo. Para implementar apenas um barramento de evento de prova de conceito para seu ambiente de desenvolvimento, como foi feito no exemplo de eShopOnContainers, pode ser suficiente uma implementação simples sobre RabbitMQ em execução em um contêiner do Docker.

No entanto, para sistemas críticos e de produção que precisam de hiperescalabilidade, talvez você queira avaliar o Barramento de Serviço do Azure. Para abstrações de alto nível e recursos que tornam o desenvolvimento de aplicativos distribuídos mais fácil, recomendamos que você avalie outros barramentos de serviço de software livre e comerciais, como NServiceBus, MassTransit e Brighter. É claro que você pode criar seus próprios recursos de barramento de serviço sobre tecnologias de nível inferior, como RabbitMQ e Docker. Mas esse trabalho pode custar muito caro para um aplicativo empresarial personalizado.

## Publicação resiliente para o barramento de evento

Um desafio ao implementar uma arquitetura orientada a eventos em vários microsserviços é como atualizar atomicamente o estado no microsserviço original enquanto publica de maneira resiliente o evento de integração relacionado no barramento de evento, de alguma maneira baseado em transações. A seguir estão algumas maneiras de fazer isso, embora haja outras abordagens também.

- Usando uma fila transacional (baseada em DTC) como MSMQ. (No entanto, essa é uma abordagem herdada.)
- Usando [mineração do log de transações](#).
- Usando o padrão [Event Sourcing](#) completo.
- Usando o [Padrão de caixa de saída](#): uma tabela de banco de dados transacional como uma fila de mensagens que será a base de um componente do criador do evento que criará e publicará o evento.

Tópicos adicionais a serem considerados ao usar comunicação assíncrona são idempotência de mensagem e eliminação de duplicação de mensagem. Esses tópicos são abordados na seção [Implementar a comunicação baseada em eventos entre microsserviços \(eventos de integração\)](#) mais adiante neste guia.

## Recursos adicionais

- **Mensagens controladas por evento**  
[https://soapatterns.org/design\\_patterns/event\\_driven\\_messaging](https://soapatterns.org/design_patterns/event_driven_messaging)
- **Canal de publicação/assinatura**  
<https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- **Udi Dahan. CQRS esclarecido**  
<https://udidahan.com/2009/12/09/clarified-cqrs/>
- **Separação das Operações de Comando e de Consulta (CQRS)**  
[/azure/architecture/patterns/cqrs](https://azure/architecture/patterns/cqrs)
- **Comunicação entre contextos limitados**  
[/previous-versions/msp-n-p/jj591572\(v=pandp.10\)](https://previous-versions/msp-n-p/jj591572(v=pandp.10))
- **Consistência eventual**  
[https://en.wikipedia.org/wiki/Eventual\\_consistency](https://en.wikipedia.org/wiki/Eventual_consistency)
- **Jimmy Bogard. Refatoração em relação à resiliência: avaliando o acoplamento**  
<https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/>

[ANTERIOR](#)

[AVANÇAR](#)

# Criando, evoluindo e fazendo o controle de versão de APIs e de contratos de microsserviços

10/09/2020 • 4 minutes to read • [Edit Online](#)

Uma API de microsserviço é um contrato entre o serviço e seus clientes. Você poderá evoluir um microsserviço de forma independente apenas se você não precisar interromper seu contrato de API, que é o motivo pelo qual o contrato é tão importante. Se você alterar o contrato, isso afetará seus aplicativos cliente ou seu Gateway de API.

A natureza da definição de API depende do protocolo que você está usando. Por exemplo, se você estiver usando mensagens (como [AMQP](#)), a API consistirá nos tipos de mensagem. Se você estiver usando serviços HTTP e RESTful, a API consistirá nas URLs e nos formatos JSON de solicitação e de resposta.

No entanto, mesmo se você estiver em dúvida sobre seu contrato inicial, uma API de serviço precisará ser alterada com o tempo. Quando isso acontecer – e, principalmente, se a API for uma API pública consumida por vários aplicativos cliente – normalmente não será possível forçar todos os clientes a atualizarem para seu novo contrato de API. Geralmente, é necessário implantar novas versões de um serviço de forma incremental de maneira que versões novas e antigas de um contrato de serviço estejam em execução simultaneamente. Portanto, é importante ter uma estratégia para o controle de versão do serviço.

Quando as alterações na API forem pequenas, como adicionar atributos ou parâmetros à sua API, os clientes que usam uma API mais antiga devem mudar e trabalhar com a nova versão do serviço. Talvez seja possível fornecer valores padrão para quaisquer atributos ausentes que sejam necessários, e os clientes talvez podem ignorar quaisquer atributos de resposta extra.

No entanto, às vezes, é necessário fazer alterações importantes e incompatíveis em uma API de serviço. Como talvez não seja possível forçar serviços ou aplicativos cliente a serem atualizados imediatamente para a nova versão, um serviço deve dar suporte a versões mais antigas da API por algum período. Se você estiver usando um mecanismo baseado em HTTP como REST, uma abordagem deverá inserir o número de versão da API na URL ou no cabeçalho HTTP. Em seguida, é possível decidir entre implementar ambas as versões do serviço simultaneamente dentro da mesma instância de serviço ou implantar instâncias diferentes que lidam com uma versão da API. Uma boa abordagem para isso é o [padrão mediador](#) (por exemplo, [biblioteca MediatR](#)) para desacoplar as diferentes versões de implementação em manipuladores independentes.

Por fim, se você estiver usando uma arquitetura REST, o [Hypermedia](#) será a melhor solução para controlar a versão de seus serviços e permitir APIs capazes de evoluir.

## Recursos adicionais

- Scott Hanselman. **ASP.NET Core facilitar o controle de versão da API Web RESTful**  
<https://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>
- **Controle de versão de uma API Web RESTful**  
[/azure/architecture/best-practices/api-design#versioning-a-restful-web-api](https://azure.microsoft.com/en-us/documentation/articles/architecture-best-practices-api-design/#versioning-a-restful-web-api)
- Roy de campo. **Controle de versão, hipermídia e REST**  
<https://www.infoq.com/articles/roy-fielding-on-versioning>

# Capacidade de endereçamento de microsserviços e o Registro do serviço

18/03/2020 • 3 minutes to read • [Edit Online](#)

Cada microsserviço tem um nome exclusivo (URL) usado para resolver sua localização. O microsserviço precisa ser endereçável independentemente do local em que está sendo executado. Se você tiver que pensar em qual computador está executando um microsserviço específico, as coisas poderão se complicar rapidamente. Da mesma forma que o DNS resolve uma URL para um determinado computador, seu microsserviço precisa ter um nome exclusivo para que seu local atual seja detectável. Os microsserviços precisam de nomes endereçáveis que os tornem independentes da infraestrutura na qual eles estão sendo executados. Isso implica que há uma interação entre a maneira como seu serviço é implantado e como ele é detectado, porque deve haver um [Registro do serviço](#). Do mesmo modo, quando um computador falha, o serviço de Registro deve ser capaz de indicar o local em que o serviço está sendo executado no momento.

O [padrão de Registro de serviço](#) é uma parte importante da descoberta de serviço. O Registro é um banco de dados que contém os locais de rede das instâncias de serviço. Um Registro de serviço precisa ser atualizado e estar altamente disponível. Os clientes podem armazenar locais de rede em cache obtidos do Registro de serviço. No entanto, essas informações eventualmente ficam desatualizadas e os clientes não podem mais descobrir instâncias de serviço. Consequentemente, um Registro de serviço consiste em um cluster de servidores que usam um protocolo de replicação para manter a consistência.

Em alguns ambientes de implantação de microsserviço (chamados clusters, a serem abordados em uma seção posterior), a descoberta de serviço é interna. Por exemplo, um ambiente do AKS (Serviço de Contêiner do Azure com Kubernetes) pode manipular o registro e o cancelamento de registro da instância de serviço. Ele também executa um proxy em cada host de cluster que desempenha a função de roteador de descoberta do servidor.

## Recursos adicionais

- **Chris Richardson. Padrão: Registro de serviços**  
<https://microservices.io/patterns/service-registry.html>
- **Auth0. O Registro de Serviços**  
<https://auth0.com/blog/an-introduction-to-microservices-part-3-the-service-registry/>
- **Gabriel Schenker. Descoberta de serviços**  
<https://lostechies.com/gabrielschenker/2016/01/27/service-discovery/>

[PRÓXIMO](#)

[ANTERIOR](#)

# Criando a interface do usuário composta baseada em microservices

18/03/2020 • 5 minutes to read • [Edit Online](#)

A arquitetura de microservices geralmente começa com os dados e a lógica de manuseio do lado do servidor, mas, em muitos casos, a ui ainda é tratada como um monólito. No entanto, uma abordagem mais avançada, chamada [micro frontends](#), é projetar sua interface do crédito com base em microservices também. Isso significa ter uma interface do usuário de composição produzida pelos microservices, em vez de ter microservices no servidor e apenas um aplicativo cliente monolítico consumindo os microservices. Com essa abordagem, os microservices que você criar poderão estar completos com lógica e representação visual.

A Figura 4-20 mostra a abordagem mais simples de apenas consumir microservices de um aplicativo cliente monolítico. Obviamente, é possível ter um serviço MVC ASP.NET no meio termo produzindo o HTML e o JavaScript. A figura é uma simplificação que destaca que você tem uma única interface do usuário do cliente (monolítica) consumindo microservices, que se concentram apenas em lógica e nos dados, e não na forma da interface do usuário (HTML e JavaScript).

## Monolithic UI consuming microservices

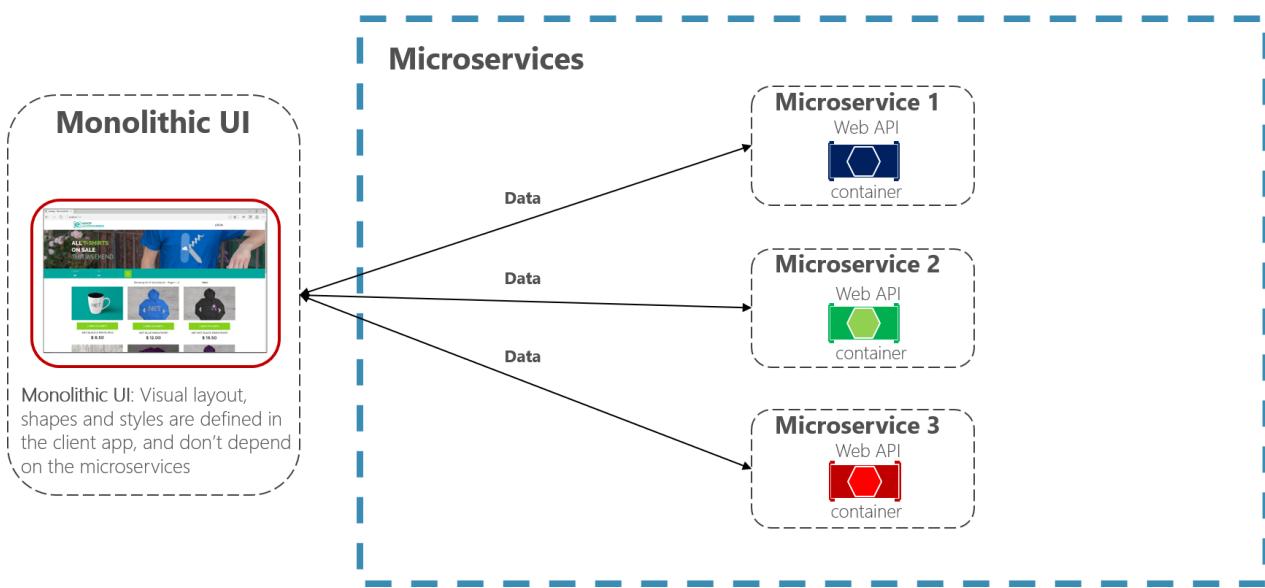


Figura 4-20. Um aplicativo de interface do usuário monolítico consumindo microservices de back-end

Em contraste, uma interface do usuário de composição é gerada precisamente e composta pelos próprios microservices. Alguns dos microservices promovem a forma visual de áreas específicas da interface do usuário. A principal diferença é que você tem componentes de interface do usuário do cliente (classes TypeScript, por exemplo) com base em modelos, sendo que o ViewModel da interface do usuário de modelagem de dados para esses modelos é obtido de cada microservice.

Em tempo de inicialização do aplicativo cliente, cada um dos componentes de interface do usuário do cliente (classes TypeScript, por exemplo) se registra com um microservice de infraestrutura capaz de fornecer ViewModels para um determinado cenário. Se o microservice alterar a forma, a interface do usuário também será alterada.

A Figura 4-21 mostra uma versão dessa abordagem de interface do usuário de composição. Isso é simplificado porque você pode ter outros microservices que estão agregando partes granulares com base em diferentes

técnicas. Isso depende se você está criando uma abordagem tradicional da Web (ASP.NET MVC) ou um SPA (Aplicativo de Página Única).

## Composite UI generated by microservices

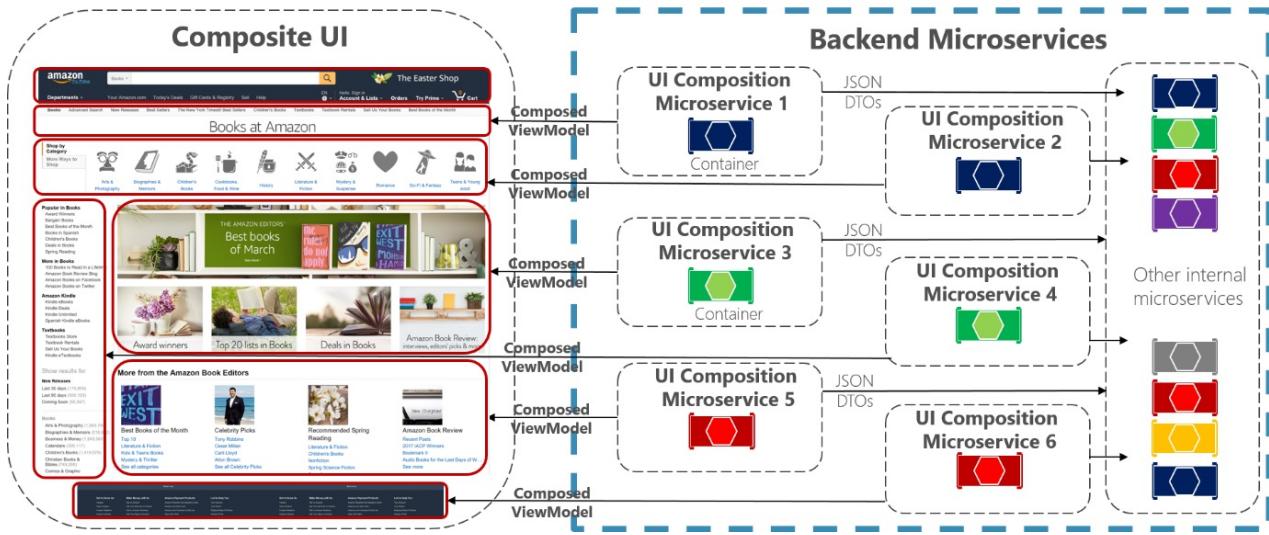


Figura 4-21. Exemplo de um aplicativo de interface do usuário de composição formatado por microsserviços de back-end

Cada um desses microsserviços de composição de interface do usuário seria semelhante a um Gateway de API pequeno. Mas, neste caso, cada um é responsável por uma pequena área de IU.

Uma abordagem de interface do usuário composta que é controlada por microsserviços pode ser mais desafiadora ou menos, dependendo de quais tecnologias de interface do usuário estão sendo usadas. Por exemplo, você não usará as mesmas técnicas para criar um aplicativo Web tradicional que você usa para criar um SPA ou para o aplicativo móvel nativo (como acontece durante o desenvolvimento de aplicativos Xamarin, que podem ser mais desafiadores para essa abordagem).

O aplicativo de exemplo [eShopOnContainers](#) usa a abordagem de interface do usuário monolítica por vários motivos. Primeiro, é uma introdução a microsserviços e contêineres. Uma interface do usuário de composição é mais avançada, mas também requer mais complexidade ao criar e desenvolver a interface do usuário. Em segundo lugar, o eShopOnContainers também oferece um aplicativo móvel nativo baseado no Xamarin, que tornaria isso mais complexo no lado C# do cliente.

No entanto, recomendamos que você use as seguintes referências para saber mais sobre a interface do usuário de composição com base em microsserviços.

## Recursos adicionais

- **Micro Frontends (blog de Martin Fowler)**  
<https://martinfowler.com/articles/micro-frontends.html>
- **Micro Frontends (site de Michael Geers)**  
<https://micro-frontends.org/>
- **Interface do usuário composta usando o ASP.NET (Particular's Workshop)**  
<https://github.com/Particular/Workshop/tree/master/demos/asp-net-core>
- **Ruben Oostinga. O Frontend Monolítico na Arquitetura de Microsserviços**  
<https://xebia.com/blog/the-monolithic-frontend-in-the-microservices-architecture/>
- **Mauro Servienti. O segredo da melhor composição da UI**

<https://particular.net/blog/secret-of-better-ui-composition>

- **Viktor Farcic. Incluindo componentes web front-end em microserviços**  
<https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/>
- **Gerenciando o front-end na arquitetura de microsserviços**  
<https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html>

[PRÓXIMO](#)

[ANTERIOR](#)

# Resiliência e a alta disponibilidade em microsserviços

09/04/2020 • 12 minutes to read • [Edit Online](#)

Lidar com falhas inesperadas é um dos problemas mais difíceis de se resolver, especialmente em um sistema distribuído. Grande parte do código que os desenvolvedores gravam envolve tratamento de exceções, e também é nisso que a maior parte do tempo é gasta no teste. O problema é mais complicado do que escrever código para tratar de falhas. O que acontece quando o computador onde o microsserviço está em execução falha? Você não precisará apenas detectar essa falha do microsserviço (um problema difícil por si só), mas também precisará de algo para reiniciar o microsserviço.

Um microsserviço precisa ser resiliente a falhas e conseguir reiniciar geralmente em outro computador para disponibilidade. Essa resiliência também se resume ao estado que foi salvo em nome de microsserviço, do qual os microsserviços podem recuperar esse estado, e se o microsserviço pode reiniciar com êxito. Em outras palavras, deve haver resiliência na capacidade de computação (o processo pode reiniciar a qualquer momento), bem como resiliência no estado ou nos dados (sem perda de dados e os dados permanecem consistentes).

Os problemas de resiliência são abordados durante outros cenários, como quando ocorrem falhas durante uma atualização de aplicativo. O microsserviço, trabalhando com o sistema de implantação, precisa determinar se pode continuar a avançar para a versão mais recente ou, em vez disso, reverter para uma versão anterior para manter um estado consistente. É preciso considerar perguntas como se há computadores suficientes disponíveis para continuar avançando e como recuperar versões anteriores do microsserviço. Isso requer que o microsserviço emita informações de integridade para que o aplicativo geral e o orquestrador possam tomar essas decisões.

Além disso, a resiliência está relacionada a como sistemas baseados em nuvem devem se comportar. Conforme mencionado, um sistema baseado em nuvem deve compreender falhas e tentar se recuperar automaticamente delas. Por exemplo, no caso de falhas de rede ou um contêiner, aplicativos cliente ou serviços cliente devem ter uma estratégia de repetição de envio de mensagens ou novas tentativas de solicitações, já que, em muitos casos, as falhas na nuvem são parciais. A seção [Implementando aplicativos resilientes](#) neste guia aborda como lidar com falhas parciais. Descreve técnicas como novas tentativas com retirada exponencial ou o padrão de Disjuntor no .NET Core usando bibliotecas como [Polly](#), que oferece uma grande variedade de políticas para lidar com esse assunto.

## Gerenciamento de integridade e diagnóstico em microsserviços

Pode parecer óbvio, e isso muitas vezes é negligenciado, mas um microsserviço deve informar sua integridade e seu diagnóstico. Caso contrário, há pouca percepção de uma perspectiva de operações. Correlacionar os eventos de diagnóstico em um conjunto de serviços independentes e lidar com a defasagem do relógio do computador para entender a ordem dos eventos é um desafio. Da mesma maneira que você interage com um microsserviço sobre protocolos e formatos de dados estabelecidos, existe a necessidade de padronização de como registrar em log a integridade e eventos de diagnóstico que, por fim, terminam em um repositório de eventos para consulta e exibição. Em uma abordagem de microserviços, ele tem chave que diferentes equipes concordem com um único formato de log. Deve haver uma abordagem consistente para exibir eventos de diagnóstico no aplicativo.

### Verificações de integridade

Integridade é diferente de diagnóstico. Integridade significa que o microsserviço reporta seu estado atual para tomar as devidas ações. Um bom exemplo é trabalhar com mecanismos de atualização e implantação para manter a disponibilidade. Embora um serviço possa não estar íntegro no momento devido a uma falha de processo ou reinicialização do computador, o serviço ainda pode estar operacional. A última coisa de que você precisa é piorar a situação executando uma atualização. A melhor abordagem é fazer uma investigação primeiro ou aguardar a recuperação do microsserviço. Os eventos de integridade de um microsserviço permitem tomar decisões bem

informadas e ajudam realmente a criar serviços que recuperam a si próprios.

Na seção [Implementação de verificações de integridade nos serviços ASP.NET Core](#) deste guia, explicamos como usar uma nova biblioteca ASP.NET HealthChecks em seus microsserviços para que eles possam relatar o próprio estado a um serviço de monitoramento para executar as ações apropriadas.

Você também tem a opção de usar uma excelente biblioteca de código-fonte aberto chamada Pulse vencer, disponível no [GitHub](#) e como um [pacote do NuGet](#). Essa biblioteca também faz verificações de integridade, mas com uma diferença, pois ela lida com dois tipos de verificação:

- **Viva** : Verifique se o microsserviço está vivo, ou seja, se ele é capaz de aceitar pedidos e responder.
- **Prontidão**: Verifique se as dependências do microsserviço (Banco de Dados, serviços de fila, etc.) estão prontas, para que o microsserviço possa fazer o que é suposto fazer.

### Usando fluxos de eventos de logs e diagnóstico

Logs fornecem informações sobre como um aplicativo ou serviço está sendo executado, incluindo exceções, avisos e mensagens informativas simples. Normalmente, cada log está em um formato de texto com uma linha por evento, embora exceções também muitas vezes mostrem o rastreamento de pilha em várias linhas.

Em aplicativos baseados em servidor monolíticos, você pode simplesmente gravar logs de um arquivo no disco (um arquivo de log) e então analisá-lo com qualquer ferramenta. Uma vez que a execução do aplicativo está limitada a um servidor ou VM fixo, geralmente não é complexo demais analisar o fluxo de eventos. No entanto, em um aplicativo distribuído em que vários serviços são executados em vários nós em um cluster do orquestrador, poder correlacionar eventos distribuídos é um desafio.

Um aplicativo baseado em microsserviço não deve tentar armazenar o fluxo de saída de eventos nem arquivos de log por si só, nem tentar gerenciar o roteamento de eventos para um local central. Ele deve ser transparente, o que significa que cada processo deve gravar apenas seu fluxo de eventos em uma saída padrão que, por baixo, será coletado pela infraestrutura do ambiente de execução em que está sendo executado. Um exemplo desses roteadores do fluxo de evento é [Microsoft.Diagnostic.EventFlow](#), que coleta os fluxos de eventos de várias fontes e publica-os em sistemas de saída. Eles podem incluir uma saída simples padrão para um ambiente de desenvolvimento ou sistemas de nuvem como o [Azure Monitor](#) e o [Diagnóstico do Azure](#). Também há boas plataformas e ferramentas de análise de log de terceiros que podem pesquisar, alertar, relatar e monitorar logs, inclusive em tempo real, como [Splunk](#).

### Orquestradores gerenciando informações de integridade e diagnóstico

Quando você cria um aplicativo baseado em microsserviço, precisa lidar com a complexidade. Logicamente, é simples lidar com um único microsserviço, mas dezenas ou centenas de tipos e milhares de instâncias de microsserviços são um problema complexo. Não envolve apenas criar sua arquitetura de microsserviço: você também precisará de alta disponibilidade, capacidade de endereçamento, resiliência, integridade e diagnóstico se você quiser ter um sistema estável e coeso.



## Microservice Platform (Orchestrators/Clusters)

**Figura 4-22.** Uma Plataforma de Microsserviço é fundamental para o gerenciamento de integridade do aplicativo

É muito difícil você mesmo resolver os problemas complexos mostrados na Figura 4-22. As equipes de desenvolvimento devem se concentrar na solução de problemas de negócios e na criação de aplicativos personalizados com abordagens baseadas em microsserviço. Elas não devem se concentrar na solução de problemas de infraestrutura complexa; se fizessem isso, o custo de qualquer aplicativo baseado em microsserviço seria enorme. Portanto, há plataformas orientadas a microsserviços, conhecidas como orquestradores ou clusters de microsserviço, que tentam resolver os problemas de disco rígido de criar e executar um serviço e usar os recursos de infraestrutura com eficiência. Isso reduz as complexidades da criação de aplicativos que usam uma abordagem de microsserviços.

Orquestradores diferentes podem parecer semelhantes, mas o diagnóstico e as verificações de integridade oferecidos por eles diferem em termos de recursos e estado de maturidade, às vezes dependendo da plataforma de sistema operacional, conforme explicado na próxima seção.

## Recursos adicionais

- O Aplicativo de Doze Fatores. XI. Logs: Trate logs como fluxos de eventos  
<https://12factor.net/logs>
- Repositório do GitHub da Biblioteca EventFlow de Diagnóstico da Microsoft.  
<https://github.com/Azure/diagnostics-eventflow>
- O que é a Azure Diagnostics  
</azure/azure-diagnostics>
- Conecte computadores Windows ao serviço Azure Monitor  
</azure/azure-monitor/platform/agent-windows>
- Registrando o que você quer dizer: usando o bloco de aplicativos de registro semântico  
[/previous-versions/msp-n-p/dn440729\(v=pandp.60\)](/previous-versions/msp-n-p/dn440729(v=pandp.60))
- Site oficial do Splunk.  
<https://www.splunk.com/>
- API de classe EventSource para ETW (Rastreamento de Eventos para Windows)  
<https://docs.microsoft.com/dotnet/api/system.diagnostics.tracing.eventsource>



# Orquestrar microsserviços e aplicativos de vários contêineres para alta escalabilidade e disponibilidade

09/04/2020 • 16 minutes to read • [Edit Online](#)

O uso de orquestradores em aplicativos prontos para produção é essencial se o aplicativo for baseado em microsserviços ou simplesmente dividido em vários contêineres. Conforme apresentado anteriormente, em uma abordagem baseada em microsserviço, cada microsserviço tem seu próprio modelo e dados para que seja autônomo de um ponto de vista de desenvolvimento e implantação. No entanto, se o aplicativo for mais tradicional, composto por diversos serviços (como o SOA), também haverá vários contêineres ou serviços que abrangem um único aplicativo de negócios e precisam ser implantados como um sistema distribuído. Esses tipos de sistemas são difíceis de escalar horizontalmente e gerenciar; portanto, um orquestrador é absolutamente necessário para ter um aplicativo pronto para produção, escalonável e com vários contêineres.

A Figura 4-23 ilustra a implantação de um aplicativo composto por vários microsserviços (contêineres) em um cluster.

## Composed Docker Applications in a Cluster

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

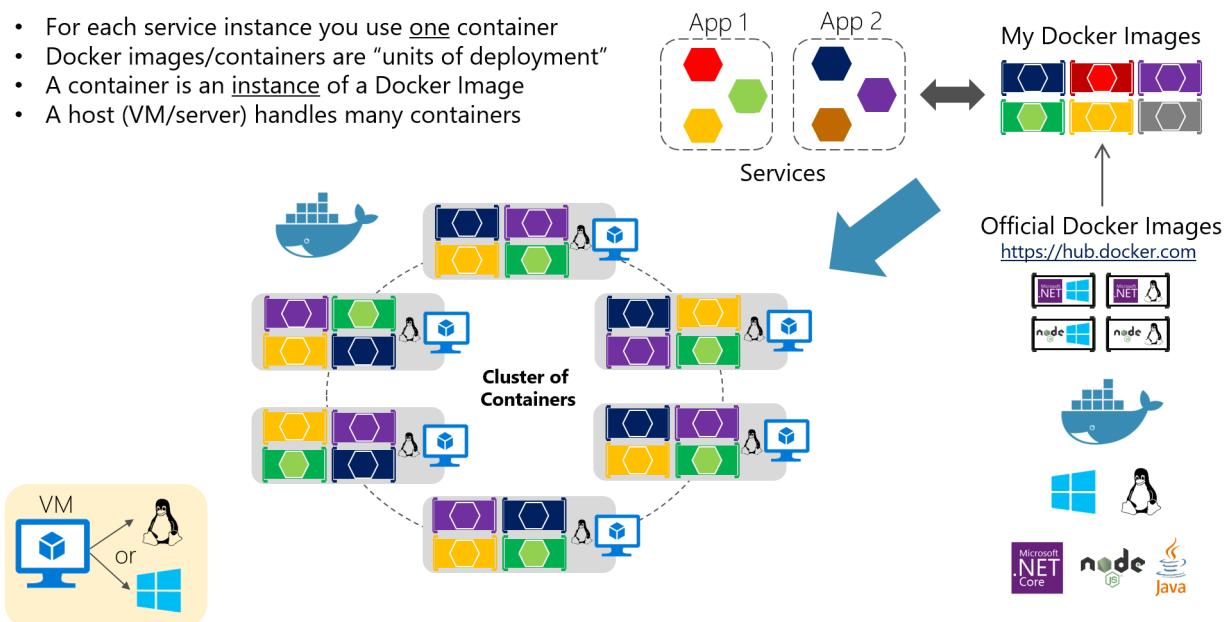


Figura 4-23. Um cluster de contêineres

você pode usar um contêiner para cada instância de serviço. Os contêineres Docker são "unidades de implantação" e um contêiner é uma instância de um Docker. Um hospedeiro lida com muitos contêineres. Parece ser uma abordagem lógica. Mas como você lida com o平衡amento de carga, roteamento e orquestração desses aplicativos compostos?

O Docker Engine simples em hosts individuais do Docker atende às necessidades de gerenciamento de instâncias de imagem única em um host, mas fica aquém quando se trata de gerenciar vários contêineres implantados em diversos hosts de aplicativos distribuídos mais complexos. Na maioria dos casos, é necessária uma plataforma de gerenciamento que inicie os contêineres automaticamente, escala horizontalmente os que têm várias instâncias por imagem, suspenda-os ou desligue-os quando preciso e também controlem como eles acessam recursos como a rede e o armazenamento de dados.

Para ir além do gerenciamento de contêineres individuais ou aplicativos compostos muito simples e em direção a

aplicativos empresariais com microsserviços, é necessário adotar a orquestração e as plataformas de clustering.

De uma perspectiva de arquitetura e desenvolvimento, ao criar aplicativos empresariais grandes, compostos e baseados em microsserviços, é importante entender as seguintes plataformas e produtos que dão suporte a cenários avançados:

**Clusters e orquestradores.** Quando é preciso expandir os aplicativos em vários hosts do Docker, como um aplicativo grande baseado em microsserviço, é essencial ter a capacidade de gerenciar todos esses hosts como um cluster único, abstraindo a complexidade da plataforma subjacente. É isso que os clusters e orquestradores de contêineres fazem. O Kubernetes é um exemplo de orquestrador e está disponível no Azure por meio do Serviço de Kubernetes do Azure.

**Agendadores.** *Agendamento* é a capacidade do administrador de iniciar contêineres em um cluster para que eles também forneçam uma interface do usuário. O agendador do cluster tem diversas responsabilidades: usar os recursos de cluster de forma eficiente, definir as restrições fornecidas pelo usuário, balancear a carga dos contêineres em nós ou hosts de maneira eficaz, ser robusto contra erros e, ao mesmo tempo, oferecer alta disponibilidade.

Os conceitos de "cluster" e "agendador" estão intimamente relacionados, então os produtos oferecidos por diferentes fornecedores geralmente têm as duas capacidades. A lista a seguir mostra a plataforma mais importante e opções de software para clusters e agendadores. Geralmente, esses orquestradores são oferecidos em nuvens públicas como o Azure.

## Plataformas de software para clustering, orquestração e agendamento de contêineres

<b>Kubernetes</b> 	O <a href="#">Kubernetes</a> é um produto de software livre que oferece funcionalidades que variam da infraestrutura do cluster e do agendamento de contêiner a capacidades de orquestração. Com ele, é possível automatizar a implantação, o escalonamento e as operações de contêineres de aplicativo em clusters de hosts.  O <a href="#">Kubernetes</a> oferece uma infraestrutura centrada no contêiner que agrupa contêineres de aplicativo em unidades lógicas para facilitar o gerenciamento e a descoberta.  O <a href="#">Kubernetes</a> é maduro no Linux e menos maduro no Windows.
<b>AKS (Serviço de Kubernetes do Azure)</b> 	O <a href="#">AKS</a> é um serviço gerenciado de orquestração de contêineres Kubernetes no Azure que simplifica o gerenciamento, a implantação e as operações do cluster Kubernetes.

## Usar orquestradores baseados em contêiner no Microsoft Azure

Diversos provedores de nuvem oferecem suporte a contêineres do Docker e seus clusters e orquestração, como o Microsoft Azure, o Amazon EC2 Container Service e o Google Container Engine. O Microsoft Azure fornece suporte ao orquestrador e ao cluster do Docker por meio do AKS (Serviço de Kubernetes do Azure).

## Usando o Serviço de Kubernetes do Azure

Um cluster do Kubernetes cria um pool de diversos hosts do Docker e os expõe como um host virtual único. Assim, é possível implantar vários contêineres no cluster e expandir com qualquer número de instâncias de

contêiner. O cluster lidará com todos os detalhes complexos de gerenciamento, como escalabilidade, integridade e assim por diante.

O AKS é uma maneira de simplificar a criação, configuração e gerenciamento de um cluster de máquinas virtuais no Azure pré-configuradas para executar aplicativos em contêineres. Ao utilizar uma configuração otimizada de ferramentas de software livre conhecidas de agendamento e orquestração, o AKS permite o uso de habilidades existentes ou recorre ao grande e crescente conhecimento da comunidade para implantar e gerenciar aplicativos baseados em contêiner no Microsoft Azure.

O Serviço de Kubernetes do Azure otimiza a configuração de ferramentas de software livre e tecnologias conhecidas do Docker especificamente para o Azure. É uma solução aberta que oferece portabilidade para contêineres e configuração de aplicativo. Selecione o tamanho, a quantidade de hosts e as ferramentas de orquestrador e deixe o AKS cuidar de todo o resto.

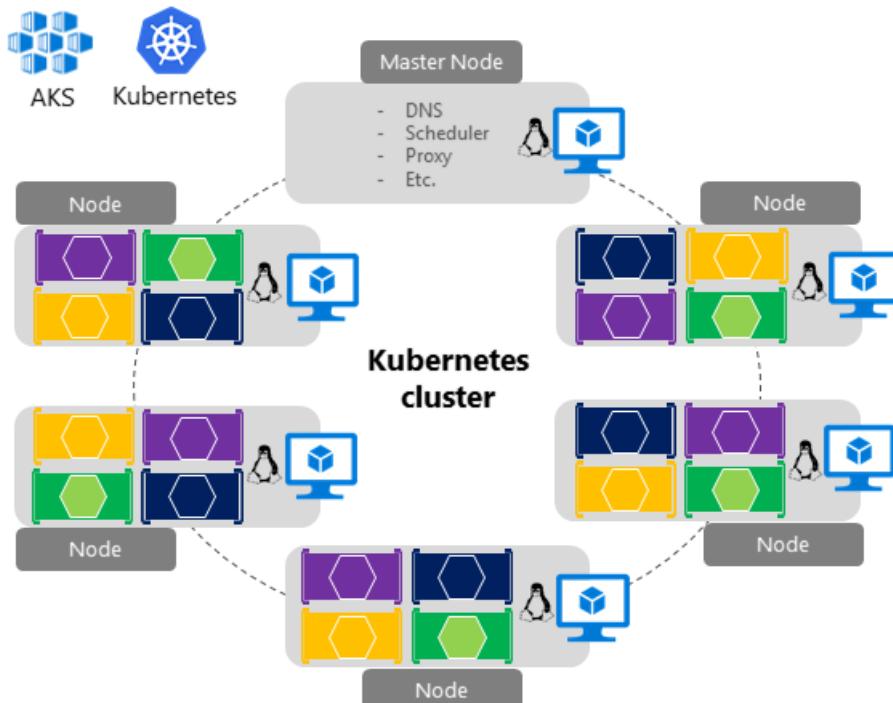


Figura 4-24. Topologia e estrutura simplificadas do cluster Kubernetes

Na figura 4-24 veja a estrutura de um cluster Kubernetes em que um nó mestre (VM) controla a maior parte da coordenação do cluster e você pode implantar contêineres no restante dos nós que são gerenciados como um único pool de um ponto de vista do aplicativo, permitindo dimensionar para milhares ou até mesmo dezenas de milhares de contêineres.

## Ambiente de desenvolvimento para Kubernetes

No ambiente de desenvolvimento, o [Docker anunciou em julho de 2018](#) que o Kubernetes também pode ser executado em um único computador de desenvolvimento (Windows 10 ou macOS) simplesmente instalando o [Docker Desktop](#). Posteriormente, você pode implantar para a nuvem (AKS) para testes de integração posteriores, conforme mostrado na figura 4-25.

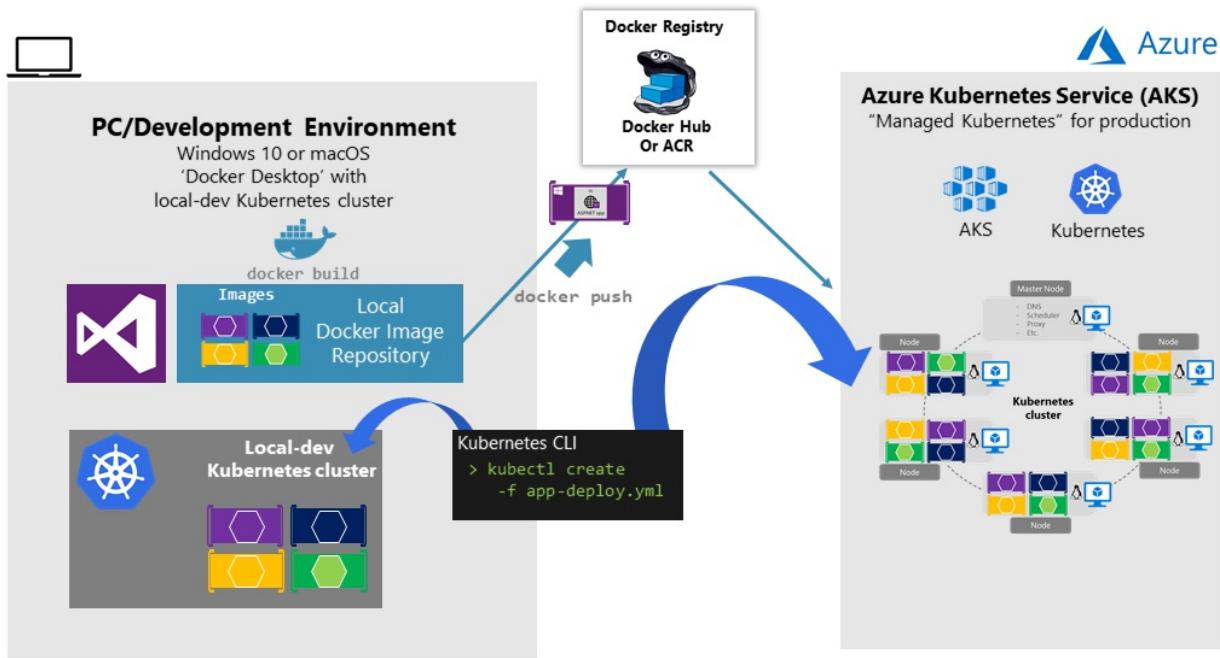


Figura 4-25. Executando Kubernetes no computador de desenvolvimento e na nuvem

## Introdução ao AKS (Serviço de Kubernetes do Azure)

Para começar a usar o AKS, implante um cluster do AKS do portal do Azure ou usando a CLI. Para saber mais sobre como implantar um cluster do Kubernetes no Azure, veja [Implantar um cluster do AKS \(Serviço de Kubernetes do Azure\)](#).

Nenhum valor é cobrado pelos softwares instalados por padrão como parte do AKS. Todas as opções padrão são implementadas com software livre. O AKS está disponível para várias máquinas virtuais no Azure. Somente as instâncias de computação escolhidas serão cobradas, bem como outros recursos adjacentes de infraestrutura consumidos, como armazenamento e rede. Não há cobranças adicionais pelo AKS.

A opção padrão de implantação de produção para Kubernetes é usar gráficos Helm, que é introduzido na próxima seção.

## Implantar com gráficos do Helm em clusters Kubernetes

Ao implantar um aplicativo em um cluster Kubernetes, você pode usar a ferramenta de CLI `kubectl.exe` original usando arquivos de implantação com base no formato nativo (arquivos `.yaml`), conforme já mencionado na seção anterior. No entanto, para aplicativos mais complexos do Kubernetes, como ao implantar aplicativos complexos baseados em microsserviços, é recomendável usar o [Helm](#).

Os Gráficos do Helm ajudam você a definir, realizar controle de versão, instalar, compartilhar, atualizar ou reverte até mesmo o aplicativo mais complexo do Kubernetes.

Indo mais além, o uso do Helm também é recomendável porque ambientes adicionais do Kubernetes no Azure, tais como [Azure Dev Spaces](#), também são baseados em gráficos do Helm.

O Helm é mantido pela [CNCF \(Fundação de Computação Nativa na Nuvem\)](#), em colaboração com Microsoft, Google, Bitnami e a comunidade de colaboradores do Helm.

Para obter mais informações sobre gráficos Helm e Kubernetes, consulte os [Gráficos de Uso de Helm para implantar eShopOnContainers no post AKS](#).

## Usar o Azure Dev Spaces para o ciclo de vida do aplicativo do

# Kubernetes

O Azure Dev Spaces oferece uma experiência rápida e iterativa de desenvolvimento kubernetes para equipes. Com algumas poucas configurações de máquina de desenvolvimento, você pode executar e depurar contêineres de forma iterativa diretamente no Azure Kubernetes Service (AKS). Desenvolva no Windows, Mac ou Linux usando ferramentas familiares como o Visual Studio, Visual Studio Code ou a linha de comando.

Conforme mencionado, o Azure Dev Spaces usa gráficos do Helm ao implantar os aplicativos baseados em contêiner.

O Azure Dev Spaces ajuda as equipes de desenvolvimento a serem mais produtivas no Kubernetes porque permite que você itera e depura rapidamente código diretamente em um cluster global kubernetes no Azure simplesmente usando o Visual Studio 2019 ou visual Studio Code. Esse cluster Kubernetes no Azure é um cluster Kubernetes gerenciado compartilhado, de modo que sua equipe pode trabalhar conjuntamente de forma colaborativa. Você pode desenvolver o código de forma isolada, depois implantar no cluster global e realizar testes de ponta a ponta com outros componentes sem replicar ou criar dependências fictícias.

Conforme mostrado na Figura 4-26, o recurso mais diferencial no Azure Dev Spaces é a capacidade de criar "espaços" que podem ser executados integrados ao restante da implantação global no cluster.

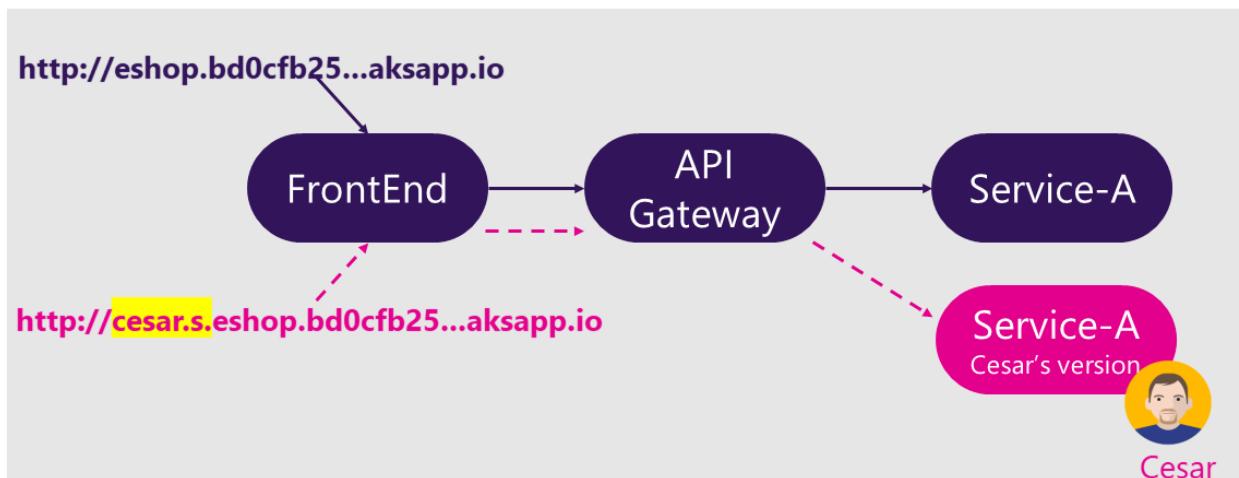


Figura 4-26. Usando vários espaços no Azure Dev Spaces

Basicamente, você pode configurar um espaço de desenvolvimento compartilhado no Azure. Cada desenvolvedor pode se concentrar apenas em sua parte do aplicativo e pode desenvolver iterativamente um código de pré-confirmação em um espaço de desenvolvimento que já contém todos os outros serviços e recursos de nuvem dos quais seus cenários dependem. As dependências estão sempre atualizadas, e os desenvolvedores trabalham de uma forma que reflete a produção.

O Azure Dev Spaces oferece o conceito de um espaço, que permite trabalhar em relativo isolamento e sem medo de interromper o trabalho da equipe. Cada espaço de desenvolvimento faz parte de uma estrutura hierárquica que permite que você substitua um ou muitos microsserviços a partir do espaço de desenvolvimento mestre "top" com seu próprio microsserviço em andamento.

Esse recurso se baseia nos prefixos de URL, portanto, ao usar qualquer prefixo de espaço de desenvolvimento na URL, uma solicitação é enviada do microsserviço de destino caso ele exista no espaço de desenvolvimento; caso contrário, ela é encaminhada até a primeira instância do microsserviço de destino encontrado na hierarquia, chegando ao espaço de desenvolvimento mestre na parte superior em algum momento.

Para obter uma visão prática sobre um exemplo concreto, consulte a [página wiki eShopOnContainers no Azure Dev Spaces](#).

Para obter mais informações, confira o artigo sobre [Desenvolvimento em equipe com o Azure Dev Spaces](#).

## Recursos adicionais

- Começando com o Azure Kubernetes Service (AKS)  
</azure/aks/kubernetes-walkthrough-portal>
- Espaços Azure Dev  
</azure/dev-spaces/azure-dev-spaces>
- **Kubernetes** O site oficial.  
<https://kubernetes.io/>

[PRÓXIMO](#)

[ANTERIOR](#)

# Processo de desenvolvimento para aplicativos baseados em Docker

18/03/2020 • 4 minutes to read • [Edit Online](#)

*Desenvolva aplicativos .NET contêiner do jeito que você gosta, seja o Ambiente de Desenvolvimento Integrado (IDE) focado com ferramentas do Visual Studio e do Visual Studio para Docker ou CLI/Editor focados com Docker CLI e Visual Studio Code.*

## Ambiente de desenvolvimento para aplicativos do Docker

### Opções de ferramenta de desenvolvimento: IDE ou editor

Seja qual for a sua preferência, um IDE avançado e completo ou um editor leve e ágil, a Microsoft oferece as ferramentas que você pode usar para desenvolver aplicativos do Docker.

**Visual Studio (para Windows).** O desenvolvimento de aplicativos .NET Core 3.1 baseado em Docker com o Visual Studio requer a versão 16.4 ou posterior do Visual Studio 2019. O Visual Studio 2019 vem com ferramentas para o Docker já incorporado. As ferramentas para Docker permitem desenvolver, executar e validar seus aplicativos diretamente no ambiente do Docker de destino. Você pode pressionar F5 para executar e depurar seu aplicativo (contêiner único ou vários contêineres) diretamente em um host do Docker ou pressionar CTRL + F5 para editar e atualizar o aplicativo sem precisar recompilar o contêiner. Essa é a opção de desenvolvimento mais eficiente para aplicativos baseados no Docker.

**Estúdio Visual para Mac.** É um IDE, evolução do Xamarin Studio, rodando no macOS. Para o desenvolvimento do .NET Core 3.1, ele requer a versão 8.4 ou posterior. Esta deve ser a escolha preferida para desenvolvedores que trabalham em máquinas macOS que também querem usar um IDE poderoso.

**Visual Studio Code e a CLI do Docker.** Se você preferir um editor leve e multiplataforma que suporte qualquer linguagem de desenvolvimento, você pode usar o Visual Studio Code e o Docker CLI. Esta é uma abordagem de desenvolvimento multiplataforma para macOS, Linux e Windows. Além disso, o Visual Studio Code dá suporte às extensões do Docker, como IntelliSense para Dockerfiles e tarefas de atalho, para executar os comandos do Docker usando o editor.

Ao instalar a [Área de trabalho do Docker CE \(Community Edition\)](#), você poderá usar uma única CLI do Docker para criar aplicativos para Windows e Linux.

### Recursos adicionais

- **Estúdio Visual.** Site oficial.  
<https://visualstudio.microsoft.com/vs/>
- **Visual Studio Code.** Site oficial.  
<https://code.visualstudio.com/download>
- **Docker Desktop para Windows Community Edition (CE)**  
<https://hub.docker.com/editions/community/docker-ce-desktop-windows>
- **Docker Desktop para Mac Community Edition (CE)**  
<https://hub.docker.com/editions/community/docker-ce-desktop-mac>

## Linguagens e estruturas do .NET para contêineres do Docker

Conforme mencionado nas seções anteriores deste guia, você pode usar o projeto do .NET Framework, do .NET

Core ou do Mono de software livre ao desenvolver aplicativos .NET em contêineres do Docker. Você poderá desenvolver em C#, em F# ou em Visual Basic ao direcionar a contêineres do Linux ou do Windows, dependendo de qual .NET Framework estiver em uso. Para saber mais detalhes sobre as linguagens do .NET detalhes, consulte a postagem no blog [The .NET Language Strategy](#) (A estratégia de linguagem do .NET).

[PRÓXIMO](#)

[ANTERIOR](#)

# Fluxo de trabalho de desenvolvimento para aplicativos do Docker

10/09/2020 • 52 minutes to read • [Edit Online](#)

O ciclo de vida de desenvolvimento de aplicativo começa em seu computador, como desenvolvedor, onde você pode codificar o aplicativo usando sua linguagem preferida e testá-lo localmente. Com esse fluxo de trabalho, não importa a linguagem, a estrutura e a plataforma escolhidas, você está sempre desenvolvendo e testando contêineres do Docker, mas localmente.

Cada contêiner (uma instância de uma imagem do Docker) inclui os seguintes componentes:

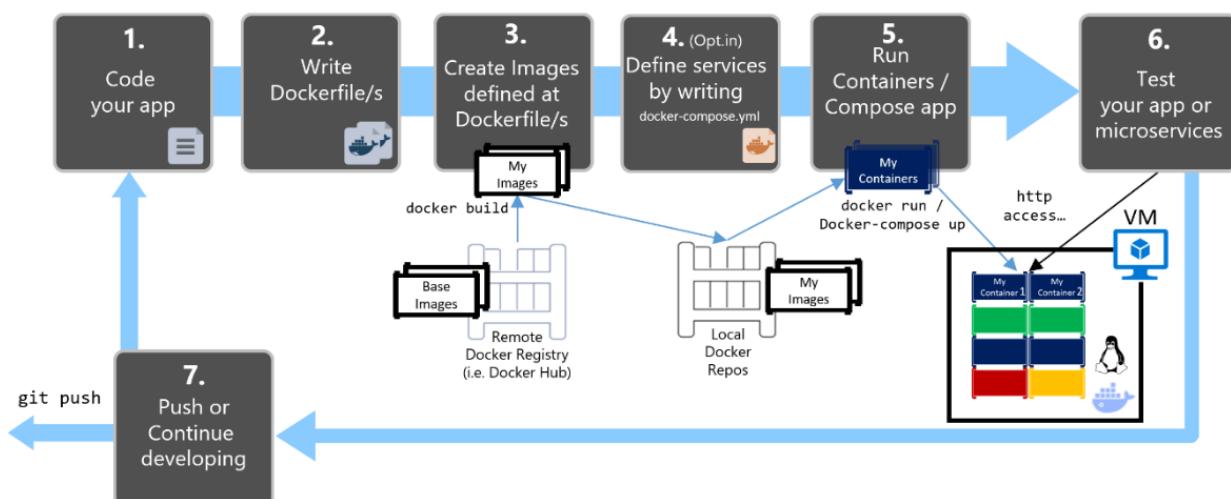
- Uma seleção de sistema operacional, por exemplo, uma distribuição do Linux, o Windows Nano Server ou o Windows Server Core.
- Arquivos adicionados durante o desenvolvimento, por exemplo, os binários e o código-fonte do aplicativo.
- Informações de configuração, como configurações de ambiente e dependências.

## Fluxo de trabalho para o desenvolvimento de aplicativos baseados em contêiner do Docker

Esta seção descreve o fluxo de trabalho de desenvolvimento do *loop interno* de aplicativos baseados em contêiner do Docker. O fluxo de trabalho de loop interno significa que ele não está considerando o fluxo de trabalho de DevOps mais amplo, que pode incluir até a implantação de produção e só se concentra na tarefa de desenvolvimento feita no computador do desenvolvedor. As etapas iniciais para configurar o ambiente não são incluídas, pois elas são executadas apenas uma vez.

Um aplicativo é composto de seus próprios serviços e de bibliotecas adicionais (dependências). A figura 5-1 a seguir ilustra as etapas básicas que são normalmente realizadas para criar um aplicativo do Docker.

### Inner-Loop development workflow for Docker apps



O processo de desenvolvimento para aplicativos Docker: 1-codificar seu aplicativo, 2-gravar Dockerfile/s, 3-criar imagens definidas em Dockerfile/s, 4-(opcional) compor serviços no arquivo Docker-Compose, 5-executar o contêiner ou o aplicativo Docker-Compose, 6-testar seu aplicativo ou seus microserviços, 7-enviar para o repositório e repetir

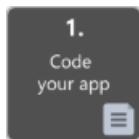
## Figura 5-1. Fluxo de trabalho passo a passo para o desenvolvimento de aplicativos do Docker em contêineres

Nesta seção, todo esse processo é detalhado e cada etapa principal é explicada com base em um ambiente do Visual Studio.

Ao usar uma abordagem de desenvolvimento de editor/CLI (por exemplo, o Visual Studio Code com a CLI do Docker no macOS ou no Windows), você precisará conhecer cada etapa e, na maioria das vezes, com mais detalhes do que ao usar o Visual Studio. Para obter mais informações sobre como trabalhar em um ambiente de CLI, confira o livro eletrônico [Containerized Docker Application lifecycle with Microsoft Platforms and Tools](#) (Ciclo de vida do aplicativo do Docker em contêineres com ferramentas e plataformas da Microsoft).

Quando você está usando o Visual Studio 2019, muitas dessas etapas são tratadas para você, o que melhora drasticamente a sua produtividade. Isso é especialmente verdadeiro quando você está usando o Visual Studio 2019 e direcionando aplicativos de vários contêineres. Por exemplo, com apenas um clique do mouse, o Visual Studio adiciona o `Dockerfile` `docker-compose.yml` arquivo e aos seus projetos com a configuração do seu aplicativo. Ao executar o aplicativo no Visual Studio, ele cria a imagem do Docker e executa o aplicativo de vários contêineres diretamente no Docker; e até mesmo permite que você depure vários contêineres de uma só vez. Esses recursos vão acelerar sua velocidade de desenvolvimento.

No entanto, apenas porque o Visual Studio automatiza essas etapas não significa que você não precisa saber o que está acontecendo nos bastidores com o Docker. Portanto, as diretrizes a seguir detalham cada etapa.



### Etapa 1. Iniciar a codificação e a criação do aplicativo inicial ou da linha de base do serviço

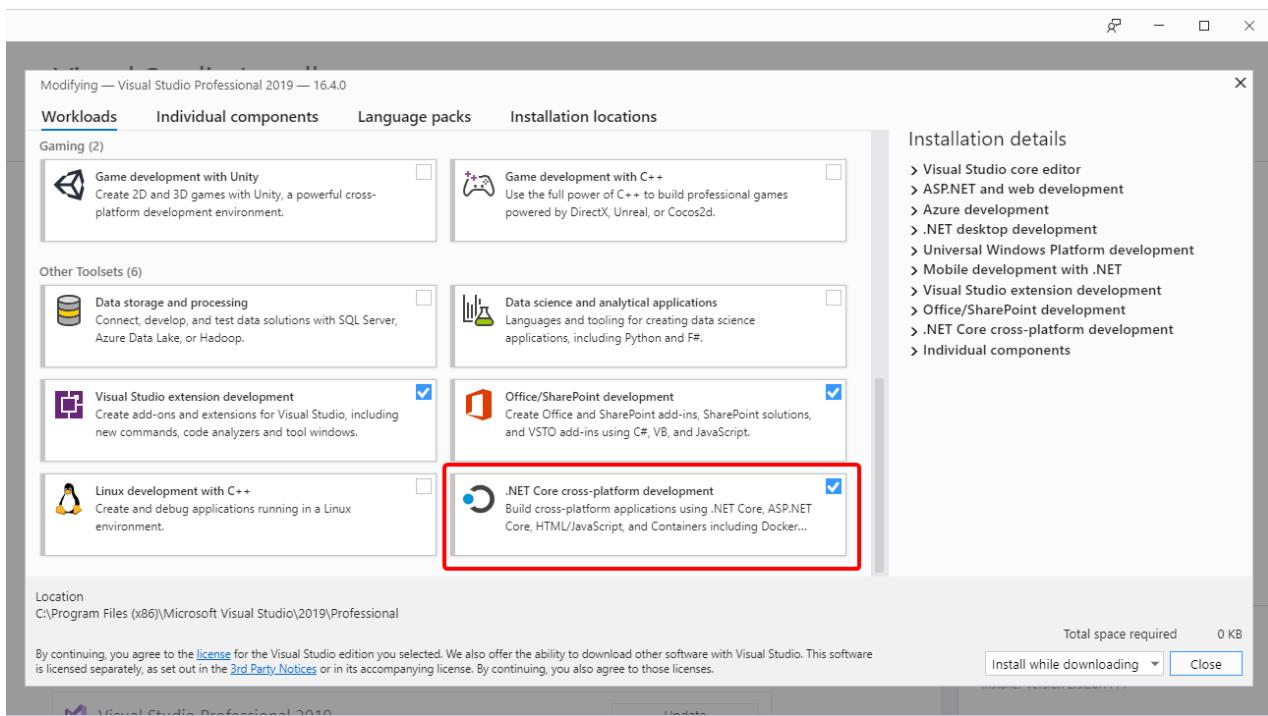
O desenvolvimento de um aplicativo do Docker é semelhante à forma como você desenvolve um aplicativo sem Docker. A diferença é que, durante o desenvolvimento para o Docker, você está implantando e testando o aplicativo ou os serviços executando-os em contêineres do Docker em seu ambiente local (seja em uma VM do Linux configurada pelo Docker ou diretamente no Windows quando está usando contêineres do Windows).

#### Configurar seu ambiente local com o Visual Studio

Para começar, instale o [Docker CE \(Community Edition\)](#) para Windows, conforme explicado nas instruções a seguir:

#### [Introdução ao Docker CE for Windows](#)

Além disso, você precisa do Visual Studio 2019 versão 16,4 ou posterior, com a carga de trabalho de **desenvolvimento de plataforma cruzada do .NET Core** instalada, conforme mostrado na Figura 5-2.



**Figura 5-2.** Selecionando a carga de trabalho de desenvolvimento de plataforma cruzada do .NET Core durante a instalação do Visual Studio 2019

Você pode iniciar a codificação do aplicativo no .NET simples (normalmente no .NET Core, se estiver planejando usar contêineres) mesmo antes de habilitar o Docker no aplicativo e antes da implantação e dos testes no Docker. No entanto, é recomendável que você comece a trabalhar no Docker assim que possível, porque que esse será o ambiente real e os problemas poderão ser descobertos o mais rápido possível. Isso é recomendável porque o Visual Studio facilita tanto o trabalho com o Docker parecendo, até mesmo, transparente — o melhor exemplo ao depurar aplicativos de vários contêineres do Visual Studio.

### Recursos adicionais

- **Introdução ao Docker CE for Windows**  
<https://docs.docker.com/docker-for-windows/>
- **Visual Studio 2019**  
<https://visualstudio.microsoft.com/downloads/>



## Etapa 2. Criar um Dockerfile relacionado a uma imagem base existente do .NET

Você precisa de um Dockerfile para cada imagem personalizada que você deseja criar. Você também precisará de um Dockerfile para cada contêiner a ser implantado automaticamente, por meio do Visual Studio, ou implantado manualmente, usando a CLI do Docker (comandos docker run e docker-compose). Se seu aplicativo contém um único serviço personalizado, é necessário um único Dockerfile. Se seu aplicativo contém vários serviços (como em uma arquitetura de microserviços), é necessário um Dockerfile para cada serviço.

O Dockerfile é colocado na pasta raiz do seu aplicativo ou serviço. Ele contém os comandos que indicam ao Docker como configurar e executar seu aplicativo ou serviço em um contêiner. Você pode criar manualmente um Dockerfile por meio de código e adicioná-lo ao seu projeto junto com as dependências do .NET.

Com o Visual Studio e suas ferramentas para Docker, esta tarefa requer apenas alguns cliques do mouse. Quando

você cria um novo projeto no Visual Studio 2019, há uma opção chamada **habilitar suporte do Docker**, como mostra a Figura 5-3.

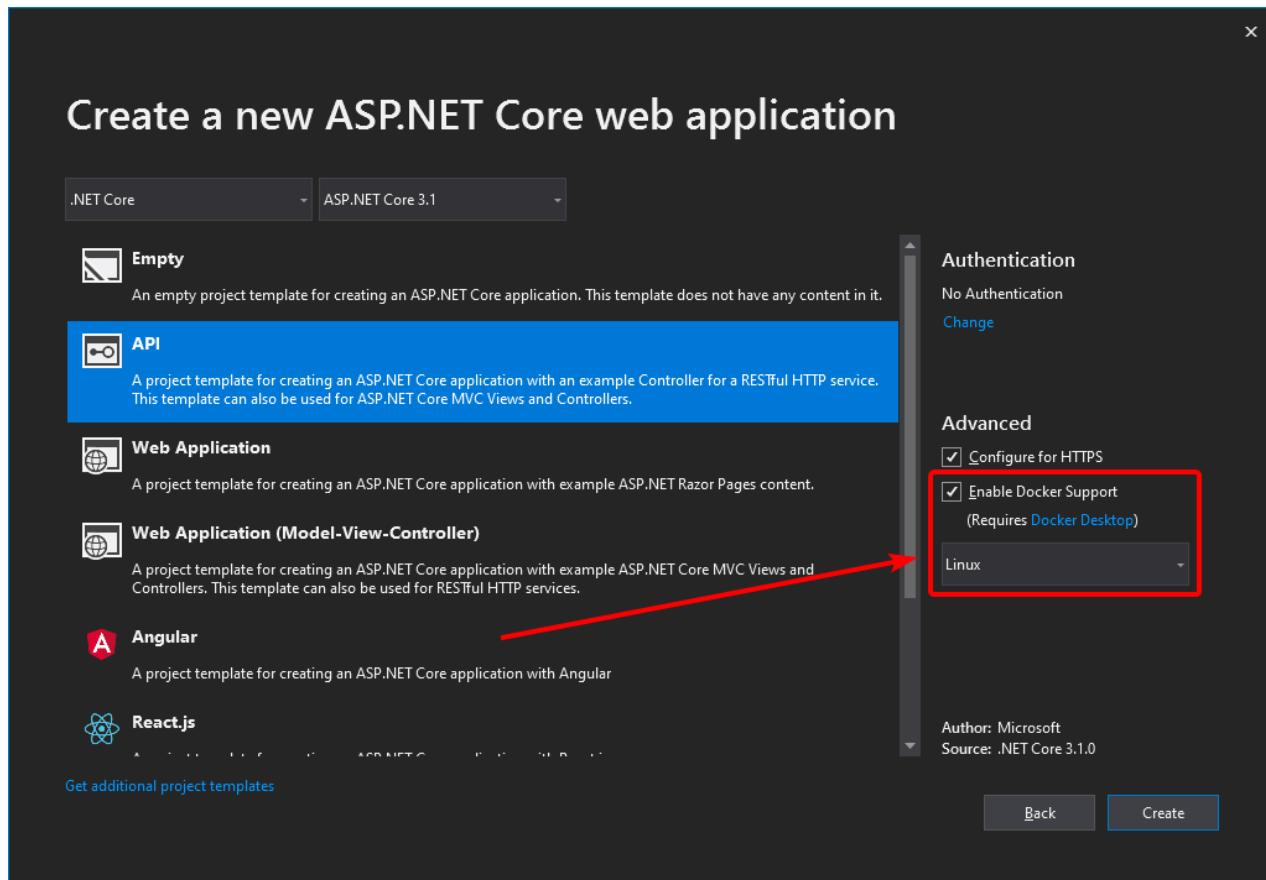


Figura 5-3. Habilitando o suporte do Docker ao criar um novo projeto de ASP.NET Core no Visual Studio 2019

Você também pode habilitar o suporte do Docker em um projeto de aplicativo Web ASP.NET Core existente clicando com o botão direito do mouse no projeto em **Gerenciador de soluções** e selecionando **Adicionar > suporte ao Docker...**, conforme mostrado na Figura 5-4.

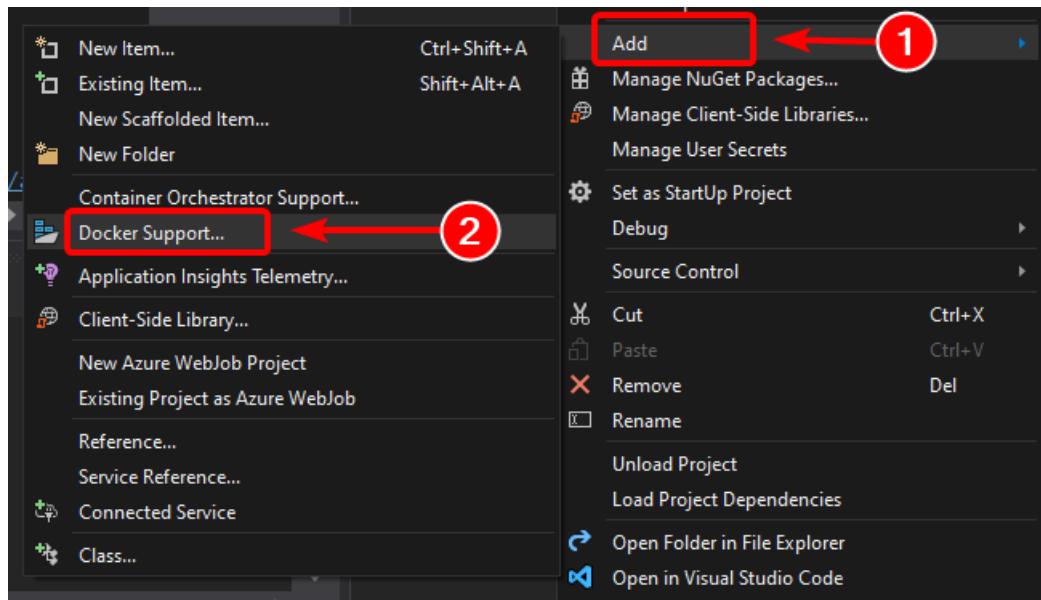


Figura 5-4. Habilitando o suporte do Docker em um projeto existente do Visual Studio 2019

Essa ação adiciona um *Dockerfile* ao projeto com a configuração necessária e só está disponível em projetos ASP.NET Core.

De maneira semelhante, o Visual Studio também pode adicionar um `docker-compose.yml` arquivo para a solução inteira com a opção **Adicionar > contêiner suporte ao orquestrador...**. Na etapa 4, exploraremos essa opção

mais detalhadamente.

## Usando uma imagem do Docker do .NET oficial existente

Geralmente, você cria uma imagem personalizada para o contêiner usando uma imagem base que você pode obter de um repositório oficial, como o Registro do [Docker Hub](#). É exatamente isso o que acontece nos bastidores quando você habilita o suporte do Docker no Visual Studio. O Dockerfile usará uma imagem `dotnet/core/aspnet` existente.

Anteriormente, explicamos quais imagens do Docker e quais repositórios você poderia usar, dependendo da estrutura e do sistema operacional que você escolheu. Por exemplo, se você quiser usar o ASP.NET Core (Linux ou Windows), a imagem a ser usada será `mcr.microsoft.com/dotnet/core/aspnet:3.1`. Assim, basta especificar qual imagem base do Docker será usada para o seu contêiner. Você pode fazer isso adicionando

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
```

 ao seu Dockerfile. Isso será realizado automaticamente pelo Visual Studio, mas se você precisar atualizar a versão, será esse valor que você atualizará.

O uso de um repositório de imagem oficial do .NET do Hub do Docker com um número de versão garante que os mesmos recursos de linguagem estejam disponíveis em todos os computadores (incluindo desenvolvimento, teste e produção).

O exemplo a seguir mostra um Dockerfile de exemplo para um contêiner do ASP.NET Core.

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "MySingleContainerWebApp.dll"]
```

Nesse caso, a imagem é baseada na versão 3,1 do oficial ASP.NET Core imagem do Docker (vários arcos para Linux e Windows). Essa é a configuração `FROM mcr.microsoft.com/dotnet/core/aspnet:3.1`. (Para obter mais informações sobre essa imagem base, consulte a página [imagem do Docker do .NET Core](#).) No Dockerfile, você também precisa instruir o Docker para escutar na porta TCP que será usada no tempo de execução (nesse caso, a porta 80, conforme configurado com a configuração de exposição).

Você pode especificar mais definições de configurações no Dockerfile, dependendo da linguagem e da estrutura usadas. Por exemplo, a linha `ENTRYPOINT` com `["dotnet", "MySingleContainerWebApp.dll"]` diz ao Docker para executar um aplicativo do .NET Core. Se você estiver usando o SDK e a CLI do .NET Core (CLI do dotnet) para compilar e executar o aplicativo do .NET, essa configuração será diferente. O essencial é que a linha `ENTRYPOINT` e outras configurações serão diferentes dependendo da linguagem e da plataforma que você escolher para seu aplicativo.

## Recursos adicionais

- **Criando imagens do Docker para aplicativos .NET Core**  
<https://docs.microsoft.com/dotnet/core/docker/building-net-docker-images>
- **Criar sua própria imagem.** Na documentação oficial do Docker.  
<https://docs.docker.com/engine/tutorials/dockerimages/>
- **Manter-se atualizado com as imagens de contêiner do .NET**  
<https://devblogs.microsoft.com/dotnet/staying-up-to-date-with-net-container-images/>
- **Usando o .NET e o Docker juntos – atualização do DockerCon 2018**  
<https://devblogs.microsoft.com/dotnet/using-net-and-docker-together-dockercon-2018-update/>

## Usando repositórios de imagens para várias arquiteturas

Um único repositório pode conter variantes de plataforma, como uma imagem do Linux e uma imagem do

Windows. Esse recurso permite que fornecedores, como a Microsoft (criadores de imagem base), criem um único repositório para abranger várias plataformas (ou seja, Linux e Windows). Por exemplo, o repositório [dotnet/core](#), disponível no registro do Hub do Docker, é compatível com Linux e Windows Nano Server usando o mesmo nome de repositório.

Ao especificar uma marcação, você direciona a uma plataforma que é explícita, como nos seguintes casos:

- `mcr.microsoft.com/dotnet/core/aspnet:3.1-buster-slim`

Destinos: somente em tempo de execução do .NET Core 3,1 no Linux

- `mcr.microsoft.com/dotnet/core/aspnet:3.1-nanoserver-1909`

Destinos: somente em tempo de execução do .NET Core 3,1 no Windows nano Server

Mas, se você especificar o mesmo nome de imagem, mesmo com a mesma marca, as imagens para várias arquiteturas (como a imagem `aspnet`) usarão a versão do Linux ou do Windows, dependendo do sistema operacional do host do Docker em que você esteja implantando, conforme mostrado no exemplo a seguir:

- `mcr.microsoft.com/dotnet/core/aspnet:3.1`

Vários arcos: tempo de execução do .NET Core 3,1 somente no Linux ou Windows nano Server, dependendo do sistema operacional do host do Docker

Dessa forma, ao efetuar pull de uma imagem de um host do Windows, será efetuado o pull da variante do Windows e, ao efetuar pull do mesmo nome de imagem de um host do Linux, será efetuado pull da variante do Linux.

### Builds de vários estágios no Dockerfile

O Dockerfile é semelhante a um script em lotes. Como o que você faria se tivesse que configurar o computador da linha de comando.

Ele começa com uma imagem base que define o contexto inicial, como o sistema de arquivos de inicialização, que se baseia no sistema operacional do host. Não é um sistema operacional, mas você pode considerá-lo como "o sistema operacional dentro do contêiner".

A execução de cada linha de comando cria uma camada no sistema de arquivos com as alterações em relação à anterior, para que, quando combinadas, produzam o sistema de arquivos resultante.

Como cada nova camada é baseada na anterior e o tamanho da imagem resultante aumenta com cada comando, as imagens poderão ficar muito grandes se precisarem incluir, por exemplo, o SDK necessário para criar e publicar um aplicativo.

É nesse ponto que os builds de vários estágios (do Docker 17.05 e superior) entram em cena para fazer a mágica.

A ideia central é que você pode separar o processo de execução do Dockerfile em estágios, em que um estágio é uma imagem inicial seguida por um ou mais comandos, e o último estágio determina o tamanho da imagem final.

Resumindo, os builds de vários estágios permitem dividir a criação em "fases" diferentes e, em seguida, montar a imagem final usando somente os diretórios relevantes dos estágios intermediários. A estratégia geral para usar esse recurso é:

1. usar uma imagem base do SDK (não importa quão grande), com todos os componentes necessários para criar e publicar o aplicativo em uma pasta e, em seguida,
2. usar uma imagem base pequena, somente em runtime e copiar a pasta de publicação do estágio anterior para produzir uma pequena imagem final.

Provavelmente a melhor maneira de entender os vários estágios é percorrer um Dockerfile detalhadamente, linha por linha, então, vamos começar com o Dockerfile inicial criado pelo Visual Studio ao adicionar o suporte ao Docker a um projeto e entraremos em algumas otimizações mais tarde.

O Dockerfile inicial pode ser como este:

```
1 FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS base
2 WORKDIR /app
3 EXPOSE 80
4
5 FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS build
6 WORKDIR /src
7 COPY src/Services/Catalog/Catalog.API/Catalog.API.csproj ...
8 COPY src/BuildingBlocks/HealthChecks/src/Microsoft.AspNetCore.HealthChecks ...
9 COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions.HealthChecks ...
10 COPY src/BuildingBlocks/EventBus/IntegrationEventLogEF/ ...
11 COPY src/BuildingBlocks/EventBus/EventBus/EventBus.csproj ...
12 COPY src/BuildingBlocks/EventBus/EventBusRabbitMQ/EventBusRabbitMQ.csproj ...
13 COPY src/BuildingBlocks/EventBus/EventBusServiceBus/EventBusServiceBus.csproj ...
14 COPY src/BuildingBlocks/WebHostCustomization/WebHost.Customization ...
15 COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions ...
16 COPY src/BuildingBlocks/HealthChecks/src/Microsoft.Extensions ...
17 RUN dotnet restore src/Services/Catalog/Catalog.API/Catalog.API.csproj
18 COPY . .
19 WORKDIR /src/src/Services/Catalog/Catalog.API
20 RUN dotnet build Catalog.API.csproj -c Release -o /app
21
22 FROM build AS publish
23 RUN dotnet publish Catalog.API.csproj -c Release -o /app
24
25 FROM base AS final
26 WORKDIR /app
27 COPY --from=publish /app .
28 ENTRYPOINT ["dotnet", "Catalog.API.dll"]
```

E estes são os detalhes, linha por linha:

- **#1 de linha:** Inicie um estágio com uma imagem base somente de tempo de execução "pequeno", chame-a **base** para referência.
- **#2 de linha:** Crie o diretório **/app** na imagem.
- **#3 de linha:** Exportar a porta **80**.
- **#5 de linha:** Comece um novo estágio com a imagem "grande" para criação/publicação. Chame-o **Build** para referência.
- **#6 de linha:** Crie o diretório **/src** na imagem.
- **#7 de linha:** Até a linha 16, copie os arquivos de projeto **. csproj** referenciados para poder restaurar pacotes posteriormente.
- **#17 de linha:** Restaure os pacotes para o projeto **Catalog. API** e os projetos referenciados.
- **#18 de linha:** Copie toda a árvore de diretórios da solução (exceto os arquivos/diretórios incluídos no arquivo **.dockerignore**) para o diretório **/src** na imagem.
- **#19 de linha:** Altere a pasta atual para o projeto **Catalog. API**.
- **#20 de linha:** Compile o projeto (e outras dependências do projeto) e a saída para o diretório **/app** na imagem.
- **#22 de linha:** Iniciar um novo estágio continuando a partir da compilação. Chame-o de **publicar** para referência.
- **#23 de linha:** Publique o projeto (e as dependências) e a saída para o diretório **/app** na imagem.
- **#25 de linha:** Inicie um novo estágio continuando da **base** e chame-o **final**.

- #26 de linha: Altere o diretório atual para /app.
- #27 de linha: Copie o diretório /app do estágio Publish para o diretório atual.
- #28 de linha: Defina o comando a ser executado quando o contêiner for iniciado.

Agora vamos explorar algumas otimizações para melhorar o desempenho de todo o processo que, no caso do eShopOnContainers, significa cerca de 22 minutos ou mais para criar a solução completa em contêineres do Linux.

Você aproveitará o recurso de cache de camada do Docker, que é bastante simples: se a imagem base e os comandos forem iguais a outros já executados, ele poderá usar apenas a camada resultante sem precisar executar os comandos, economizando tempo.

Portanto, vamos nos concentrar no estágio **build**. As linhas 5 a 6 são basicamente as mesmas, mas as linhas 7 a 17 são diferentes para cada serviço do eShopOnContainers, portanto precisam ser executadas todas as vezes, no entanto, se você tiver alterado as linhas 7 a 16 para:

```
COPY . .
```

Ele seria igual para cada serviço, ele poderia copiar toda a solução e criar uma camada maior, mas:

1. o processo de cópia seria executado somente na primeira vez (e durante a recriação, se algum arquivo fosse alterado) e usaria o cache para todos os outros serviços e
2. como a imagem maior ocorre em um estágio intermediário, ela não afeta o tamanho da imagem final.

A próxima otimização significativa envolve o comando `dotnet restore` executado na linha 17, que também é diferente para cada serviço do eShopOnContainers. Se você alterar essa linha para apenas:

```
RUN dotnet restore
```

Isso restauraria os pacotes de toda a solução, mas, novamente, isso seria feito apenas uma vez, em vez de 15 vezes com a estratégia atual.

No entanto, `dotnet restore` só será executado se houver um único arquivo de projeto ou de solução na pasta, portanto, isso é um pouco mais complicado e a maneira de resolvê-lo, sem entrar em muitos detalhes, é esta:

1. adicione as seguintes linhas ao `.dockerignore`:
  - `*.sln`, para ignorar todos os arquivos de solução na árvore da pasta principal
  - `!eShopOnContainers-ServicesAndWebApps.sln`, para incluir apenas esse arquivo de solução.
2. inclua o argumento `/ignoreprojectextensions:.dcproj` em `dotnet restore`, para que ele também ignore o projeto docker-compose e só restaure os pacotes da solução eShopOnContainers-ServicesAndWebApps.

Para a otimização final, acontece que a linha 20 é redundante e a linha 23 também compila o aplicativo e vem, naturalmente, logo após a linha 20, o que gera outro comando demorado.

O arquivo resultante é:

```

1 FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS base
2 WORKDIR /app
3 EXPOSE 80
4
5 FROM mcr.microsoft.com/dotnet/core/sdk:3.1 AS publish
6 WORKDIR /src
7 COPY . .
8 RUN dotnet restore /ignorereprojectextensions:.dcproj
9 WORKDIR /src/src/Services/Catalog/Catalog.API
10 RUN dotnet publish Catalog.API.csproj -c Release -o /app
11
12 FROM base AS final
13 WORKDIR /app
14 COPY --from=publish /app
15 ENTRYPOINT ["dotnet", "Catalog.API.dll"]

```

### Criação da imagem base do zero

Você pode criar sua própria imagem base do Docker do zero. Esse cenário não é recomendável para um iniciante no Docker, mas se você desejar definir os bits específicos da sua própria imagem base, isso será possível.

### Recursos adicionais

- **Imagens do .NET Core em vários arcos.**  
<https://github.com/dotnet/announcements/issues/14>
- **Criar uma imagem base.** Documentação oficial do Docker.  
<https://docs.docker.com/develop/develop-images/baseimages/>



## Etapa 3. Criar imagens personalizadas do Docker e inserir seu aplicativo ou serviço nelas

Para cada serviço do seu aplicativo, você precisa criar uma imagem relacionada. Se seu aplicativo é composto de um único serviço ou aplicativo Web, você precisa apenas de uma única imagem.

Observe que as imagens do Docker são criadas automaticamente para você no Visual Studio. As etapas a seguir são necessárias apenas para o fluxo de trabalho de editor/CLI e são explicadas com a finalidade de esclarecer o que acontece nos bastidores.

Você, como desenvolvedor, precisa desenvolver e testar localmente até enviar um recurso concluído por push ou mudar para o sistema de controle do código-fonte (por exemplo, para o GitHub). Isso significa que você precisa criar as imagens do Docker e implantar contêineres em um host do Docker local (VM do Windows ou Linux) e executar, testar e depurar nesses contêineres locais.

Para criar uma imagem personalizada no seu ambiente local, usando a CLI do Docker e o Dockerfile, você pode usar o comando docker build, como na Figura 5-5.

```

PS C:\dev\netcore-webapi-microservice-docker> docker build -t cesardl/netcore-webapi-microservice-docker:first .
Sending build context to Docker daemon 1.148 MB
Step 1 : FROM microsoft/dotnet:latest
latest: Pulling from microsoft/dotnet
5c90d4a2d1a8: Downloading [=====>] 18.34 MB/51.35 MB
ab30c63719b1: Downloading [=====>] 18.48 MB/18.55 MB
c6072700a242: Downloading [=====>] 18.34 MB/42.53 MB
121d7eeef6c20: Waiting
eb57cf4f29ee: Waiting
b2c5ae2d325b: Waiting

```

Figura 5-5. Criando uma imagem personalizada do Docker

Opcionalmente, em vez de executar diretamente o comando docker build na pasta do projeto, você pode, primeiro, gerar uma pasta implantável com as bibliotecas e os binários do .NET necessários, executando `dotnet publish` e, em seguida, usar o comando `docker build`.

Isso criará uma imagem do Docker com o nome `cesardl/netcore-webapi-microservice-docker:first`. Nesse caso, `:first` é uma marcação que representa uma versão específica. Você poderá repetir esta etapa para cada imagem personalizada que tiver que criar para o seu aplicativo composto do Docker.

Quando um aplicativo é composto de vários contêineres (ou seja, ele é um aplicativo multicontêineres), você também pode usar o comando `docker-compose up --build` para compilar todas as imagens relacionadas com um único comando usando os metadados expostos nos arquivos `docker-compose.yml` relacionados.

Você pode encontrar as imagens existentes no seu repositório local usando o comando `docker images`, conforme mostrado na Figura 5-6.

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
cesardl/netcore-webapi-microservice-docker	first	384c4ac1809b	4 minutes ago	579.8 MB
microsoft/dotnet	latest	49aaaf5daa850	30 hours ago	548.6 MB
ubuntu	latest	cf62323fa025	5 days ago	125 MB
hello-world	latest	c54a2cc56cbb	12 days ago	1.848 kb

Figura 5-6. Exibição de imagens existentes usando o comando `docker images`

### Criação de imagens do Docker com o Visual Studio

Ao usar o Visual Studio para criar um projeto com suporte ao Docker, você não cria explicitamente uma imagem. Em vez disso, a imagem é criada quando você pressiona F5 (ou Ctrl+F5) para executar o aplicativo ou o serviço convertido para Docker. Esta etapa é automática no Visual Studio e você não verá ela ocorrer, mas é importante que você saiba o que está acontecendo nos bastidores.



### Etapa 4. Definir os serviços no `docker-compose.yml` ao compilar um aplicativo de vários contêineres do Docker

O arquivo `docker-compose.yml` permite que você defina um conjunto de serviços relacionados para que sejam implantados como um aplicativo composto com comandos de implantação. Ele também configura as relações de dependência e a configuração de tempo de execução.

Para usar um arquivo `docker-compose.yml`, você precisa criar o arquivo na pasta principal ou raiz da sua solução, com conteúdo semelhante ao do exemplo a seguir:

```

version: '3.4'

services:

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

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

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

  sqldata:
    image: mcr.microsoft.com/mssql/server:latest
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"

```

Esse arquivo docker-compose.yml é uma versão simplificada e mesclada. Ele contém dados de configuração estáticos para cada contêiner (como o nome da imagem personalizada), o que é sempre necessário, além das informações de configuração que poderão depender do ambiente de implantação, como a cadeia de conexão. Nas próximas seções, você aprenderá como dividir a configuração do docker-compose.yml em vários arquivos docker-compose e substituir valores dependendo do ambiente e do tipo de execução (depuração ou lançamento).

O arquivo docker-compose.yml de exemplo define quatro serviços: o serviço `webmvc` (um aplicativo Web), dois microsserviços (`ordering-api` e `basket-api`) e um contêiner de fonte de dados, `sqldata`, com base no SQL Server para Linux, em execução como um contêiner. Cada serviço será implantado como um contêiner, portanto, uma imagem do Docker é necessária para cada um.

O arquivo docker-compose.yml não apenas especifica quais contêineres estão sendo usados, mas também como eles são configurados individualmente. Por exemplo, a definição de contêiner `webmvc` no arquivo .yml:

- usa uma imagem `eshop/web:latest` predefinida. Entretanto, você também poderia definir a imagem para ser compilada como parte da execução do docker-compose, com uma configuração adicional baseada em uma seção `build`: no arquivo docker-compose.
- Inicializa duas variáveis de ambiente (`CatalogUrl` e `OrderingUrl`).

- Encaminha a porta 80, exposta no contêiner, para a porta 80 externa, no computador host.
- vincula o aplicativo Web ao catálogo e ao serviço de pedidos com a configuração depends\_on. Isso faz com que o serviço aguarde até que esses serviços sejam iniciados.

Vamos abordar novamente o arquivo docker-compose.yml em uma seção posterior, ao explicar como implementar microsserviços e aplicativos de vários contêineres.

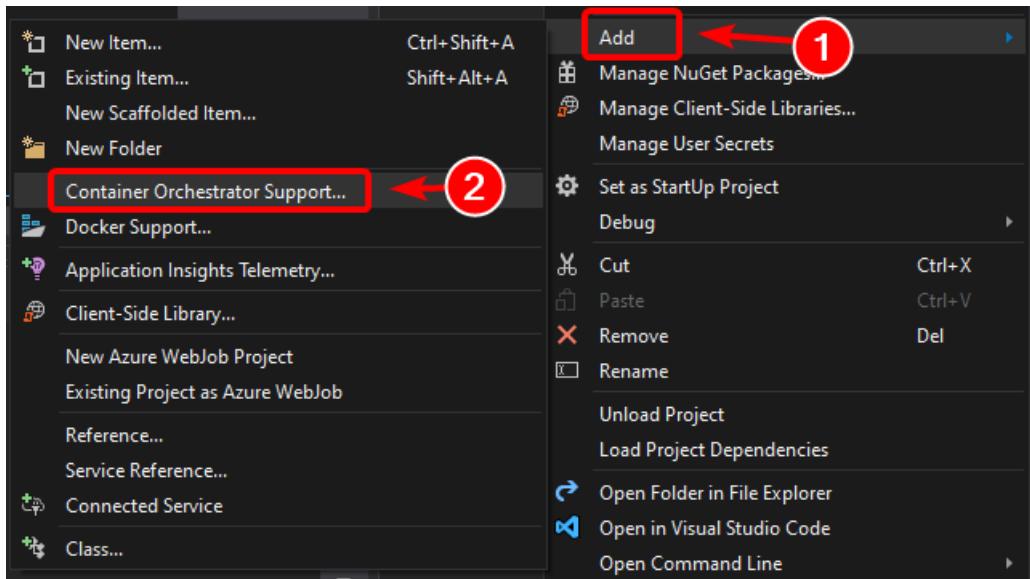
### Trabalhando com Docker-Compose. yml no Visual Studio 2019

Além de adicionar um Dockerfile a um projeto, como mencionamos anteriormente, o Visual Studio 2017 (da versão 15,8 em) pode adicionar suporte do Orchestrator para Docker Compose a uma solução.

Quando você adiciona o suporte ao orquestrador de contêineres, conforme mostrado na Figura 5-7, pela primeira vez, o Visual Studio cria o Dockerfile para o projeto e cria um projeto (seção de serviço) na solução com vários arquivos `docker-compose*.yml` globais e, em seguida, adiciona o projeto nesses arquivos. Depois, você pode abrir os arquivos docker-compose.yml e atualizá-los com mais recursos.

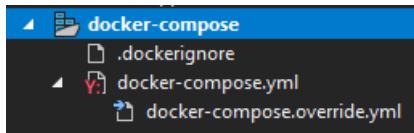
Você precisa repetir essa forma de operação para cada projeto que deseja incluir no arquivo docker-compose.yml.

No momento da redação deste artigo, o Visual Studio dá suporte aos orquestradores **Docker Compose** e **kubernetes/Helm**.



**Figura 5-7.** Adicionando suporte ao Docker no Visual Studio 2019 clicando com o botão direito do mouse em um projeto ASP.NET Core

Depois de adicionar suporte ao orquestrador à solução no Visual Studio, você também verá um novo nó (no arquivo de projeto `docker-compose.dcproj`) no Gerenciador de Soluções contendo os arquivos docker-compose.yml adicionados, conforme mostrado na Figura 5-8.



**Figura 5-8.** O nó da árvore Docker-Compose adicionado ao Visual Studio 2019 Gerenciador de soluções

Você pode implantar um aplicativo de vários contêineres com um único arquivo docker-compose.yml usando o comando `docker-compose up`. No entanto, o Visual Studio adiciona um grupo desses arquivos, para que você possa substituir os valores de acordo com o ambiente (desenvolvimento ou produção) e o tipo de execução (lançamento ou depuração). Essa funcionalidade será explicada nas seções posteriores.



## Etapa 5. Compilar e executar seu aplicativo do Docker

Se seu aplicativo tem apenas um contêiner, você pode executá-lo implantando-o no host do Docker (VM ou servidor físico). No entanto, se seu aplicativo contiver vários serviços, você poderá implantá-lo como um aplicativo composto, usando um único comando da CLI (`docker-compose up`) ou com o Visual Studio, que usará esse comando nos bastidores. Vamos examinar as diferentes opções.

### Opção A: executar um aplicativo de contêiner único

#### Usando a CLI do Docker

Você pode executar um contêiner do Docker usando o comando `docker run`, conforme mostrado na Figura 5-9:

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

O comando acima criará uma instância de contêiner da imagem especificada, cada vez que for executado. Você pode usar o `--name` parâmetro para dar um nome ao contêiner e, em seguida, usar `docker start {name}` (ou usar a ID do contêiner ou o nome automático) para executar uma instância de contêiner existente.

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

Figura 5-9. Executando um contêiner do Docker usando o comando `docker run`

Nesse caso, o comando associa a porta interna 5000 do contêiner à porta 80 do computador host. Isso significa que o host está escutando na porta 80 e encaminhando para a porta 5000 no contêiner.

O hash mostrado é a ID do contêiner e também é atribuído um nome legível aleatório se a `--name` opção não for usada.

#### Como usar o Visual Studio

Se ainda não adicionou o suporte ao orquestrador de contêineres, você também poderá executar um aplicativo de contêiner único no Visual Studio pressionando **Ctrl+F5** e usar **F5** para depurar o aplicativo dentro do contêiner. O contêiner é executado localmente usando o `docker run`.

### Opção B: executando um aplicativo de vários contêineres

Na maioria dos cenários empresariais, um aplicativo do Docker será ser composto de vários serviços, o que significa que você precisará executar um aplicativo de vários contêineres, conforme mostrado na Figura 5-10.

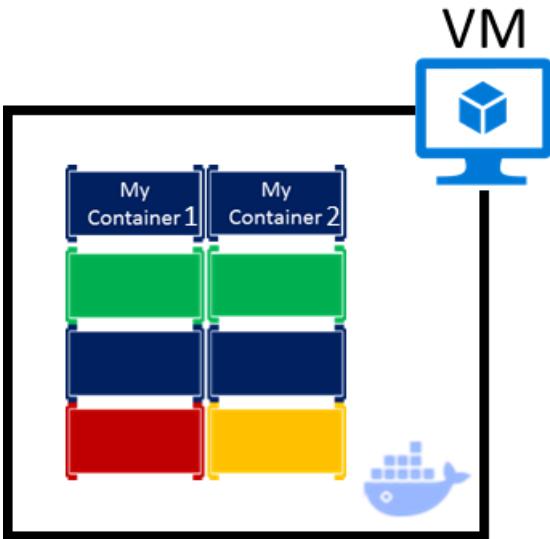


Figura 5-10. VM com contêineres do Docker implantados

#### Usando a CLI do Docker

Para executar um aplicativo de vários contêineres com a CLI do Docker, use o comando `docker-compose up`. Esse comando usa o arquivo `docker-compose.yml`, que existe no nível da solução, para implantar um aplicativo de vários contêineres. A figura 5-11 mostra os resultados da execução do comando no diretório principal da solução, que contém o arquivo `docker-compose.yml`.

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

Figura 5-11. Resultados de exemplo ao executar o comando `docker-compose up`

Depois que o comando `docker-compose up` é executado, o aplicativo e os contêineres relacionados são implantados no host do Docker, como é mostrado na Figura 5-10.

#### Como usar o Visual Studio

A execução de um aplicativo de vários contêineres usando o Visual Studio 2019 não pode ser mais simples. Basta pressionar **Ctrl+F5** para executar ou **F5** para depuração, como de costume, configurando o projeto `docker-compose` como o projeto de inicialização. O Visual Studio lida com toda a instalação necessária, para que você possa criar pontos de interrupção como de costume e depurar o que finalmente se torna processos independentes em execução em "servidores remotos", com o depurador já anexado, assim como esse.

Conforme mencionado anteriormente, sempre que você adiciona suporte à solução do Docker a um projeto de uma solução, esse projeto é configurado no arquivo global (nível da solução) `docker-compose.yml`, o que permite que você execute ou depure toda a solução de uma só vez. O Visual Studio iniciará um contêiner para cada projeto que tenha suporte habilitado à solução do Docker e executará todas as etapas internas para você (`dotnet publish`, `docker build`, etc.).

Se você quiser dar uma olhada na parte chata do trabalho, confira o arquivo:

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

O ponto importante aqui é que, como mostrado na Figura 5-12, no Visual Studio 2019, há um comando **Docker** adicional para a ação de tecla **F5**. Essa opção permite que você execute ou depure um aplicativo de vários contêineres executando todos os contêineres que estão definidos nos arquivos `docker-compose.yml` no nível da solução. A capacidade de depurar soluções de vários contêineres significa que você poderá definir vários pontos

de interrupção, cada ponto de interrupção em um projeto (contêiner) diferente e que, durante a depuração no Visual Studio, você vai parar nos pontos de interrupção definidos em diferentes projetos e em execução em diferentes contêineres.

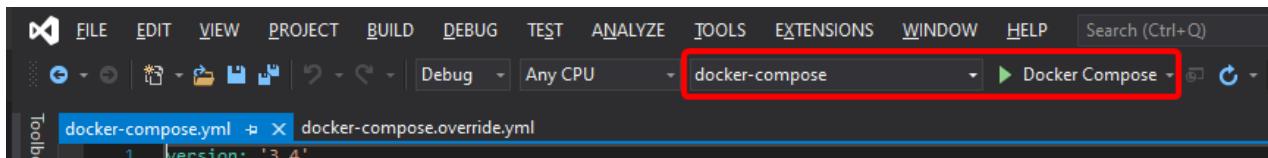


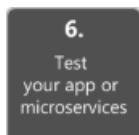
Figura 5-12. Executando aplicativos de vários contêineres no Visual Studio 2019

### Recursos adicionais

- Implantar um contêiner ASP.NET em um host remoto do Docker  
[/azure/vs-azure-tools-docker-hosting-web-apps-in-docker](#)

### Uma observação sobre teste e implantação com orquestradores

Os comandos docker-compose up e docker run (ou executar e depurar os contêineres no Visual Studio) são adequados para contêineres de teste em seu ambiente de desenvolvimento. Mas você não deve usar essa abordagem para implantações de produção, com orquestradores de destino como [Kubernetes](#) ou [Service Fabric](#). Se você estiver usando o Kubernetes, precisará usar [pods](#) para organizar contêineres e [Serviços](#) para agrupá-los. Você também pode usar [implantações](#) para organizar a criação e a modificação de pods.



## Etapa 6. Testar seu aplicativo do Docker usando o host local do Docker

Esta etapa varia de acordo com o que seu aplicativo está fazendo. Em um aplicativo Web .NET Core simples implantado como um único contêiner ou serviço, você pode acessar o serviço abrindo um navegador no host do Docker e navegando até o site, conforme mostrado na Figura 5-13. (Se a configuração do Dockerfile mapear o contêiner para uma porta no host que seja diferente de 80, inclua a porta do host na URL).



Figura 5-13. Exemplo de teste do seu aplicativo do Docker localmente usando localhost

Se o localhost não estiver apontando para o IP do host do Docker (por padrão, obrigatório ao usar o Docker CE), use o endereço IP da placa de rede do computador para navegar até o serviço.

Observe que a URL no navegador usa a porta 80 para o exemplo de contêiner específico que está sendo discutido. No entanto, internamente as solicitações estão sendo redirecionadas para a porta 5000, pois é assim que ele foi implantado com o comando docker run, conforme explicado em uma etapa anterior.

Você também pode testar o aplicativo usando o comando curl no terminal, conforme mostrado na Figura 5-14. Em uma instalação do Docker no Windows, o IP padrão do host do Docker é sempre 10.0.75.1, além do endereço IP real do computador.

```

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

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

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

```

**Figura 5-14.** Exemplo de teste do seu aplicativo do Docker localmente usando curl

### Testando e Depurando contêineres com o Visual Studio 2019

Ao executar e depurar os contêineres com o Visual Studio 2019, você pode depurar o aplicativo .NET da mesma forma como faria ao executar sem contêineres.

### Testando e depurando sem o Visual Studio

Se você estiver desenvolvendo usando a abordagem do editor/CLI, os contêineres de depuração serão mais difíceis e você provavelmente desejará depurar gerando rastreamentos.

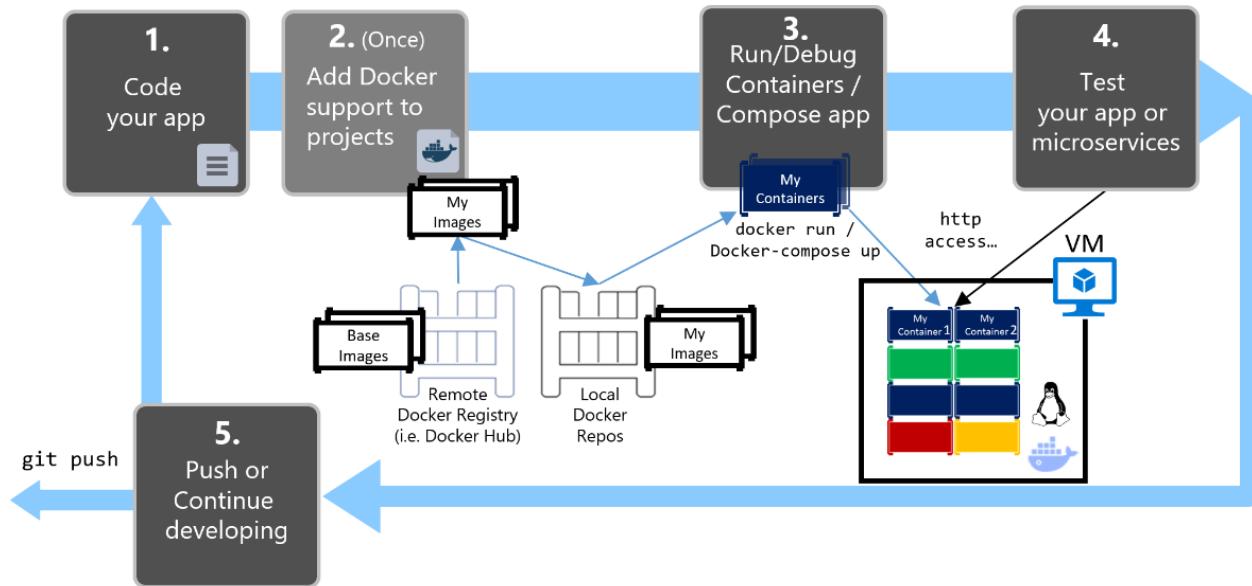
### Recursos adicionais

- **Depurando aplicativos em um contêiner do Docker local**  
<https://docs.microsoft.com/visualstudio/containers/edit-and-refresh>
- **Steve Lasker. Crie, depure, implante aplicativos ASP.NET Core com o Docker.** Vídeo.  
<https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T115>

## Fluxo de trabalho simplificado ao desenvolver contêineres com o Visual Studio

Efetivamente, o fluxo de trabalho ao usar o Visual Studio é muito mais simples do que ao usar a abordagem de editor/CLI. A maioria das etapas necessárias pelo Docker, relacionadas ao Dockerfile e aos arquivos docker-compose.yml, são ocultas ou simplificadas pelo Visual Studio, conforme mostrado na Figura 5-15.

# VS development workflow for Docker apps



O processo de desenvolvimento para aplicativos Docker: 1-codificar seu aplicativo, 2-gravar Dockerfile/s, 3-criar imagens definidas em Dockerfile/s, 4-(opcional) compor serviços no arquivo Docker-Compose. yml, 5-executar o contêiner ou o aplicativo Docker-Compose, 6-testar seu aplicativo ou seus microserviços, 7-enviar para o repositório e repetir

Figura 5-15. Fluxo de trabalho simplificado ao desenvolver com o Visual Studio

Além disso, será necessário executar a etapa 2 (adicionar suporte do Docker aos seus projetos) apenas uma vez. Portanto, o fluxo de trabalho é semelhante às suas tarefas de desenvolvimento regulares, quando utiliza o .NET para qualquer outro desenvolvimento. É necessário saber o que está acontecendo nos bastidores (o processo de build da imagem, quais imagens base você está usando, a implantação de contêineres, etc.) e, às vezes, você também precisará editar o Dockerfile ou o arquivo docker-compose.yml para personalizar comportamentos. Mas a maior parte do trabalho é bastante simplificada com o uso do Visual Studio, o que o torna muito mais produtivo.

## Recursos adicionais

- Steve Lasker. Desenvolvimento do Docker do .NET com o Visual Studio (2017)  
<https://channel9.msdn.com/Events/Visual-Studio/Visual-Studio-2017-Launch/T11>

## Usando comandos do PowerShell em um DockerFile para configurar contêineres do Windows

Os [contêineres do Windows](#) permitem converter seus aplicativos existentes do Windows em imagens do Docker e implantá-los com as mesmas ferramentas que o resto do ecossistema do Docker. Para usar contêineres do Windows, você executa comandos do PowerShell no Dockerfile, conforme mostrado no exemplo a seguir:

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

Nesse caso, estamos usando uma imagem base do Windows Server Core (a configuração FROM) e instalando IIS com um comando do PowerShell (a configuração RUN). Da mesma forma, você também pode usar comandos do PowerShell para configurar outros componentes, como ASP.NET 4.x, .NET 4.6 ou qualquer outro software do Windows. Por exemplo, o seguinte comando, em um Dockerfile, configura o ASP.NET 4.5:

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

## Recursos adicionais

- **aspnet-docker/Dockerfile.** Comandos do PowerShell de exemplo a serem executados em Dockerfiles para incluir recursos do Windows.  
<https://github.com/Microsoft/aspnet-docker/blob/master/4.7.1-windowsservercore-Itsc2016/runtime/Dockerfile>

[ANTERIOR](#)

[AVANÇAR](#)

# Projetando e desenvolvendo aplicativos .NET baseados em microsserviço e em vários contêineres

18/03/2020 • 2 minutes to read • [Edit Online](#)

*Desenvolver aplicações de microsserviço em contêiner significa que você está construindo aplicações de vários contêineres. No entanto, um aplicativo de vários contêineres também pode ser mais simples — por exemplo, um aplicativo de três níveis — e pode não ser construído usando uma arquitetura de microsserviço.*

Anteriormente, fizemos a pergunta "O Docker é necessário ao criar uma arquitetura de microsserviço?" A resposta é claramente não. O Docker é um habilitador e pode fornecer benefícios significativos, mas os contêineres e o Docker não são um requisito rígido para os microsserviços. Por exemplo, você pode criar um aplicativo baseado em microsserviços com ou sem o Docker ao usar o Azure Service Fabric, que dá suporte a microsserviços executados como processos simples ou como contêineres do Docker.

No entanto, se você souber como projetar e desenvolver um aplicativo baseado em microsserviços que também seja baseado em contêineres do Docker, será possível projetar e desenvolver qualquer outro modelo de aplicativo mais simples. Por exemplo, você pode projetar um aplicativo de três camadas que também exija uma abordagem de vários contêineres. Por esse motivo e como as arquiteturas de microsserviço são uma tendência importante no campo dos contêineres, esta seção concentra-se em uma implementação de arquitetura de microsserviço que usa contêineres do Docker.

[PRÓXIMO](#)

[ANTERIOR](#)

# Projete um aplicativo orientado a microserviços

18/03/2020 • 30 minutes to read • [Edit Online](#)

Esta seção se concentra no desenvolvimento de um aplicativo empresarial hipotético do lado do servidor.

## Especificações do aplicativo

O aplicativo hipotético lida com as solicitações por meio da execução de lógica de negócios, do acesso ao bancos de dados, retornando as respostas em HTML, JSON ou XML. Digamos que o aplicativo deve oferecer suporte a uma variedade de clientes, incluindo navegadores de área de trabalho que executam SPAs (aplicativos de página única), aplicativos Web tradicionais, aplicativos Web móveis e aplicativos móveis nativos. O aplicativo também deve expor uma API para ser consumida por terceiros. Ele também deve ser capaz de integrar seus microsserviços ou aplicativos externos de forma assíncrona, ajudando na resiliência dos microsserviços no caso de falhas parciais.

O aplicativo consistirá nesses tipos de componentes:

- Componentes de apresentação. Eles são responsáveis por gerenciar a interface do usuário e o consumo de serviços remotos.
- Lógica de negócios ou domínio. Essa é a lógica de domínio do aplicativo.
- Lógica de acesso a banco de dados. Consiste em componentes de acesso a dados responsáveis por acessar bancos de dados (SQL ou NoSQL).
- Lógica de integração do aplicativo. Isso inclui um canal de mensagens, principalmente com base em agentes de mensagens.

O aplicativo exigirá alta escalabilidade, permitindo que seus subsistemas verticais escalem horizontalmente de forma autônoma, pois alguns subsistemas exigirão maior escalabilidade que outros.

O aplicativo deverá ter a possibilidade de ser implantado em ambientes de várias infraestruturas (várias nuvens públicas e localmente) e, idealmente, ser multiplataforma, podendo ser mudado facilmente do Linux para o Windows (ou vice-versa).

## Contexto da equipe de desenvolvimento

Pressupõe-se também o seguinte sobre o processo de desenvolvimento do aplicativo:

- Você tem várias equipes de desenvolvimento com foco em diferentes áreas de negócios do aplicativo.
- Os novos membros da equipe devem se tornar produtivos rapidamente, e o aplicativo deve ser fácil de entender e modificar.
- O aplicativo terá uma evolução de longo prazo e regras de negócio em constante mudança.
- Você precisará de facilidade de manutenção a longo prazo, o significa a agilidade na implementação de novas alterações no futuro possibilitando a atualização de vários subsistemas com impacto mínimo nos outros subsistemas.
- Você deseja a prática de integração contínua e implantação contínua do aplicativo.
- Você deseja aproveitar as tecnologias emergentes (estruturas, linguagens de programação, etc.) durante a evolução do aplicativo. Você não quer fazer migrações completas do aplicativo ao mudar para novas tecnologias, pois isso resultaria em custos altos e afetaria a previsibilidade e a estabilidade do aplicativo.

# Escolha de uma arquitetura

Qual deve ser a arquitetura de implantação do aplicativo? As especificações do aplicativo, junto com o contexto de desenvolvimento, sugerem enfaticamente que você deve projetar o aplicativo decompondo-o em subsistemas autônomos, na forma de microsserviços e contêineres de colaboração, em que um microsserviço é um contêiner.

Nessa abordagem, cada serviço (contêiner) implementa um conjunto de funções coesas e estreitamente relacionadas. Por exemplo, um aplicativo pode consistir em serviços como: serviço de catálogo, serviço de pedidos, serviço de carrinho de compras, serviço de perfil do usuário, etc.

Os microsserviços se comunicam usando protocolos como HTTP (REST), mas também de forma assíncrona (usando AMQP, por exemplo) sempre que possível, especialmente ao propagar atualizações com eventos de integração.

Os microsserviços são desenvolvidos e implantados como contêineres independentes uns dos outros. Isso significa que uma equipe de desenvolvimento pode desenvolver e implantar um determinado microsserviço sem afetar outros subsistemas.

Cada microsserviço tem seu próprio banco de dados, permitindo que ele seja totalmente separado dos outros microsserviços. Quando necessário, a consistência entre bancos de dados de diferentes microsserviços é obtida com o uso de eventos de integração no nível do aplicativo (por meio de um barramento de eventos lógico), conforme manipulados na CQRS (segregação de responsabilidade de comando e consulta). Por isso, as restrições de negócios devem adotar consistência eventual entre os vários microsserviços e bancos de dados relacionados.

## eShopOnContainers: um aplicativo de referência para .NET Core e microsserviços implantados com o uso de contêineres

Para que você possa se concentrar na arquitetura e nas tecnologias, em vez de pensar em um domínio corporativo hipotético desconhecido, selecionamos um domínio corporativo bem conhecido, ou seja, um aplicativo simplificado de comércio eletrônico (loja eletrônica) que apresenta um catálogo de produtos, recebe pedidos de clientes, verifica o estoque e executa outras funções de negócios. O código-fonte desse aplicativo baseado em contêineres está disponível no repositório [eShopOnContainers](#) do GitHub.

O aplicativo consiste em vários subsistemas, incluindo vários front-ends de interface do usuário da loja (um aplicativo Web e um aplicativo móvel nativo), juntamente com os microsserviços e contêineres de back-end para todas as operações necessárias do lado do servidor com vários Gateways de API como pontos de entrada consolidados para os microsserviços internos. A figura 6-1 mostra a arquitetura do aplicativo de referência.

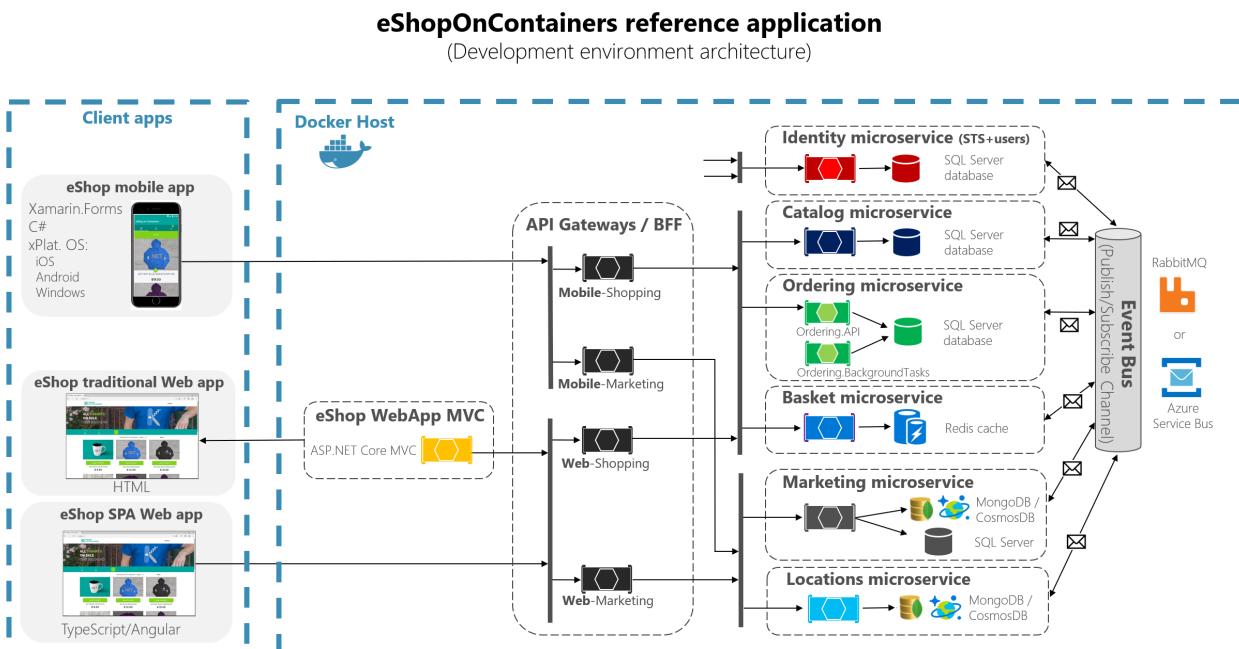


Figura 6-1. A arquitetura do aplicativo de referência eShopOnContainers para o ambiente de desenvolvimento

O diagrama acima mostra que os clientes Mobile e SPA se comunicam com pontos finais de gateway de API único, que então se comunicam com microsserviços. Os clientes web tradicionais comunicam-se ao microserviço MVC, que se comunica com microsserviços através do gateway api.

**Ambiente de hospedagem.** Na Figura 6-1, você vê vários contêineres implantados em um único host do Docker. Esse seria o caso ao implantar em um único host do Docker com o comando docker-compose up. No entanto, se você estiver usar um orquestrador ou cluster de contêineres, cada contêiner poderá ser executado em um host (nó) diferente, e qualquer nó poderá ser executado em qualquer número de contêineres, como explicado anteriormente na seção sobre arquitetura.

**Arquitetura de comunicação.** O aplicativo eShopOnContainers usa dois tipos de comunicação, dependendo do tipo de ação funcional (consultas versus atualizações e transações):

- Comunicação entre cliente HTTP e microsserviço por meio de gateways de API. Isso é usado para consultas e ao aceitar comandos transacionais ou de atualização dos aplicativos cliente. A abordagem usando gateways de API é explicada em detalhes nas seções posteriores.
- Comunicação baseada em evento assíncrono. Isso ocorre por meio de um barramento de eventos para propagar atualizações em microsserviços ou integrar com aplicativos externos. O barramento de eventos pode ser implementado com qualquer tecnologia de infraestrutura de agente de mensagens, como RabbitMQ, ou usando barramentos de serviços de nível mais alto (nível de abstração), como Barramento de Serviço do Azure, NServiceBus, MassTransit ou Brighter.

O aplicativo é implantado como um conjunto de microsserviços na forma de contêineres. Aplicativos cliente podem se comunicar com esses microsserviços em execução como contêineres por meio de URLs públicas publicadas pelos gateways de API.

### **Soberania de dados por microsserviço**

No aplicativo de exemplo, cada microsserviço tem seu próprio banco de dados ou fonte de dados, embora todos os bancos de dados do SQL Server sejam implantados em um único contêiner. Essa decisão de design foi tomada apenas para facilitar para um desenvolvedor ao obter o código do GitHub, cloná-lo e abri-lo no Visual Studio ou o Visual Studio Code. Ou, alternativamente, torna fácil compilar as imagens docker personalizadas usando o .NET Core CLI e o Cli Docker e, em seguida, implantá-las e executá-las em um ambiente de desenvolvimento Docker. De uma forma ou de outra, o uso de contêineres para fontes de dados permite que os desenvolvedores criem e implantem em questão de minutos, sem a necessidade de provisionar um banco de dados externo ou qualquer outra fonte de dados com dependências rígidas na infraestrutura (de nuvem ou local).

Em um ambiente de produção real, por questões de alta disponibilidade e escalabilidade, os bancos de dados devem ser baseados em servidores de banco de dados na nuvem ou locais, mas não em contêineres.

Portanto, as unidades de implantação para os microsserviços (e até mesmo para bancos de dados deste aplicativo) são contêineres do Docker e o aplicativo de referência é um aplicativo de vários contêineres que adota princípios de microsserviços.

### **Recursos adicionais**

- EShopOnContainers GitHub repo. Código fonte para o aplicativo de referência  
<https://aka.ms/eShopOnContainers/>

## **Benefícios de uma solução baseada em microsserviços**

Uma solução baseada em microsserviços como esta tem muitos benefícios:

**Cada microsserviço é relativamente pequeno, fácil de gerenciar e desenvolver.** Especificamente:

- É fácil para um desenvolvedor entender e se familiarizar rapidamente com boa produtividade.
- Os contêineres são rapidamente iniciados, tornando os desenvolvedores mais produtivos.

- Um IDE como Visual Studio pode carregar projetos menores rapidamente, fazendo com que os desenvolvedores sejam mais produtivos.
- Cada microsserviço pode ser criado, desenvolvido e implantado independentemente de outros microsserviços. Isso proporciona agilidade, porque é mais fácil implantar novas versões dos microsserviços com frequência.

**É possível escalar horizontalmente áreas individuais do aplicativo.** Por exemplo, suponha que o serviço de catálogo ou do carrinho de compras tenha que ser expandido, mas não o processo de pedidos. Uma infraestrutura de microsserviços será muito mais eficiente do que uma arquitetura monolítica em relação aos recursos usados ao expandir.

**Você pode dividir o trabalho de desenvolvimento entre várias equipes.** Cada serviço pode ser de propriedade de única uma equipe de desenvolvimento. Cada equipe pode gerenciar, desenvolver, implantar e dimensionar o serviço de maneira independente das outras equipes.

**Os problemas são mais isolados.** Se houver um problema em um serviço, somente esse serviço será inicialmente afetado (exceto quando um design incorreto for usado, com dependências diretas entre microsserviços) e os outros serviços poderão continuar lidando com as solicitações. Por outro lado, um componente com defeito em uma arquitetura de implantação monolítica poderá derrubar todo o sistema, especialmente quando envolver recursos, como uma perda de memória. Além disso, quando um problema em um microsserviço for resolvido, você poderá implantar apenas o microsserviço afetado sem afetar o restante do aplicativo.

**Você pode usar as tecnologias mais recentes.** Com a possibilidade de começar a desenvolver serviços de forma independente e executá-los lado a lado (graças aos contêineres e ao .NET Core), você pode usar as últimas tecnologias e estruturas de acordo com a conveniência, em vez de ficar preso a uma pilha ou estrutura mais antiga para todo o aplicativo.

## Desvantagens de uma solução baseada em microsserviços

Uma solução baseada em microserviços como esta também tem algumas desvantagens:

**Aplicação distribuída.** A distribuição do aplicativo cria complexidades para os desenvolvedores ao projetar e criar os serviços. Por exemplo, os desenvolvedores devem implementar a comunicação inter-serviço usando protocolos como HTTP ou AMPQ, o que adiciona complexidade para testes e tratamento de exceções. Isso também adiciona latência ao sistema.

**Complexidade de implantação.** Um aplicativo com vários tipos de microsserviços e que necessite de alta escalabilidade (ele precisa ser capaz de criar várias instâncias por serviço e equilibrar os serviços em vários hosts) gera um alto grau de complexidade de implantação para o gerenciamento e as operações de TI. Se você não estiver usando uma infraestrutura orientada a microsserviços (como um agendador e orquestrador), essa complexidade adicional poderá exigir esforços de desenvolvimento muito maiores que o próprio aplicativo de negócios.

**Transações atômicas.** Geralmente, as transações atômicas entre vários microsserviços não são possíveis. Os requisitos corporativos precisam adotar a consistência eventual entre vários microsserviços.

**Maiores necessidades de recursos globais** (memória total, unidades e recursos de rede para todos os servidores ou hosts). Em muitos casos, ao substituir um aplicativo monolítico por uma abordagem de microsserviços, a quantidade de recursos globais iniciais necessários para o novo aplicativo baseado em microsserviços será maior do que as necessidades de infraestrutura do aplicativo monolítico original. Isso ocorre porque o maior grau de granularidade e serviços distribuídos exige mais recursos globais. No entanto, considerando o baixo custo de recursos em geral e o benefício de expandir apenas determinadas áreas do aplicativo, comparados aos custos de longo prazo relacionados ao desenvolvimento de aplicativos monolíticos, o aumento no uso de recursos geralmente compensa nas grandes aplicações de longo prazo.

**Problemas com a comunicação direta entre cliente e microsserviço.** Se o aplicativo for grande, com dezenas de microsserviços, haverá desafios e limitações se houver necessidade de comunicações diretas entre o cliente e o microsserviço. Um problema é uma potencial incompatibilidade entre as necessidades do cliente e as APIs expostas por cada um dos microsserviços. Em alguns casos, o aplicativo cliente precisará fazer muitas solicitações separadas para compor a interface do usuário, tornando-se ineficiente na Internet e impraticável em uma rede móvel. Portanto, as solicitações do aplicativo cliente ao sistema de back-end devem ser minimizadas.

Outro problema com a comunicação direta entre cliente e microsserviço é que alguns microsserviços podem usar protocolos que não sejam compatíveis com a Web. Um serviço pode usar um protocolo binário, enquanto outro pode usar o sistema de mensagens AMQP. Esses protocolos não são compatíveis com firewalls e são mais bem usados internamente. Normalmente, um aplicativo deve usar protocolos como HTTP e WebSockets para a comunicação do lado de fora do firewall.

Outra desvantagem dessa abordagem direta entre cliente e serviço é que ela dificulta a refatoração dos contratos desses microsserviços. Ao longo do tempo, talvez os desenvolvedores queiram alterar a forma como o sistema está partitionado em serviços. Por exemplo, eles podem mesclar dois serviços ou dividir um serviço em dois ou mais serviços. No entanto, se os clientes se comunicarem diretamente com os serviços, esse tipo de refatoração poderá interromper a compatibilidade com aplicativos cliente.

Conforme mencionado na seção de arquitetura, ao projetar e criar um aplicativo complexo baseado em microsserviços, considere o uso de vários Gateways de API refinados em vez da abordagem mais simples de comunicação direta entre cliente e microsserviço.

**Particionamento dos microsserviços.** Por fim, independentemente da abordagem escolhida para sua arquitetura de microsserviços, outro desafio é decidir como partitionar um aplicativo de ponta a ponta em vários microsserviços. Conforme observado na seção de arquitetura do guia, há várias técnicas e abordagens que você pode escolher. Basicamente, você precisa identificar áreas do aplicativo que sejam separados de outras áreas e que tenham um número baixo de dependências rígidas. Em muitos casos, isso se alinha ao partitionamento de serviços de acordo com o caso de uso. Por exemplo, em nosso aplicativo de loja eletrônica, temos um serviço de pedidos que é responsável por toda a lógica de negócios relacionada ao processo de pedido. Também temos o serviço de catálogo e o serviço de carrinho de compras, que implementam outros recursos. Idealmente, cada serviço deveria ter somente um pequeno conjunto de responsabilidades. Isso é semelhante ao princípio SRP (princípio de responsabilidade única) aplicado a classes, que declara que uma classe deve ter somente uma razão para ser alterada. Mas, nesse caso, estamos lidando com microsserviços, portanto o escopo será maior que o de uma única classe. Acima de tudo, um microsserviço deve ser completamente autônomo, de ponta a ponta, incluindo a responsabilidade por suas próprias fontes de dados.

## Padrões de arquitetura e design externos versus internos

A arquitetura externa é a arquitetura de microsserviço composta por vários serviços, de acordo com os princípios descritos na seção de arquitetura deste guia. No entanto, dependendo da natureza de cada microsserviço e, independentemente da arquitetura de microsserviço de alto nível que você escolhe, é comum e muitas vezes aconselhável, ter arquiteturas internas distintas para os diferentes microsserviços, cada qual baseada em padrões diferentes. Os microsserviços podem até usar tecnologias e linguagens de programação diferentes. A figura 6-2 ilustra essa diversidade.

## External architecture per application

## Internal architecture per microservice

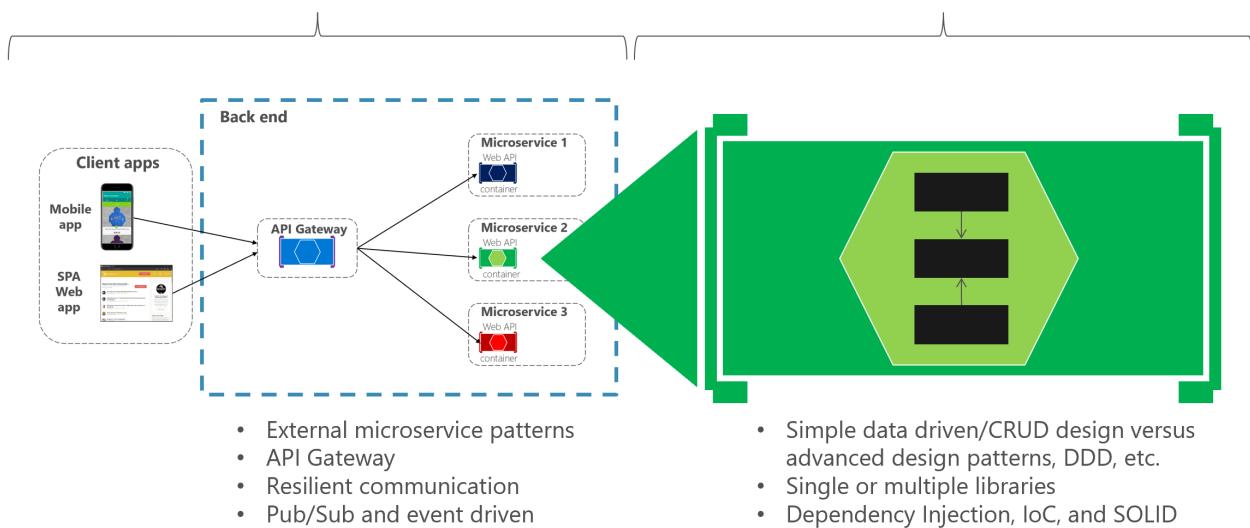


Figura 6-2. Arquitetura e design externos versus internos

Por exemplo, em nosso *eShopOnContainers* de exemplo, os microsserviços de catálogo, carrinho de compras e de perfil do usuário são simples (basicamente, subsistemas CRUD). Portanto, a arquitetura interna e o design deles são bastante simples. No entanto, você pode ter outros microsserviços, como o microsserviço de pedidos, que é mais complexo e representa regras de negócios em constante mudança, com um alto grau de complexidade de domínio. Nesses casos, pode ser interessante implementar padrões mais avançados em um microsserviço específico, como aqueles definidos nas abordagens de DDD (design controlado por domínio), como estamos fazendo no microsserviço de pedidos do *eShopOnContainers*. (Examinaremos esses padrões de DDD em uma seção posterior que explica a implementação do microsserviço de pedidos do *eShopOnContainers*).

Outro motivo para uma tecnologia diferente para cada microsserviço seria a natureza de cada microsserviço. Por exemplo, talvez seja melhor usar uma linguagem de programação funcional, como F#, ou uma linguagem como a R, se você estiver destinando a domínios de aprendizado de máquina e inteligência artificial, em vez de uma linguagem de programação mais orientada a objeto, como a C#.

O resultado é que cada microsserviço pode ter uma arquitetura interna diferente com base nos diferentes padrões de design. Nem todos os microsserviços devem ser implementados usando padrões avançados de DDD, porque que isso seria excesso de engenharia. Da mesma forma, microsserviços complexos, com lógica de negócios em constante mudança, não devem ser implementados como componentes de CRUD, ou você poderá ficar com um código de baixa qualidade.

## O novo mundo: vários padrões de arquitetura e microsserviços poliglotas

Há muitos padrões de arquitetura usados por desenvolvedores e arquitetos de software. Veja alguns a seguir (combinando estilos de arquitetura e padrões de arquitetura):

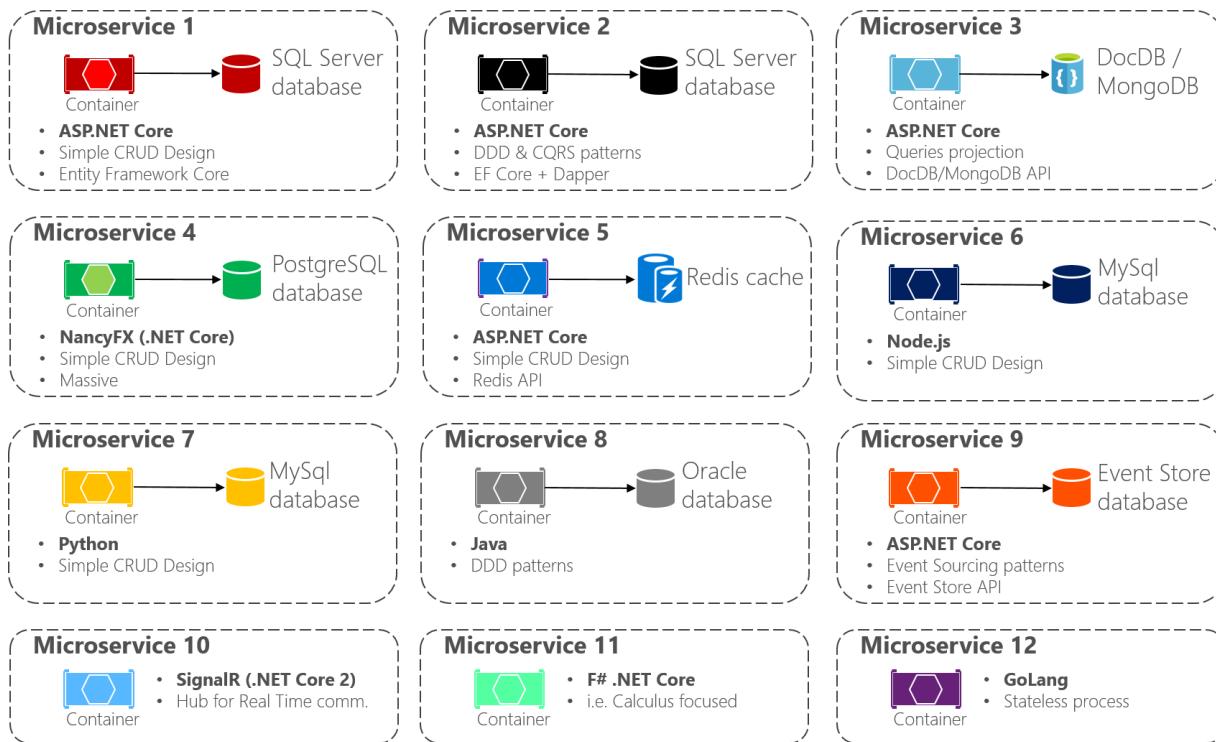
- CRUD simples, de camada e nível únicos.
- [Tradicional em N camadas](#).
- [Design controlado po domínio em N camadas](#).
- [Arquitetura limpa](#) (com a usada no *eShopOnWeb*)
- [Segregação de Responsabilidade de Comando e Consulta](#) (CQRS).

- **EDA** (Arquitetura controlada por eventos).

Você também pode criar microserviços com várias tecnologias e linguagens, como APIs Web do ASP.NET Core, NancyFx, SignalR do ASP.NET Core (disponível no .NET Core 2), F#, Node.js, Python, Java, C++, GoLang e muito mais.

O ponto importante é que não há um padrão de arquitetura ou estilo específico nem qualquer tecnologia em particular que seja ideal para todas as situações. A figura 6-3 mostra algumas abordagens e tecnologias (embora não estejam em nenhuma ordem específica) que podem ser usadas em microserviços diferentes.

## The Multi-Architectural-Patterns and polyglot microservices world



**Figura 6-3.** Padrões de várias arquitetura e o mundo de microserviços poliglotas

Os microserviços poliglotas e de padrão de várias arquiteturas significam que você pode combinar linguagens e tecnologias com as necessidades de cada microserviço e ainda fazer com que eles conversem entre si. Conforme mostrado na Figura 6-3, em aplicativos compostos de muitos microserviços (de contextos delimitados, na terminologia de design controlado por domínio ou simplesmente "subsistemas", como microserviços autônomos), você pode implementar cada microserviço de maneira diferente. Cada um pode ter um padrão de arquitetura diferente e usar linguagens e bancos de dados diferentes, dependendo da natureza, dos requisitos empresariais e das prioridades do aplicativo. Em alguns casos, os microserviços podem ser semelhantes. Mas, geralmente, esse não é o caso, porque o limite de contexto e os requisitos de cada subsistema costumam ser diferentes.

Por exemplo, em um aplicativo CRUD simples de manutenção, não faz sentido projetar e implementar padrões de DDD. Mas, para seu negócio principal ou domínio principal, é interessante aplicar padrões mais avançados para lidar com a complexidade dos negócios e com as regras de negócio em constante mudança.

Especialmente quando você lida com grandes aplicativos compostos por vários subsistemas, você não deve aplicar uma única arquitetura de nível superior baseada em um único padrão de arquitetura. Por exemplo, a CQRS não deve ser aplicada como uma arquitetura de alto nível para um aplicativo inteiro, mas pode ser útil para um conjunto específico de serviços.

Não há solução definitiva nem um padrão de arquitetura correto para cada caso específico. Você não pode ter "um padrão de arquitetura para controlar tudo". Dependendo das prioridades de cada microserviço, você deverá escolher uma abordagem diferente para cada um deles, conforme explicado nas seções a seguir.

[PRÓXIMO](#)

[ANTERIOR](#)

# Criando um microsserviço de CRUD simples controlado por dados

10/09/2020 • 33 minutes to read • [Edit Online](#)

Esta seção descreve como criar um microsserviço simples que execute operações de CRUD (criar, ler, atualizar e excluir) em uma fonte de dados.

## Criando um microsserviço de CRUD simples

Do ponto de vista do design, esse tipo de microsserviço em contêineres é muito simples. Talvez o problema a ser resolvido seja simples ou talvez a implementação seja apenas uma prova de conceito.

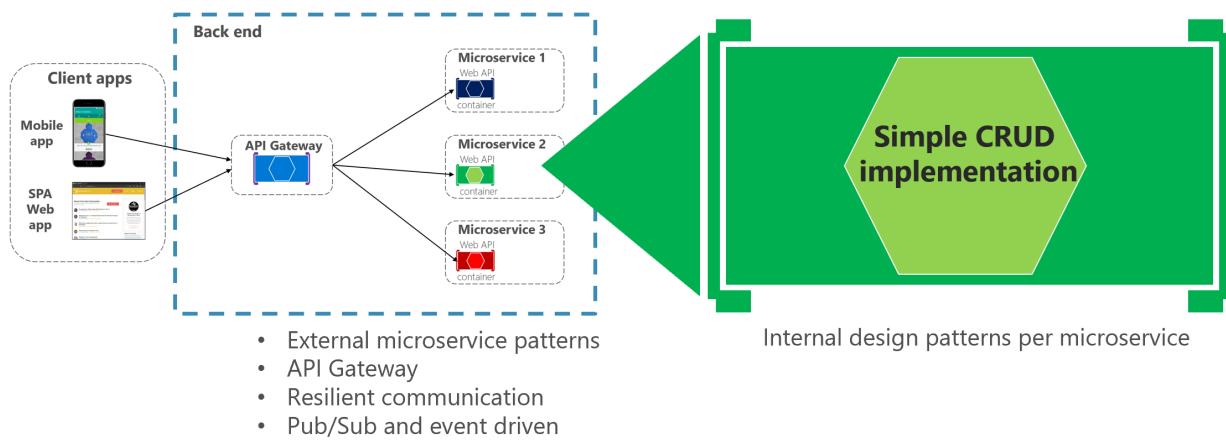


Figura 6-4. Design interno para microsserviços de CRUD simples

Um exemplo desse tipo de serviço de unidade de dados é o microsserviço de catálogo do aplicativo de exemplo eShopOnContainers. Esse tipo de serviço implementa todas as suas funcionalidades em um único projeto da API Web ASP.NET Core que inclui classes para seu modelo de dados, sua lógica de negócios e seu código de acesso a dados. Ele também armazena os dados relacionados em um banco de dados em execução no SQL Server (como outro contêiner para fins de Desenvolvimento/Teste), mas também pode ser qualquer host normal do SQL Server, conforme mostrado na Figura 6-5.

# Data-Driven/CRUD microservice container

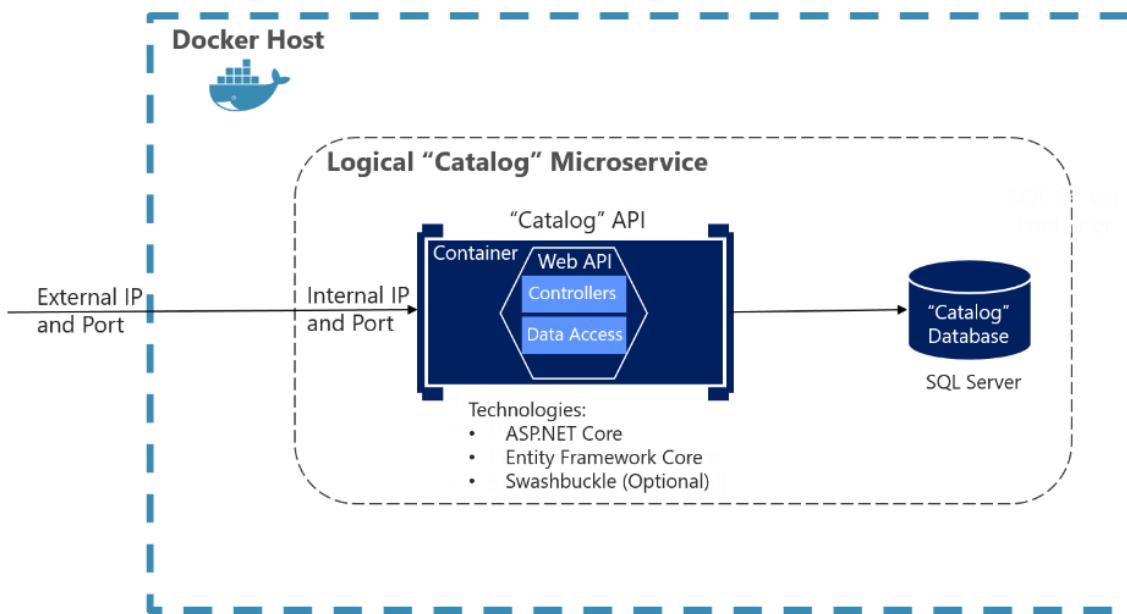


Figura 6-5. Design de microsserviço controlado por dados/CRUD simples

O diagrama anterior mostra o microserviço de catálogo lógico, que inclui seu banco de dados de catálogo, que pode estar ou não no mesmo host do Docker. Ter o banco de dados no mesmo host do Docker pode ser bom para desenvolvimento, mas não para produção. Quando estiver desenvolvendo esse tipo de serviço, você somente precisará do [ASP.NET Core](#) e de uma API de acesso a dados ou de um ORM (mapeador relacional de objeto) como o [Entity Framework Core](#). Você também pode gerar metadados do [Swagger](#) automaticamente por meio do [Swashbuckle](#) para fornecer uma descrição do que o serviço oferece, conforme será explicado na próxima seção.

Observe que executar um servidor de banco de dados como o SQL Server em um contêiner do Docker é ótimo para ambientes de desenvolvimento, porque todas as dependências podem funcionar sem precisar provisionar um banco de dados na nuvem ou localmente. Isso é muito conveniente ao executar testes de integração. No entanto, para ambientes de produção, executar um servidor de banco de dados em um contêiner não é recomendável, porque, geralmente, essa abordagem não oferece alta disponibilidade. Para um ambiente de produção no Azure, é recomendável usar o BD SQL do Azure ou qualquer outra tecnologia de banco de dados que possa fornecer alta disponibilidade e alta escalabilidade. Por exemplo, para uma abordagem NoSQL, você pode escolher o CosmosDB.

Por fim, editando os arquivos de metadados Dockerfile e docker-compose.yml, você pode configurar como a imagem desse contêiner será criada, ou seja, qual imagem base ele usará, além das configurações de design, como nomes internos e externos e portas TCP.

## Implementando um microsserviço de CRUD simples com o ASP.NET Core

Para implementar um microsserviço CRUD simples usando o .NET Core e o Visual Studio, comece criando um projeto simples de API Web do ASP.NET Core (em execução no .NET Core, para que ele possa ser executado em um host Linux do Docker), como é mostrado na Figura 6-6.

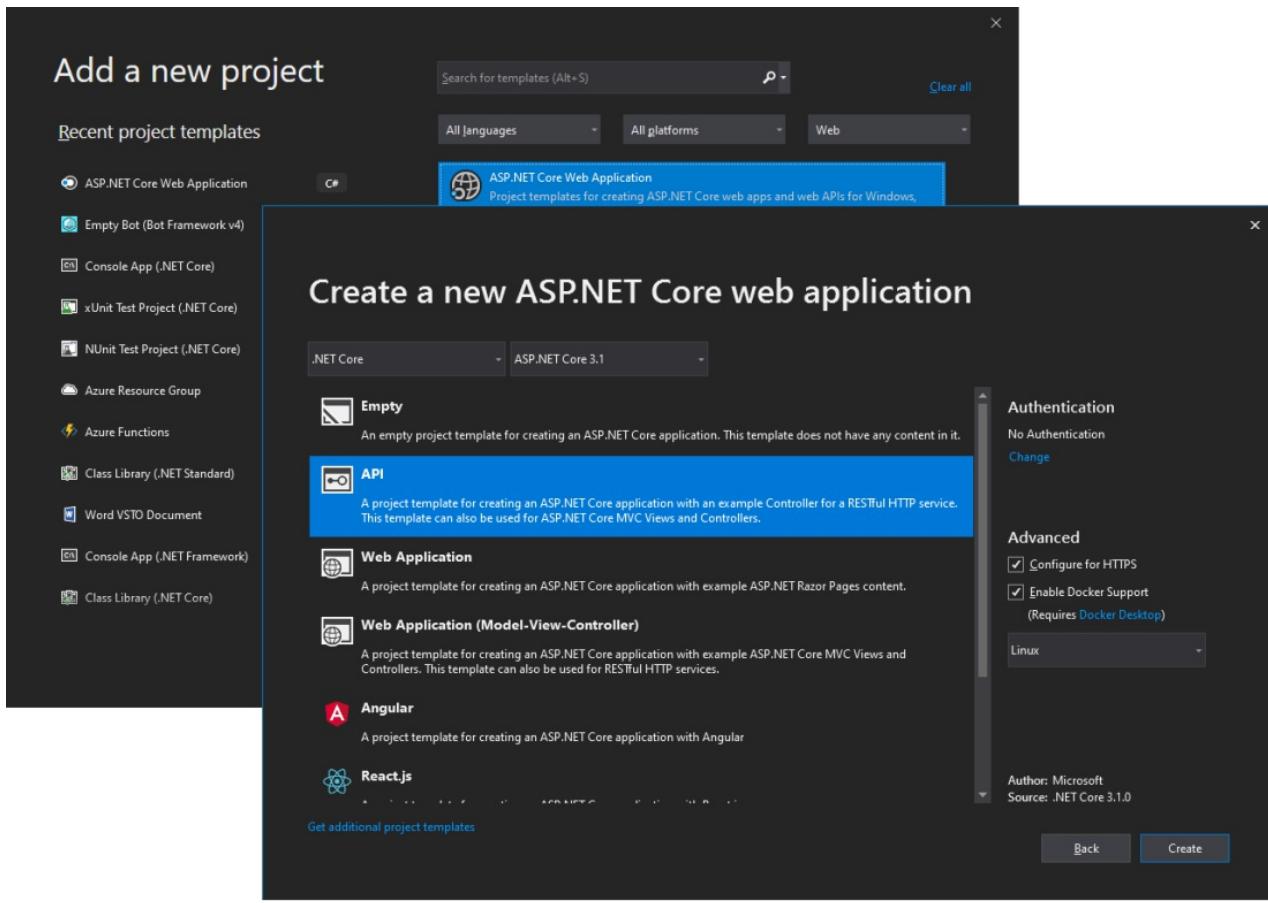


Figura 6-6. Criando um projeto de API Web ASP.NET Core no Visual Studio 2019

Para criar um Projeto de API Web do ASP.NET Core, primeiro selecione um Aplicativo Web do ASP.NET Core e, em seguida, selecione o tipo de API. Depois de criar o projeto, você poderá implementar os controladores de MVC como faria em qualquer outro projeto de API Web, usando a API do Entity Framework ou uma outra API. Em um novo projeto de API Web, você verá que a única dependência existente nesse microsserviço é em relação ao próprio ASP.NET Core. Internamente, dentro da dependência *Microsoft.AspNetCore.All*, ele faz referência a Entity Framework e a muitos outros pacotes NuGet do .NET Core, como mostra a Figura 6-7.

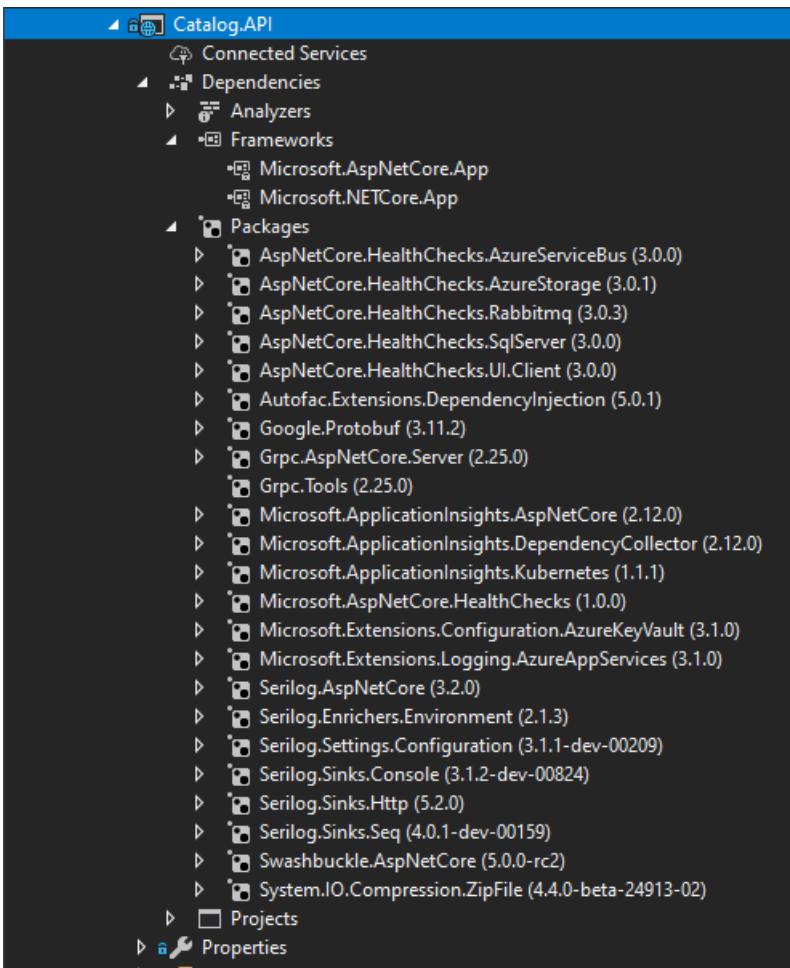


Figura 6-7. Dependências em um microsserviço de API Web de CRUD simples

O projeto de API inclui referências ao pacote NuGet Microsoft.AspNetCore.App, que inclui referências a todos os pacotes essenciais. Ele pode incluir alguns outros pacotes também.

### Implementando serviços da API Web de CRUD com o Entity Framework Core

O Entity Framework (EF) Core é uma versão de multiplataforma leve, extensível e de plataforma cruzada da popular tecnologia de acesso a dados do Entity Framework. O EF Core é um ORM (mapeador relacional de objeto) que permite que os desenvolvedores do .NET trabalhem com um banco de dados usando objetos .NET.

O microsserviço de catálogo usa o EF e o provedor do SQL Server porque seu banco de dados está em execução em um contêiner com o SQL Server para a imagem do Docker do Linux. No entanto, o banco de dados poderia ser implantado em qualquer SQL Server, como o Windows local ou o BD SQL do Azure. A única coisa que você precisaria alterar seria a cadeia de conexão no microsserviço do ASP.NET Web API.

#### O modelo de dados

Com o EF Core, o acesso a dados é executado usando um modelo. Um modelo é composto por classes de entidade (modelo de domínio) e um contexto derivado (DbContext) que representa uma sessão com o banco de dados, permitindo que você consulte e salve os dados. Você pode gerar um modelo a partir de um banco de dados existente, codificar manualmente um modelo para corresponder ao banco de dados ou usar a técnica de migrações do EF para criar um banco de dados do seu modelo, usando a abordagem Code-First (que facilita a evolução do banco de dados à medida que seu modelo muda ao longo do tempo). Para o microsserviço de catálogo, a última abordagem foi usada. Veja um exemplo da classe de entidade CatalogItem no exemplo de código a seguir, que é uma classe de entidade POCO (objeto CRL básico) simples.

```

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

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

    // Additional code ...
}

```

Você também precisará de um `DbContext` que represente uma sessão com o banco de dados. Para o microsserviço de catálogo, a classe `CatalogContext` é derivada da classe base `DbContext`, conforme é mostrado no exemplo a seguir:

```

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

    // Additional code ...
}

```

É possível ter implementações adicionais do `DbContext`. Por exemplo, no microsserviço `Catalog.API` de exemplo, há um segundo `DbContext` chamado `CatalogContextSeed` no qual ele populará automaticamente os dados de exemplo na primeira vez que tentar acessar o banco de dados. Esse método é útil para dados de demonstração e também para cenários de teste automatizados.

Dentro do `DbContext`, use o método `OnModelCreating` para personalizar os mapeamentos de entidade de banco de dados/objeto e outros [pontos de extensibilidade do EF](#).

#### Consultando dados de controladores da API Web

As instâncias das classes de entidade normalmente são recuperadas do banco de dados usando LINQ (consulta integrada à linguagem), conforme é mostrado no exemplo a seguir:

```

[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _catalogContext;
    private readonly CatalogSettings _settings;
    private readonly ICatalogIntegrationEventService _catalogIntegrationEventService;

    public CatalogController(
        CatalogContext context,
        IOptionsSnapshot<CatalogSettings> settings,
        ICatalogIntegrationEventService catalogIntegrationEventService)
    {
        _catalogContext = context ?? throw new ArgumentNullException(nameof(context));
        _catalogIntegrationEventService = catalogIntegrationEventService
            ?? throw new ArgumentNullException(nameof(catalogIntegrationEventService));

        _settings = settings.Value;
        context.ChangeTracker.QueryTrackingBehavior = QueryTrackingBehavior.NoTracking;
    }

    // GET api/v1/[controller]/items[?pageSize=3&pageIndex=10]
    [HttpGet]
    [Route("items")]
    [ProducesResponseType(typeof(PaginatedItemsViewModel<CatalogItem>), (int) HttpStatusCode.OK)]
    [ProducesResponseType(typeof(IEnumerable<CatalogItem>), (int) HttpStatusCode.OK)]
    [ProducesResponseType((int) HttpStatusCode.BadRequest)]
    public async Task<IActionResult> ItemsAsync()
    {
        [FromQuery] int pageSize = 10,
        [FromQuery] int pageIndex = 0,
        string ids = null
    {
        if (!string.IsNullOrEmpty(ids))
        {
            var items = await GetItemsByIdsAsync(ids);

            if (!items.Any())
            {
                return BadRequest("ids value invalid. Must be comma-separated list of numbers");
            }

            return Ok(items);
        }

        var totalItems = await _catalogContext.CatalogItems
            .LongCountAsync();

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

        itemsOnPage = ChangeUriPlaceholder(itemsOnPage);

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

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

```

#### Salvando dados

Dados são criados, excluídos e modificados no banco de dados usando as instâncias de suas classes de entidade. Você poderia adicionar um código como o exemplo embutido em código a seguir (dados fictícios, neste caso) aos controladores da API Web.

```

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

```

#### Injeção de dependência nos controladores da API Web e do ASP.NET Core

Na ASP.NET Core, você pode usar a DI (injeção de dependência) pronta para uso. Não é necessário configurar um contêiner de IoC (inversão de controle) de terceiros, embora seja possível conectar o contêiner de IoC de sua preferência à infraestrutura do ASP.NET Core, caso deseje. Nesse caso, isso significa que você pode injetar diretamente o DbContext do EF necessário ou repositórios adicionais por meio do construtor do controlador.

Na `CatalogController` classe mencionada anteriormente, `CatalogContext` (que herda de `DbContext`) tipo é injetado junto com os outros objetos necessários no `CatalogController()` Construtor.

Uma configuração importante a ser definida no projeto de API Web é o registro da classe DbContext no contêiner de IoC do serviço. Normalmente, você faz isso na `Startup` classe chamando o `services.AddDbContext<CatalogContext>()` método dentro do `ConfigureServices()` método, conforme mostrado no exemplo **simplificado** a seguir:

```

public void ConfigureServices(IServiceCollection services)
{
    // Additional code...
    services.AddDbContext<CatalogContext>(options =>
    {
        options.UseSqlServer(Configuration["ConnectionString"],
        sqlServerOptionsAction: sqlOptions =>
        {
            sqlOptions.MigrationsAssembly(
                typeof(Startup).GetTypeInfo().Assembly.GetName().Name);

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

```

#### Recursos adicionais

- Consultando dados

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

- Salvando dados

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

## A cadeia de conexão e as variáveis de ambiente do BD usadas pelos contêineres do Docker

É possível usar as configurações do ASP.NET Core e adicionar uma propriedade `ConnectionString` no arquivo `settings.json`, conforme é mostrado no exemplo a seguir:

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

O arquivo settings.json pode ter valores padrão para a propriedade ConnectionString ou para qualquer outra propriedade. No entanto, essas propriedades serão substituídas pelos valores das variáveis de ambiente que você especificar no arquivo docker-compose.override.yml, ao usar o Docker.

Usando os arquivos docker-compose.yml ou docker-compose.override.yml, é possível inicializar essas variáveis de ambiente para que o Docker as configure como variáveis de ambiente do sistema operacional, como é mostrado no arquivo docker-compose.override.yml a seguir (a cadeia de conexão e as outras linhas são quebradas automaticamente neste exemplo, mas não são quebradas automaticamente em seu próprio arquivo).

```
# docker-compose.override.yml

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

Os arquivos docker-compose.yml no nível da solução não são apenas mais flexíveis do que arquivos de configuração no nível do projeto ou do microserviço, mas também serão mais seguros se você substituir as variáveis de ambiente declaradas em arquivos docker-compose com valores definidos das suas ferramentas de implantação, como de tarefas de implantação do Docker do Azure DevOps Services.

Por fim, você pode obter esse valor do código usando Configuration["ConnectionString"], como foi mostrado no método ConfigureServices em um exemplo de código anterior.

No entanto, para ambientes de produção, é recomendável explorar outras maneiras de armazenar segredos como as cadeias de conexão. Uma excelente maneira de gerenciar segredos do aplicativo é usar [Azure Key Vault](#).

O Azure Key Vault ajuda a armazenar e proteger as chaves de criptografia e os segredos usados por aplicativos e serviços de nuvem. Um segredo é qualquer coisa sobre a qual você deseja manter controle estrito, como chaves de API, cadeias de conexão, senhas, etc., e o controle estrito inclui log de uso, definição de expiração, gerenciamento de acesso, *entre outros*.

O Azure Key Vault permite um nível de controle muito detalhado do uso de segredos do aplicativo sem a necessidade de permitir que qualquer pessoa os conheça. Os segredos podem até mesmo ser girados para segurança avançada, sem interromper o desenvolvimento ou as operações.

Os aplicativos precisam ser registrados no Active Directory da organização, para que possam usar o Key Vault.

Confira a [documentação Conceitos do Key Vault](#) para obter mais detalhes.

## Implementando o controle de versão em ASP.NET Web APIs

À medida que os requisitos de negócios mudam, novas coleções de recursos podem ser adicionadas, as relações entre os recursos podem mudar e a estrutura dos dados nos recursos pode ser corrigida. Atualizar uma API Web para lidar com novos requisitos é um processo relativamente simples, mas você precisa considerar os efeitos que essas alterações causarão nos aplicativos cliente que consomem a API Web. Embora o desenvolvedor que projeta e implementa uma API da Web tenha controle total sobre essa API, o desenvolvedor não tem o mesmo grau de controle sobre os aplicativos cliente que podem ser criados por organizações de terceiros operando remotamente.

O controle de versão permite que uma API Web indique as funcionalidades e os recursos que ela expõe. Assim, um aplicativo cliente pode enviar solicitações para uma versão específica de uma funcionalidade ou de um recurso. Há várias abordagens para implementar o controle de versão:

- Controle de versão de URI
- Controle de versão de cadeia de consulta
- Controle de versão de cabeçalho

O controle de versão de cadeia de caracteres de consulta e de URI são os mais simples de implementar. O controle de versão de cabeçalho é uma boa abordagem. No entanto, o controle de versão de cabeçalho não é tão explícito e simples como o controle de versão de URI. Como o controle de versão de URL é o mais simples e o mais explícito, o aplicativo de exemplo eShopOnContainers usa esse o controle de versão de URI.

Com o controle de versão de URI, como no aplicativo de exemplo eShopOnContainers, sempre que você modificar a API Web ou alterar o esquema de recursos, você adicionará um número de versão ao URI de cada recurso. Os URIs existentes devem continuar a operar como antes, retornando os recursos que estão em conformidade com o esquema que corresponde à versão solicitada.

Como mostrado no exemplo de código a seguir, a versão pode ser definida usando o atributo Rota no controlador de API Web, o que torna a versão explícita no URI (v1, neste caso).

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

Esse mecanismo de controle de versão é simples e depende do roteamento que o servidor faz da solicitação para o ponto de extremidade apropriado. No entanto, para obter um controle de versão mais sofisticado, e o melhor método ao usar o REST, você deve usar hipermídia e implementar o [HATEOAS \(Hypertext as the Engine of Application State\)](#).

## Recursos adicionais

- Scott Hanselman. [ASP.NET Core facilitar o controle de versão da API Web RESTful](#)  
<https://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>
- [Controle de versão de uma API Web RESTful](#)  
[/azure/architecture/best-practices/api-design#versioning-a-restful-web-api](https://azure.microsoft.com/en-us/architecture/best-practices/api-design/#versioning-a-restful-web-api)
- Roy de campo. [Controle de versão, hipermídia e REST](#)  
<https://www.infoq.com/articles/roy-fielding-on-versioning>

## Gerando metadados de descrição do Swagger para a API Web ASP.NET Core

O [Swagger](#) é uma estrutura de software livre muito usada, apoiada por um grande ecossistema de ferramentas que ajuda a projetar, compilar, documentar e consumir APIs RESTful. Ele está se tornando o padrão no domínio de metadados de descrição de APIs. Você deve incluir metadados de descrição do Swagger com qualquer tipo de

microserviço, microserviços controlados por dados ou microserviços avançados controlados por domínio (conforme explicado na seção a seguir).

A essência do Swagger é a especificação do Swagger, que são metadados de descrição de API em um arquivo JSON ou YAML. A especificação cria o contrato RESTful para a API, detalhando todos os recursos e as operações em dois formatos, legível por pessoas e legível por computadores, para facilitar o desenvolvimento, a descoberta e a integração.

A especificação é a base da OAS (especificação OpenAPI) e é desenvolvida em uma comunidade aberta, transparente e colaborativa para padronizar a maneira que as interfaces RESTful são definidas.

A especificação define a estrutura de como um serviço pode ser descoberto e como seus recursos são entendidos. Para obter mais informações, incluindo um editor na Web e exemplos de especificações do Swagger de empresas como Spotify, Uber, Slack e Microsoft, consulte o site do Swagger (<https://swagger.io>).

### Por que usar Swagger?

As principais razões para gerar metadados do Swagger para suas APIs são as seguintes.

**Capacidade para outros produtos consumirem e integrarem suas APIs automaticamente.** Dezenas de produtos e [ferramentas comerciais](#) e diversas [estruturas e bibliotecas](#) são compatíveis com o Swagger. A Microsoft tem produtos e ferramentas de alto nível que podem consumir automaticamente as APIs baseadas no Swagger, como os seguintes:

- [AutoRest](#). É possível gerar classes de cliente do .NET automaticamente para chamar o Swagger. Essa ferramenta pode ser usada na CLI e também se integra ao Visual Studio, facilitando o uso por meio da GUI.
- [Microsoft Flow](#). É possível [usar e integrar sua API](#) automaticamente em um fluxo de trabalho do Microsoft Flow de alto nível, sem precisar de nenhuma habilidade de programação.
- [Microsoft PowerApps](#). É possível consumir a API Automaticamente de [aplicativos móveis PowerApps](#) criados com o [PowerApps Studio](#), sem precisar de nenhuma habilidade de programação.
- [Aplicativos Lógicos do Serviço de Aplicativo do Azure](#). É possível [usar e integrar sua API a um Aplicativo Lógico do Serviço de Aplicativo do Azure](#) automaticamente, sem precisar de nenhuma habilidade de programação.

**Capacidade de gerar a documentação da API automaticamente.** Ao criar APIs RESTful em grande escala, como aplicativos complexos baseados em microserviços, você precisa lidar com vários pontos de extremidade com modelos de dados diferentes usados no conteúdo de solicitação e de resposta. Ter uma documentação adequada e um gerenciador de API sólido, como o Swagger oferece, é fundamental para o sucesso da API e da adoção pelos desenvolvedores.

Os metadados do Swagger são o que o Microsoft Flow, o PowerApps e os Aplicativos Lógicos do Azure usam para entender como usar as APIs e conectar-se a elas.

Há várias opções para automatizar a geração de metadados do Swagger para aplicativos de API REST do ASP.NET Core, na forma de páginas de ajuda de API funcional, baseadas na *swagger-ui*.

Provavelmente, o melhor conhecimento é o [swashbuckle](#) que está sendo usado atualmente no [eShopOnContainers](#) e abordaremos em detalhes neste guia, mas também há a opção de usar o [NSwag](#), que pode gerar clientes de # API do typescript e do c, bem como # controladores c, de uma especificação Swagger ou openapi e até mesmo examinando o. dll que contém os controladores, usando [NSwagStudio](#).

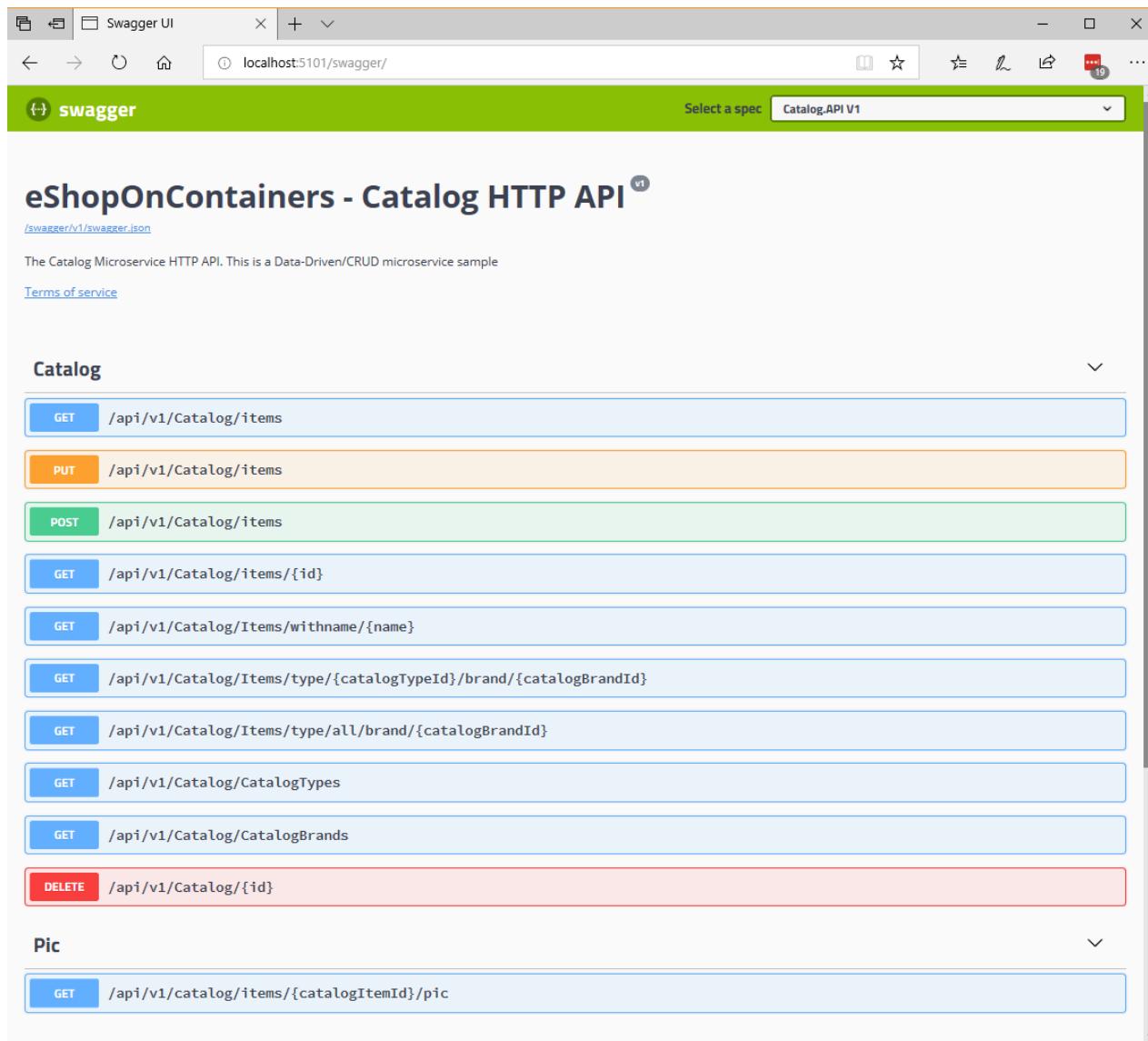
### Como automatizar a geração de metadados do Swagger para a API com o pacote NuGet Swashbuckle

A geração manual de metadados do Swagger (em um arquivo JSON ou YAML) pode ser um trabalho entediante. No entanto, é possível automatizar a descoberta de serviços do ASP.NET Web API usando o [pacote NuGet Swashbuckle](#) para gerar dinamicamente os metadados do Swagger para a API.

O Swashbuckle gera automaticamente os metadados do Swagger para os projetos do ASP.NET Web API. Ele é compatível com projetos da API Web ASP.NET Core, do ASP.NET Web API tradicional e de outros tipos, como os microsserviços de Aplicativo de API do Azure, Aplicativo Móvel Azure, Azure Service Fabric com base no ASP.NET. Ele também é compatível com a API Web simples implantada em contêineres, como no aplicativo de referência.

O Swashbuckle combina o Gerenciador de API e o Swagger ou o [swagger-ui](#) para fornecer uma experiência avançada de descoberta e de documentação aos consumidores da API. Além do mecanismo gerador de metadados do Swagger, o Swashbuckle também contém uma versão incorporada do swagger-ui, que ele oferecerá automaticamente quando for instalado.

Isso significa que você pode complementar a API com uma ótima interface do usuário de descoberta para ajudar os desenvolvedores a usarem a API. Isso exige uma quantidade muito pequena de código e manutenção porque ela é gerada automaticamente, permitindo que você se concentre na API. O resultado do Gerenciador de API é semelhante ao da Figura 6-8.



**Figura 6-8.** Gerenciador de API do Swashbuckle baseado nos metadados do Swagger – microsserviço de catálogo do eShopOnContainers

A documentação da API de interface do usuário do Swagger gerada pelo Swashbuckle inclui todas as ações publicadas. O Gerenciador de API não é o mais importante aqui. Quando a API Web consegue se descrever nos metadados do Swagger, ela pode ser usada diretamente por meio das ferramentas baseadas no Swagger, incluindo geradores de código de classe de proxy de cliente que podem direcionar a várias plataformas. Por exemplo, conforme mencionado, o [AutoRest](#) gera as classes de cliente do .NET automaticamente. Mas ferramentas adicionais como [swagger-codegen](#) também estão disponíveis, permitindo automaticamente a geração de código

das bibliotecas clientes da API, de stubs de servidor e da documentação.

Atualmente, o Swashbuckle consiste em cinco pacotes NuGet internos no metapacote de alto nível [swashbuckle.AspNetCore](#) para aplicativos ASP.NET Core.

Depois de instalar esses pacotes NuGet em seu projeto de API Web, você precisa configurar o Swagger na classe de inicialização, como no seguinte código **simplificado**:

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

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

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

        // Other ConfigureServices() code...
    }

    public void Configure(IApplicationBuilder app,
        IHostingEnvironment env,
        ILoggerFactory loggerFactory)
    {
        // Other Configure() code...
        // ...
        app.UseSwagger()
            .UseSwaggerUI(c =>
            {
                c.SwaggerEndpoint("/swagger/v1/swagger.json", "My API V1");
            });
    }
}
```

Depois que isso for feito, você poderá iniciar o aplicativo e procurar o JSON do Swagger e os pontos de extremidade da interface do usuário usando URLs como estas:

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

```
http://<your-root-url>/swagger/
```

Você já viu a interface do usuário gerada, criada pelo Swashbuckle para uma URL como <http://<your-root-url>/swagger>. Na Figura 6-9, veja também como é possível testar qualquer método de API.

The screenshot shows the Swagger UI interface for testing an API. The URL is `localhost:5101/swagger/`. The method is `GET` and the endpoint is `/api/v1/Catalog/items`. The parameters are:

- `pageSize`: integer (query) value: 12
- `pageIndex`: integer (query) value: 0
- `ids`: string (query) value: ids

Buttons: `Execute` (highlighted in blue), `Clear`.

Responses: `Curl` command: `curl -X GET "http://localhost:5101/api/v1/Catalog/items?pageSize=12&pageIndex=0" -H "accept: text/plain"`, Request URL: `http://localhost:5101/api/v1/Catalog/items?pageSize=12&pageIndex=0`, Server response: `Code` 200, `Details` Response body:

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

Figura 6-9. Interface do usuário do Swashbuckle testando o método da API de itens/catálogo

O detalhe da API de interface do usuário do Swashbuckle apresenta uma amostra da resposta e pode ser usado para executar a API real, que é ótima para a descoberta do desenvolvedor. A Figura 6-10 mostra os metadados JSON do Swagger gerados por meio do microsserviço eShopOnContainers (que é o que as ferramentas usam em segundo plano) quando você solicita `http://<your-root-url>/swagger/v1/swagger.json` usando o [Postman](#).

```

1  {
2    "swagger": "2.0",
3    "info": {
4      "version": "v1",
5      "title": "eShopOnContainers - Catalog HTTP API",
6      "description": "The Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice sample",
7      "termsOfService": "Terms Of Service"
8    },
9    "basePath": "/",
10   "paths": {
11     "/api/v1/Catalog/Items": {
12       "get": {
13         "tags": [
14           "Catalog"
15         ],
16         "operationId": "ApiV1CatalogItemsGet",
17         "consumes": [],
18         "produces": [],
19         "parameters": [
20           {
21             "name": "pageSize",
22             "in": "modelbinding",
23             "required": false,
24             "type": "integer",
25             "format": "int32"
26           }
27         ]
28       }
29     }
30   }
31 }
```

**Figura 6-10.** Metadados JSON do Swagger

É simples assim. E como são gerados automaticamente, os metadados do Swagger aumentarão à medida que você adicionar mais funcionalidades à API.

### Recursos adicionais

- **ASP.NET Web API páginas de ajuda usando o Swagger**  
<https://docs.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger>
- **Introdução ao swashbuckle e ASP.NET Core**  
<https://docs.microsoft.com/aspnet/core/tutorials/getting-started-with-swashbuckle>
- **Introdução ao NSwag e ASP.NET Core**  
<https://docs.microsoft.com/aspnet/core/tutorials/getting-started-with-nswag>

[ANTERIOR](#) [AVANÇAR](#)

# Definindo o aplicativo de vários contêineres com o docker-compose.yml

08/04/2020 • 28 minutes to read • [Edit Online](#)

Neste guia, o arquivo [docker-compose.yml](#) foi introduzido na seção [Passo 4. Defina seus serviços em docker-compose.yml ao construir um aplicativo Docker de vários contêineres](#). No entanto, há outras maneiras de usar os arquivos docker-compose e vale a pena explorá-las com mais detalhes.

Por exemplo, você pode descrever explicitamente como deseja implantar o aplicativo de vários contêineres no arquivo docker-compose.yml. Opcionalmente, você também pode descrever como vai criar as imagens personalizadas do Docker. (As imagens personalizadas do Docker também podem ser criadas com a CLI do Docker).

Basicamente, você define cada um dos contêineres que deseja implantar, além de determinadas características para cada implantação de contêiner. Com um arquivo de descrição de implantação de vários contêineres pronto, você poderá implantar toda a solução por meio de uma única ação orquestrada pelo comando da CLI [docker-compose up](#) ou poderá implantá-la de forma transparente no Visual Studio. Caso contrário, você precisará usar a CLI do Docker para implantar contêiner por contêiner, em várias etapas, usando o comando `docker run` na linha de comando. Portanto, cada serviço definido no docker-compose.yml deve especificar exatamente uma imagem ou build. As outras chaves são opcionais e são semelhantes aos correspondentes de `docker run` na linha de comando.

O código YAML a seguir é a definição de um arquivo docker-compose.yml possivelmente global, mas único, do exemplo eShopOnContainers. Esse não é o verdadeiro arquivo docker-compose do eShopOnContainers. Em vez disso, ele é uma versão simplificada e consolidada em um único arquivo, o que não é a melhor maneira de se trabalhar com arquivos docker-compose, como será explicado posteriormente.

```

version: '3.4'

services:
  webmvc:
    image: eshop/webmvc
    environment:
      - CatalogUrl=http://catalog-api
      - OrderingUrl=http://ordering-api
      - BasketUrl=http://basket-api
    ports:
      - "5100:80"
    depends_on:
      - catalog-api
      - ordering-api
      - basket-api

  catalog-api:
    image: eshop/catalog-api
    environment:
      - ConnectionString=Server=sqldata;Initial Catalog=CatalogData;User Id=sa;Password=your@password
    expose:
      - "80"
    ports:
      - "5101:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
    depends_on:
      - sqldata

  ordering-api:
    image: eshop/ordering-api
    environment:
      - ConnectionString=Server=sqldata;Database=Services.OrderingDb;User Id=sa;Password=your@password
    ports:
      - "5102:80"
    #extra hosts can be used for standalone SQL Server or services at the dev PC
    extra_hosts:
      - "CESARDLSURFBOOK:10.0.75.1"
    depends_on:
      - sqldata

  basket-api:
    image: eshop/basket-api
    environment:
      - ConnectionString=sqldata
    ports:
      - "5103:80"
    depends_on:
      - sqldata

  sqldata:
    environment:
      - SA_PASSWORD=your@password
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"

  basketdata:
    image: redis

```

A chave de raiz desse arquivo é `services`. Sob essa tecla, você define os serviços que `docker-compose up` deseja implantar e executar quando executa o comando ou quando você implanta no Visual Studio usando este arquivo `docker-compose.yml`. Nesse caso, o arquivo `docker-compose.yml` tem vários serviços definidos, conforme descrito na tabela a seguir.

NOME DO SERVIÇO	DESCRIÇÃO
webmvc	Contêiner, incluindo o aplicativo MVC ASP.NET Core, que consome os microsserviços do C# do lado do servidor
catálogo-api	Contêiner, incluindo o microsserviço de API Web do ASP.NET Core de Catálogo
ordenação-api	Contêiner, incluindo o microsserviço de API Web do ASP.NET Core de Pedidos
sqldata	Contêiner que executa o SQL Server para Linux armazenando os bancos de dados de microsserviços
cesta-api	Contêiner com o microsserviço de API Web do ASP.NET Core de Cesta
basketdata	Contêiner que executa o serviço de Cache Redis, com o banco de dados da cesta como um Cache Redis

### Um contêiner de API de serviço Web simples

Com foco em um único contêiner, o microsserviço de contêiner catalog-api tem uma definição simples:

```

catalog-api:
  image: eshop/catalog-api
  environment:
    - ConnectionString=Server=sqldata;Initial Catalog=CatalogData;User Id=sa;Password=your@password
  expose:
    - "80"
  ports:
    - "5101:80"
  #extra hosts can be used for standalone SQL Server or services at the dev PC
  extra_hosts:
    - "CESARDLSURFBOOK:10.0.75.1"
  depends_on:
    - sqldata

```

Esse serviço em contêiner tem a seguinte configuração básica:

- É baseado na imagem personalizada **de eshop/catalog-api**. Para simplificar, não há compilação: configuração de tecla no arquivo. Isso significa que a imagem precisa ser previamente compilada (com docker build) ou ser baixada (com o comando docker pull) de qualquer registro do Docker.
- Ele define uma variável de ambiente denominada **ConnectionString**, com a cadeia de conexão a ser usada pelo Entity Framework para acessar a instância do SQL Server que contém o modelo de dados de catálogo. Nesse caso, o mesmo contêiner do SQL Server está mantendo vários bancos de dados. Dessa forma, você precisará de menos memória no computador de desenvolvimento do Docker. No entanto, você também poderia implantar um contêiner do SQL Server para cada banco de dados de microsserviço.
- O nome do SQL Server é **sqldata**, que é o mesmo nome usado para o contêiner que está executando a instância do SQL Server para Linux. Isso é conveniente; ser capaz de usar essa resolução de nome (interna para o host Docker) resolverá o endereço da rede para que você não precise saber o IP interno para os contêineres que você está acessando de outros contêineres.

Como a cadeia de conexão é definida por uma variável de ambiente, você pode definir essa variável por meio de um mecanismo diferente e em outro momento. Por exemplo, você pode definir uma cadeia de conexão diferente durante a implantação em produção nos hosts finais ou fazer isso através dos pipelines de CI/CD no Azure

DevOps Services ou no sistema de DevOps de sua preferência.

- Ele expõe a porta 80 para acesso interno ao serviço **de catalog-api** dentro do host Docker. Atualmente o host é uma VM do Linux, porque ele se baseia em uma imagem do Docker para Linux, mas você também pode configurar o contêiner para ser executado em uma imagem do Windows.
- Ele encaminha a porta 80, exposta no contêiner, para a porta 5101 no computador host do Docker (a VM do Linux).
- Ele vincula o serviço web ao serviço **sqldata** (a instância do SQL Server para banco de dados Linux em execução em um contêiner). Quando você especificar essa dependência, o contêiner catalog-api não será iniciado até que o contêiner sqldata já tenha começado; isso é importante porque o catalog-api precisa ter o banco de dados SQL Server funcionando primeiro. No entanto, esse tipo de dependência de contêiner não é suficiente em muitos casos, porque o Docker verifica apenas no nível de contêiner. Às vezes, o serviço (o SQL Server, neste caso) poderá ainda não estar pronto, portanto, é aconselhável implementar uma lógica de repetição com retirada exponencial no microsserviço cliente. Dessa forma, se um contêiner de dependência não estiver pronto durante um breve período de tempo, o aplicativo continuará resiliente.
- Ele é configurado para permitir o acesso\_a servidores externos: a configuração de hosts extras permite que você acesse servidores externos ou máquinas fora do host Docker (ou seja, fora do VM Linux padrão, que é um host Docker de desenvolvimento), como uma instância local do SQL Server no seu PC de desenvolvimento.

Há também outras `docker-compose.yml` configurações mais avançadas que discutiremos nas seguintes seções.

### **Usando arquivos docker-compose para destinar-se a vários ambientes**

Os `docker-compose.*.yml` arquivos são arquivos de definição e podem ser usados por várias infra-estruturas que entendem esse formato. A ferramenta mais simples é o comando `docker-compose`.

Portanto, ao usar o comando `docker-compose` você pode ter os seguintes principais cenários como destino.

#### **Ambientes de desenvolvimento**

Ao desenvolver aplicativos, é importante ser capaz de executar um aplicativo em um ambiente de desenvolvimento isolado. Você pode usar o comando CLI para criar esse ambiente ou visual studio, que usa `docker-compor` sob as capas.

O arquivo `docker-compose.yml` permite que você configure e documente todas as dependências de serviço do seu aplicativo (outros serviços, cache, bancos de dados, filas, etc.). Usando o comando da CLI `docker-compose`, você pode criar e iniciar um ou mais contêineres para cada dependência com um único comando (`docker-compose up`).

Os `docker-compose.yml` são arquivos de configuração interpretados pelo mecanismo do Docker, mas também servem como arquivos de documentação convenientes, com informações sobre a composição de seu aplicativo para vários contêineres.

#### **Ambientes de teste**

Uma parte importante de qualquer processo de CD (implantação contínua) ou CI (integração contínua) são os testes de unidade e testes de integração. Esses testes automatizados requerem um ambiente isolado para que não sejam afetados pelos usuários ou qualquer outra alteração nos dados do aplicativo.

Com o Docker Compose, você pode criar e destruir esse ambiente isolado muito facilmente em alguns comandos do seu prompt de comando ou scripts, como os seguintes comandos:

```
docker-compose -f docker-compose.yml -f docker-compose-test.override.yml up -d  
./run_unit_tests  
docker-compose -f docker-compose.yml -f docker-compose-test.override.yml down
```

### **Implantações de produção**

Você também pode usar o Compose para implantar em um mecanismo remoto do Docker. Um caso comum é a implantação em uma única instância de host do Docker (como uma VM ou servidor de produção, provisionados com o [Docker Machine](#)).

Se estiver usando qualquer outro orquestrador (Azure Service Fabric, Kubernetes, etc.), precisará adicionar definições de configuração de instalação e metadados, como aquelas em docker-compose.yml, mas no formato exigido pelo outro orquestrador.

De qualquer maneira, o docker-compose é uma ferramenta conveniente e um formato de metadados para fluxos de trabalho de desenvolvimento, teste e produção, embora o fluxo de trabalho de produção possa variar no orquestrador que você está usando.

### Usando vários arquivos docker-compose para lidar com diversos ambientes

Ao destinar-se a ambientes diferentes, você deve usar vários arquivos compose. Isso permite que você crie diversas variantes de configuração, dependendo do ambiente.

#### Substituindo o arquivo base docker-compose

Você pode usar um único arquivo docker-compose.yml, como nos exemplos simplificados mostrados nas seções anteriores. No entanto, isso não é recomendável para a maioria dos aplicativos.

Por padrão, o Compose lê dois arquivos, um docker-compose.yml e um arquivo docker-compose.override.yml opcional. Conforme mostrado na Figura 6-11, quando você estiver usando o Visual Studio e habilitar o suporte ao Docker, o Visual Studio também criará um arquivo docker-compose.vs.debug.g.yml adicional para depurar o aplicativo. Dê uma olhada nesse arquivo na pasta obj\ Docker\ da pasta da solução principal.

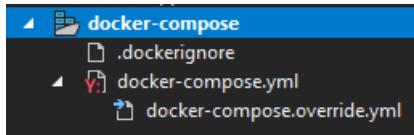


Figura 6-11. arquivos de composição de docker no Visual Studio 2019

estrutura de arquivo de projeto de composição de docker:

- *.dockerignore* - usado para ignorar arquivos
- *docker-compose.yml* - usado para compor microsserviços
- *docker-compose.override.yml* - usado para configurar ambiente de microsserviços

Edita os arquivos docker-compose com qualquer editor, como o Visual Studio Code ou o Sublime, e execute o aplicativo com o comando docker-compose up.

Por convenção, o arquivo docker-compose.yml contém sua configuração base e outras configurações estáticas. Isso significa que a configuração do serviço não deve ser alterada de acordo com o ambiente de implantação que você tem como destino.

O arquivo docker-compose.override.yml, como o próprio nome sugere, contém definições de configuração que substituem a configuração base, como a configuração que depende do ambiente de implantação. Você também pode ter vários arquivos de substituição com nomes diferentes. Os arquivos de substituição geralmente contêm informações adicionais, exigidas pelo aplicativo, bem como as informações específicas de um ambiente ou uma implantação.

#### Destinando-se a vários ambientes

Um caso de uso típico é aquele em que você define vários arquivos compose para destinar-se a vários ambientes, como de produção, de preparo, de CI ou de desenvolvimento. Para dar suporte a essas diferenças, você pode dividir a configuração do Compose em vários arquivos, conforme mostrado na Figura 6-12.

## Multiple docker-compose files

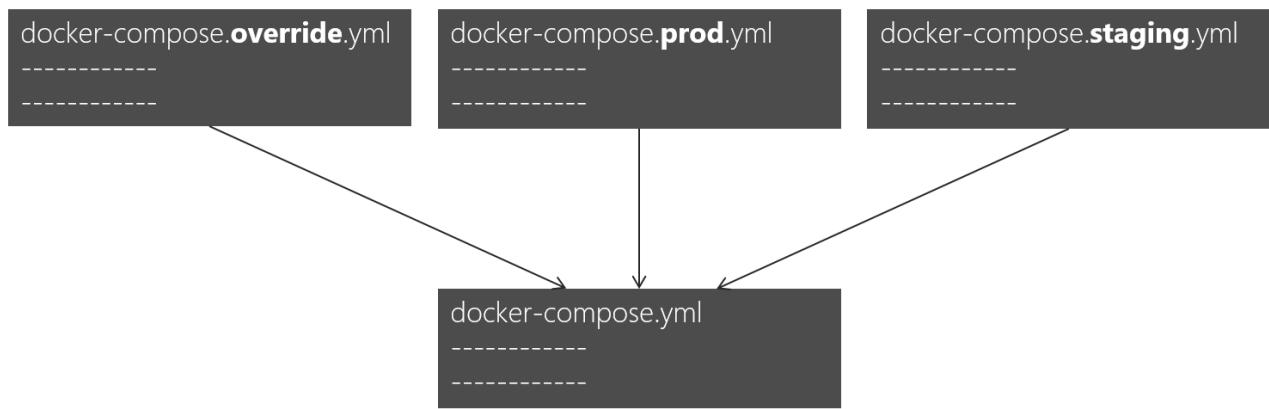


Figura 6-12. Vários arquivos docker-compose substituindo valores no arquivo base docker-compose.yml

Você pode combinar vários arquivos docker-compose\*.yml para lidar com diferentes ambientes. Você inicia com o arquivo base docker-compose.yml. Este arquivo base contém as configurações base ou estática que não mudam dependendo do ambiente. Por exemplo, o aplicativo eShopOnContainers tem o seguinte arquivo docker-compose.yml (simplificado com menos serviços) como o arquivo base.

```

#docker-compose.yml (Base)
version: '3.4'
services:
  basket-api:
    image: eshop/basket-api:${TAG:-latest}
    build:
      context: .
      dockerfile: src/Services/Basket/Basket.API/Dockerfile
    depends_on:
      - basketdata
      - identity-api
      - rabbitmq

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

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

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

  sqldata:
    image: mcr.microsoft.com/mssql/server:2017-latest

  nosqldata:
    image: mongo

  basketdata:
    image: redis

  rabbitmq:
    image: rabbitmq:3-management

```

Os valores do arquivo base docker-compose.yml não devem se alterar devido aos diferentes ambientes de implantação de destino.

Se você se concentrar na definição do serviço webmvc, por exemplo, verá como essa informação é muito semelhante independentemente do ambiente ao qual você se destina. Você tem as seguintes informações:

- O nome do serviço: webmvc.

- Imagem personalizada do contêiner: eshop/webmvc.
- O comando para compilar a imagem personalizada do Docker, que indica qual Dockerfile usar.
- As dependências de outros serviços, para que este contêiner não se inicie até que os outros contêineres de dependência sejam iniciados.

Você pode ter outras configurações, mas o ponto importante é que, no arquivo base docker-compose.yml, é necessário definir apenas as informações que são comuns entre ambientes. Então, no docker-compose.override.yml ou em arquivos semelhantes de produção ou de preparo, você deve colocar a configuração específica para cada ambiente.

Normalmente, o docker-compose.override.yml é usado para o ambiente de desenvolvimento, como no exemplo a seguir, do eShopOnContainers:

```
#docker-compose.override.yml (Extended config for DEVELOPMENT env.)
version: '3.4'

services:
# Simplified number of services here:

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

ports:
- "5103:80"

catalog-api:
environment:
- ASPNETCORE_ENVIRONMENT=Development
- ASPNETCORE_URLS=http://0.0.0.0:80
- ConnectionString=${ESHOP_AZURE_CATALOG_DB:-}
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=Pass@word
- PicBaseUrl=${ESHOP_AZURE_STORAGE_CATALOG_URL:-http://localhost:5202/api/v1/catalog/items/[0]/pic/}
- EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
- EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
- EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
- AzureStorageAccountName=${ESHOP_AZURE_STORAGE_CATALOG_NAME}
- AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_CATALOG_KEY}
- UseCustomizationData=True
- AzureServiceBusEnabled=False
- AzureStorageEnabled=False
- ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
- OrchestratorType=${ORCHESTRATOR_TYPE}

ports:
- "5101:80"

marketing-api:
environment:
- ASPNETCORE_ENVIRONMENT=Development
- ASPNETCORE_URLS=http://0.0.0.0:80
- ConnectionString=${ESHOP_AZURE_MARKETING_DB:-}
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.MarketingDb;User Id=sa;Password=Pass@word
- MongoConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}
```

```

- MongoDB=MarketingDb
- EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
- EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
- EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
- identityUrl=http://identity-api
- IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
- CampaignDetailFunctionUri=${ESHOP_AZUREFUNC_CAMPAIGN_DETAILS_URI}
- PicBaseUrl=${ESHOP_AZURE_STORAGE_MARKETING_URL:-http://localhost:5110/api/v1/campaigns/[0]/pic/}
- AzureStorageAccountName=${ESHOP_AZURE_STORAGE_MARKETING_NAME}
- AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_MARKETING_KEY}
- AzureServiceBusEnabled=False
- AzureStorageEnabled=False
- ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
- OrchestratorType=${ORCHESTRATOR_TYPE}
- UseLoadTest=${USE_LOADTEST:-False}

ports:
- "5110:80"

webmvc:
environment:
- ASPNETCORE_ENVIRONMENT=Development
- ASPNETCORE_URLS=http://0.0.0.0:80
- PurchaseUrl=http://webshoppingapigw
- IdentityUrl=http://10.0.75.1:5105
- MarketingUrl=http://webmarketingapigw
- CatalogUrlHC=http://catalog-api/hc
- OrderingUrlHC=http://ordering-api/hc
- IdentityUrlHC=http://identity-api/hc
- BasketUrlHC=http://basket-api/hc
- MarketingUrlHC=http://marketing-api/hc
- PaymentUrlHC=http://payment-api/hc
- SignalrHubUrl=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5202
- UseCustomizationData=True
- ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
- OrchestratorType=${ORCHESTRATOR_TYPE}
- UseLoadTest=${USE_LOADTEST:-False}

ports:
- "5100:80"

sqldata:
environment:
- SA_PASSWORD=Pass@word
- ACCEPT_EULA=Y
ports:
- "5433:1433"

nosqldata:
ports:
- "27017:27017"

basketdata:
ports:
- "6379:6379"

rabbitmq:
ports:
- "15672:15672"
- "5672:5672"

```

Neste exemplo, a configuração de substituição de desenvolvimento expõe algumas portas para o host, define variáveis de ambiente com URLs de redirecionamento e especifica cadeias de conexão para o ambiente de desenvolvimento. Todas essas configurações são apenas para o ambiente de desenvolvimento.

Quando você executa `docker-compose up` (ou o inicia no Visual Studio), o comando lerá as substituições automaticamente como se estivesse mesclando os dois arquivos.

Suponha que você queira outro arquivo Compor para o ambiente de produção, com diferentes valores de configuração, portas ou strings de conexão. Você pode criar outro arquivo de substituição, como um arquivo

chamado `docker-compose.prod.yml`, com diferentes configurações e variáveis de ambiente. Esse arquivo poderá ser armazenado em outro repositório GIT ou gerenciado e protegido por uma equipe diferente.

#### Como implantar com um arquivo de substituição específico

Para usar vários arquivos de substituição, ou um arquivo de substituição com um nome diferente, use a opção `-f` com o comando docker-compose e especifique os arquivos. O Compose mescla os arquivos na ordem em que eles são especificados na linha de comando. O exemplo a seguir mostra como implantar com arquivos de substituição.

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

#### Usando variáveis de ambiente em arquivos docker-compose

É conveniente, especialmente em ambientes de produção, ter a possibilidade de obter informações de configuração de variáveis de ambiente, como mostramos em exemplos anteriores. Referencie uma variável de ambiente nos arquivos docker-compose usando a sintaxe  `${MY_VAR}`. A seguinte linha de um arquivo `docker-compose.prod.yml` mostra como referenciar o valor de uma variável de ambiente.

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

As variáveis de ambiente são criadas e inicializadas de diferentes maneiras, dependendo do ambiente de host (Linux, Windows, cluster de nuvem, etc.). Entretanto, uma abordagem conveniente é usar um arquivo `.env`. Os arquivos docker-compose são compatíveis com a declaração de variáveis de ambiente padrão no arquivo `.env`. Esses valores das variáveis de ambiente são os valores padrão. Mas elas podem ser substituídas pelos valores que você definiu em cada um dos ambientes (sistema operacional de host ou variáveis de ambiente do cluster). Coloque esse arquivo `.env` na pasta em que o comando docker-compose é executado.

O exemplo a seguir mostra um arquivo `.env` parecido com o [.env](#) do aplicativo eShopOnContainers.

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

O docker-compose espera que cada linha de um arquivo `.env` esteja no formato `<variável>=<valor>`.

Os valores definidos no ambiente de tempo de execução sempre sobrepõem os valores definidos dentro do arquivo `.env`. Da mesma forma, os valores passados através de argumentos de linha de comando também anulam os valores padrão definidos no arquivo `.env`.

#### Recursos adicionais

- Visão geral do Docker Compose

<https://docs.docker.com/compose/overview/>

- Arquivos de composição múltipla

<https://docs.docker.com/compose/extends/#multiple-compose-files>

#### Criando imagens do Docker do ASP.NET Core otimizadas

Se estiver explorando o Docker e o .NET Core em fontes na Internet, você encontrará Dockerfiles que demonstram a simplicidade da criação de uma imagem do Docker por meio da cópia da fonte em um contêiner. Esses exemplos sugerem que, ao usar uma configuração simples, você pode ter uma imagem do Docker com o ambiente empacotado com seu aplicativo. O exemplo a seguir mostra um Dockerfile simples neste sentido.

```
FROM mcr.microsoft.com/dotnet/core/sdk:3.1
WORKDIR /app
ENV ASPNETCORE_URLS http://+:80
EXPOSE 80
COPY . .
RUN dotnet restore
ENTRYPOINT ["dotnet", "run"]
```

Um Dockerfile como esse funcionará. No entanto, você pode otimizar substancialmente suas imagens, especialmente as imagens de produção.

No modelo de contêiner e microsserviços, você está constantemente iniciando contêineres. O modo comum de uso de contêineres não reinicia um contêiner em suspensão porque o contêiner é descartável. Orquestradores (como o Kubernetes e o Azure Service Fabric) simplesmente criam instâncias de imagens. Isso significa que você precisará otimizar por meio da pré-compilação do aplicativo quando ele for criado, fazendo com que o processo de instanciação seja mais rápido. Quando o contêiner é iniciado, ele deve estar pronto para executar. Não restaure e compile em `dotnet restore` tempo `dotnet build` de execução usando os comandos e CLI, como você pode ver em postagens de blog sobre .NET Core e Docker.

A equipe do .NET está realizando um trabalho importante para tornar o .NET Core e o ASP.NET Core uma estrutura otimizada para contêineres. O .NET Core não é apenas uma estrutura leve com um volume de memória pequeno; a equipe se concentrou em imagens do Docker otimizadas para três cenários principais e as publicou no Registro no Hub do Docker em `dotnet/core`, começando na versão 2.1:

- Desenvolvimento:** A prioridade é a capacidade de iterar rapidamente e depurar mudanças, e onde o tamanho é secundário.
- Build:** A prioridade é compilar o aplicativo, e a imagem inclui binários e outras dependências para otimizar binários.
- Produção:** O foco é a implantação rápida e o início dos contêineres, de modo que essas imagens estão limitadas aos binários e ao conteúdo necessário para executar o aplicativo.

A equipe .NET fornece quatro variantes básicas em `dotnet/core` (no Docker Hub):

- sdk:** para cenários de desenvolvimento e build
- aspnet:** para cenários de produção do ASP.NET
- runtime:** para cenários de produção do .NET
- runtime-deps:** para cenários de produção de [aplicações independentes](#)

Para uma inicialização mais rápida, as imagens de runtime também definem automaticamente `aspnetcore_urls` para a porta 80 e usam o Ngen para criar um cache de imagens nativas de assemblies.

#### Recursos adicionais

- Criando imagens otimizadas do Docker com o ASP.NET Core  
[/archive/blogs/stevelasker/building-optimized-docker-images-with-asp-net-core](#)
- Criando imagens do Docker para .NET Core Applications  
<https://docs.microsoft.com/dotnet/core/docker/building-net-docker-images>

[PRÓXIMO](#)

[ANTERIOR](#)

# Use um servidor de banco de dados em execução como um contêiner

18/03/2020 • 13 minutes to read • [Edit Online](#)

Você pode ter seus bancos de dados (SQL Server, PostgreSQL, MySQL etc.) em servidores autônomo regulares, em clusters locais ou em serviços de PaaS na nuvem como o BD SQL do Azure. No entanto, para ambientes de desenvolvimento e teste, ter seus bancos de dados funcionando como `docker-compose up` contêineres é conveniente, porque você não tem nenhuma dependência externa e simplesmente executar o comando inicia todo o aplicativo. Ter esses bancos de dados como contêineres também é excelente para testes de integração, porque o banco de dados é iniciado no contêiner e sempre é preenchido com os dados de exemplo, assim, os testes podem ser mais previsíveis.

## SQL Server em execução como um contêiner com um banco de dados relacionado a microsserviço

No eShopOnContainers, há um `sqldata` contêiner chamado , como definido no arquivo `docker-compose.yml`, que executa um SQL Server para linux com os bancos de dados SQL para todos os microserviços que precisam de um.

Um ponto-chave nos microsserviços é que cada microserviço possui seus dados relacionados, por isso deve ter seu próprio banco de dados. No entanto, os bancos de dados podem estar em qualquer lugar. Neste caso, todos eles estão no mesmo contêiner para manter os requisitos de memória Docker o mais baixo possível. Tenha em mente que esta é uma solução boa o suficiente para o desenvolvimento e, talvez, para testes, mas não para a produção.

O contêiner do SQL Server no aplicativo de exemplo é configurado com o seguinte código YAML no arquivo `docker-compose.yml`, que é executado quando você executa o `docker-compose up`. Observe que o código YAML consolidou informações de configuração do arquivo genérico `docker-compose.yml` e o arquivo `docker-compose.override.yml`. (Normalmente, separe as configurações de ambiente com as informações de base ou estáticas relacionada à imagem do SQL Server.)

```
sqldata:  
  image: mcr.microsoft.com/mssql/server:2017-latest  
  environment:  
    - SA_PASSWORD=Pass@word  
    - ACCEPT_EULA=Y  
  ports:  
    - "5434:1433"
```

De maneira semelhante, em vez de usar `docker-compose` , o seguinte comando `docker run` pode ser executado naquele contêiner:

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

No entanto, se você estiver implantando um aplicativo multicontêiner, como eShopOnContainers, será mais conveniente usar o comando `docker-compose up` para que ele implante todos os contêineres necessários para o aplicativo.

Quando você inicia esse contêiner do SQL Server pela primeira vez, o contêiner inicializa o SQL Server com a

senha que você fornece. Depois que o SQL Server está em execução como um contêiner, você pode atualizar o banco de dados conectando por meio de qualquer conexão de SQL normal, como SQL Server Management Studio, Visual Studio ou código C#.

O aplicativo eShopOnContainers inicializa cada banco de dados de microsserviço usando dados de exemplo propagando-os com os dados na inicialização, conforme explica a seção a seguir.

Ter o SQL Server em execução como um contêiner não é apenas útil para uma demonstração em que você talvez não tenha acesso a uma instância do SQL Server. Como mencionado, também é ótimo para ambientes de desenvolvimento e teste para que você possa executar facilmente testes de integração começando com uma imagem limpa do SQL Server e dados conhecidos propagando novos dados de exemplo.

#### Recursos adicionais

- Execute a imagem sql server docker no Linux, Mac ou Windows  
[/sql/linux/sql-server-linux-setup-docker](#)
- Conectar e consultar o SQL Server no Linux com sqlcmd  
[/sql/linux/sql-server-linux-connect-and-query-sqlcmd](#)

## Propagação com os dados de teste na inicialização do aplicativo Web

Para adicionar dados ao banco de dados quando o aplicativo for `Main` iniciado, `Program` você pode adicionar código como o seguinte ao método na classe do projeto de API da Web:

```

public static int Main(string[] args)
{
    var configuration = GetConfiguration();

    Log.Logger = CreateSerilogLogger(configuration);

    try
    {
        Log.Information("Configuring web host ({ApplicationContext}...)", AppName);
        var host = CreateHostBuilder(configuration, args);

        Log.Information("Applying migrations ({ApplicationContext}...)", AppName);
        host.MigrateDbContext<CatalogContext>((context, services) =>
        {
            var env = services.GetService<IWebHostEnvironment>();
            var settings = services.GetService<IOptions<CatalogSettings>>();
            var logger = services.GetService<ILogger<CatalogContextSeed>>();

            new CatalogContextSeed()
                .SeedAsync(context, env, settings, logger)
                .Wait();
        })
        .MigrateDbContext<IntegrationEventLogContext>((_, __) => { });

        Log.Information("Starting web host ({ApplicationContext}...)", AppName);
        host.Run();

        return 0;
    }
    catch (Exception ex)
    {
        Log.Fatal(ex, "Program terminated unexpectedly ({ApplicationContext})!", AppName);
        return 1;
    }
    finally
    {
        Log.CloseAndFlush();
    }
}

```

Há uma ressalva importante ao aplicar migrações e semear um banco de dados durante a inicialização do contêiner. Uma vez que o servidor de banco de dados pode não estar disponível por qualquer motivo, você deve lidar com repetições enquanto espera que o servidor esteja disponível. Esta lógica de repetição `MigrateDbContext()` é tratada pelo método de extensão, conforme mostrado no código a seguir:

```

public static IWebHost MigrateDbContext<TContext>(
    this IWebHost host,
    Action<TContext>
        seeder)
    where TContext : DbContext
{
    var underK8s = host.IsInKubernetes();

    using (var scope = host.Services.CreateScope())
    {
        var services = scope.ServiceProvider;

        var logger = services.GetRequiredService<ILogger<TContext>>();

        var context = services.GetService<TContext>();

        try
        {
            logger.LogInformation("Migrating database associated with context {DbContextName}",
typeof(TContext).Name);

            if (underK8s)
            {
                InvokeSeeder(seeder, context, services);
            }
            else
            {
                var retry = Policy.Handle<SqlException>()
                    .WaitAndRetry(new TimeSpan[]
                    {
                        TimeSpan.FromSeconds(3),
                        TimeSpan.FromSeconds(5),
                        TimeSpan.FromSeconds(8),
                    });

                //if the sql server container is not created on run docker compose this
                //migration can't fail for network related exception. The retry options for DbContext only
                //apply to transient exceptions
                // Note that this is NOT applied when running some orchestrators (let the orchestrator to
                recreate the failing service)
                retry.Execute(() => InvokeSeeder(seeder, context, services));
            }

            logger.LogInformation("Migrated database associated with context {DbContextName}",
typeof(TContext).Name);
        }
        catch (Exception ex)
        {
            logger.LogError(ex, "An error occurred while migrating the database used on context
{DbContextName}", typeof(TContext).Name);
            if (underK8s)
            {
                throw; // Rethrow under k8s because we rely on k8s to re-run the pod
            }
        }
    }

    return host;
}

```

O código a seguir na classe CatalogContextSeed personalizada preenche os dados.

```

public class CatalogContextSeed
{
    public static async Task SeedAsync(IApplicationBuilder applicationBuilder)
    {
        var context = (CatalogContext)applicationBuilder
            .ApplicationServices.GetService(typeof(CatalogContext));
        using (context)
        {
            context.Database.Migrate();
            if (!context.CatalogBrands.Any())
            {
                context.CatalogBrands.AddRange(
                    GetPreconfiguredCatalogBrands());
                await context.SaveChangesAsync();
            }
            if (!context.CatalogTypes.Any())
            {
                context.CatalogTypes.AddRange(
                    GetPreconfiguredCatalogTypes());
                await context.SaveChangesAsync();
            }
        }
    }

    static IEnumerable<CatalogBrand> GetPreconfiguredCatalogBrands()
    {
        return new List<CatalogBrand>()
        {
            new CatalogBrand() { Brand = "Azure" },
            new CatalogBrand() { Brand = ".NET" },
            new CatalogBrand() { Brand = "Visual Studio" },
            new CatalogBrand() { Brand = "SQL Server" }
        };
    }

    static IEnumerable<CatalogType> GetPreconfiguredCatalogTypes()
    {
        return new List<CatalogType>()
        {
            new CatalogType() { Type = "Mug" },
            new CatalogType() { Type = "T-Shirt" },
            new CatalogType() { Type = "Backpack" },
            new CatalogType() { Type = "USB Memory Stick" }
        };
    }
}

```

Quando você executa testes de integração, é útil ter uma maneira de gerar dados consistentes com seus testes de integração. Poder criar tudo do zero, incluindo uma instância do SQL Server em execução em um contêiner, é ótimo para ambientes de teste.

## Banco de dados InMemory do EF Core versus SQL Server em execução como um contêiner

Outra boa opção para executar testes é usar o provedor de banco de dados do Entity Framework InMemory. Você pode especificar essa configuração no método ConfigureServices da classe de Inicialização em seu projeto de API da Web:

```

public class Startup
{
    // Other Startup code ...
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton< IConfiguration>(Configuration);
        // DbContext using an InMemory database provider
        services.AddDbContext< CatalogContext >(opt => opt.UseInMemoryDatabase());
        // (Alternative: DbContext using a SQL Server provider
        // services.AddDbContext< CatalogContext >(c =>
        //{
        //     c.UseSqlServer(Configuration["ConnectionString"]);
        // }
        //));
    }

    // Other Startup code ...
}

```

No entanto, há um problema importante. O banco de dados na memória não é compatível com muitas restrições específicas de um determinado banco de dados. Por exemplo, você pode adicionar um índice exclusivo em uma coluna em seu modelo EF Core e escrever um teste com relação ao seu banco de dados na memória para verificar se ele não permite que você adicione um valor duplicado. Porém, quando você estiver usando o banco de dados na memória, não poderá manipular índices exclusivos em uma coluna. Portanto, o banco de dados em memória não se comporta exatamente como um banco de dados real do SQL Server: ele não emula restrições específicas do banco de dados.

Mesmo assim, um banco de dados na memória é útil para testes e protótipos. Porém, se você quiser criar testes de integração precisos que levem em conta o comportamento de uma implementação do banco de dados específica, você precisará usar um banco de dados real como o SQL Server. Para essa finalidade, executar o SQL Server em um contêiner é uma ótima escolha e mais preciso do que o provedor de banco de dados do EF Core InMemory.

## Usando um Serviço de Cache Redis em execução em um contêiner

Você pode executar o Redis em um contêiner, especialmente para desenvolvimento e teste e para cenários de prova de conceito. Este cenário é conveniente, porque você pode ter todas as suas dependências em execução em contêineres, não apenas para os computadores de desenvolvimento locais, mas para seus ambientes de teste em seus pipelines CI/CD.

No entanto, quando você executa o Redis em produção, é melhor procurar uma solução de alta disponibilidade, como o Redis Microsoft Azure, que é executado como uma PaaS (plataforma como serviço). No seu código, basta alterar suas cadeias de caracteres de conexão.

O Redis fornece uma imagem do Docker com Redis. Essa imagem está disponível no Hub do Docker nesta URL:

[https://hub.docker.com/\\_/redis/](https://hub.docker.com/_/redis/)

Você pode executar diretamente um contêiner do Docker Redis executando o seguinte comando da CLI do Docker em seu prompt de comando:

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

A imagem do Redis inclui expose:6379 (a porta usada pelo Redis), de modo que a vinculação de contêiner padrão o tornará automaticamente disponível aos contêineres vinculados.

No eShopOnContainers, `basket-api` o microserviço usa um cache Redis em execução como um contêiner. Esse `basketdata` contêiner é definido como parte do arquivo `docker-compose.yml` de vários contêineres, conforme mostrado no exemplo a seguir:

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

Este código no docker-compose.yml define `basketdata` um contêiner nomeado com base na imagem redis e publicando a porta 6379 internamente. Isso significa que ele só estará acessível a partir de outros contêineres executados dentro do host Docker.

Finalmente, no arquivo `docker-compose.override.yml`, o `basket-api` microserviço para a amostra eShopOnContainers define a string de conexão a ser usada para o contêiner Redis:

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

Como mencionado anteriormente, o `basketdata` nome do microserviço é resolvido pelo DNS da rede interna do Docker.

[PRÓXIMO](#)

[ANTERIOR](#)

# Implementando comunicação baseada em evento entre microsserviços (eventos de integração)

09/04/2020 • 14 minutes to read • [Edit Online](#)

Conforme descrito anteriormente, quando você usa comunicação baseada em evento, um microsserviço publica um evento quando algo importante acontece, como quando ele atualiza uma entidade de negócios. Outros microsserviços assinam esses eventos. Quando um microsserviço recebe um evento, ele pode atualizar suas próprias entidades de negócios, o que pode levar à publicação de mais eventos. Essa é a essência do conceito de consistência eventual. Este sistema de publicação/assinatura normalmente é executado por meio de uma implementação de um barramento de evento. O barramento de evento pode ser criado como uma interface com a API necessária para assinar e cancelar a assinatura de eventos e eventos de publicação. Também pode ter uma ou mais implementações com base em qualquer comunicação de mensagens ou entre processos, como uma fila de mensagens ou um barramento de serviço que seja compatível com a comunicação assíncrona e um modelo de publicação/assinatura.

Você pode usar eventos para implementar transações comerciais que abranjam vários serviços, o que lhe dá consistência eventual entre esses serviços. Uma transação eventualmente consistente consiste em uma série de ações distribuídas. Em cada ação, o microsserviço atualiza uma entidade de negócios e publica um evento que dispara a próxima ação. Figura 6-18 abaixo, mostra um evento PriceUpdated publicado através de ônibus de eventos, de modo que a atualização de preços é propagada para a Cesta e outros microsserviços.

## Implementing asynchronous event-driven communication with an event bus

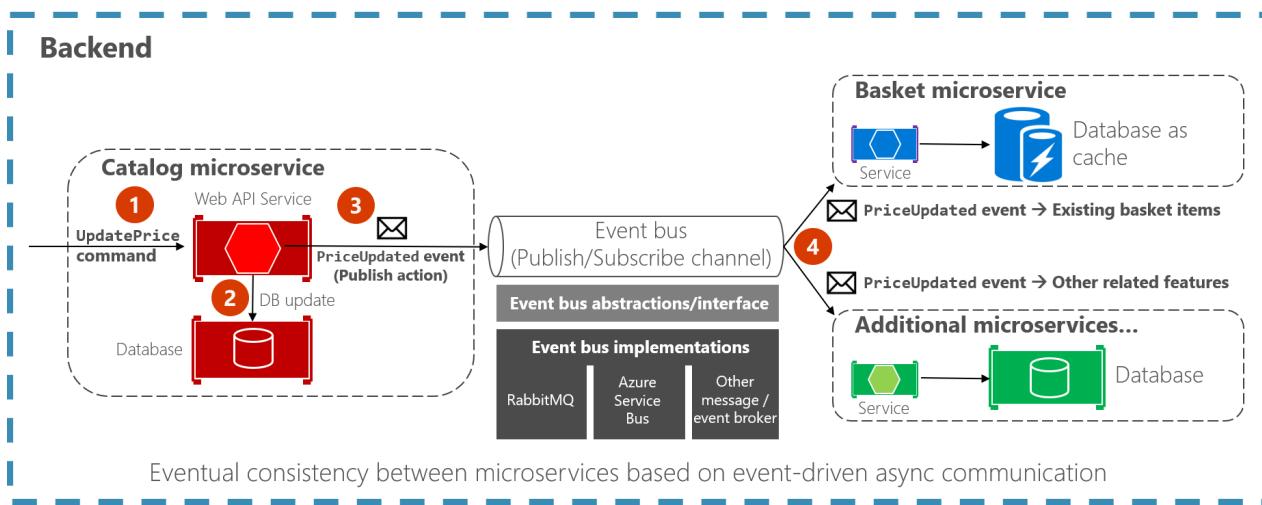


Figura 6-18. Comunicação controlada por evento com base em um barramento de evento

Esta seção descreve como você faz para implementar esse tipo de comunicação com o .NET usando uma interface de barramento de evento genérica, conforme mostra a Figura 6-18. Há várias implementações possíveis, cada uma usando uma tecnologia ou infraestrutura diferente, como RabbitMQ, Barramento de Serviço do Azure ou qualquer outro software livre de terceiros ou barramento de serviço comercial.

## Usando agentes de mensagem e barramentos de serviços para sistemas de produção

Conforme observado na seção de arquitetura, você pode escolher entre várias tecnologias de mensagem para

implementar o barramento do evento abstrato. Porém, essas tecnologias estão em diferentes níveis. Por exemplo, RabbitMQ, um transporte de agente de mensagens, está em um nível inferior a produtos comerciais, como o Barramento de Serviço do Azure, o NServiceBus, o MassTransit ou o Brighter. A maioria desses produtos pode operar sobre RabbitMQ ou Barramento de Serviço do Azure. Sua escolha do produto depende de quantos recursos e quanta escalabilidade imediata você precisa para seu aplicativo.

Para implementar apenas uma prova de conceito de barramento de evento para seu ambiente de desenvolvimento, como fizemos no exemplo de eShopOnContainers, pode ser suficiente uma implementação simples sobre RabbitMQ em execução como um contêiner. Porém, para sistemas críticos e de produção que precisam de alta escalabilidade, talvez você queira avaliar e usar o Barramento de Serviço do Azure.

Se você precisar de abstrações de alto nível e os recursos mais avançados, como [Sagas](#), para processos de execução longa que facilitam o desenvolvimento distribuído barramentos, valerá a pena avaliar outros barramentos de serviço comerciais e de software livre como NServiceBus, MassTransit e Brighter. Nesse caso, as abstrações e a API a serem usadas em geral serão diretamente aquelas fornecidas por esses barramentos de serviço de alto nível, em vez das suas próprias abstrações (como [abstrações de barramento de evento simples fornecidas no eShopOnContainers](#)). Nesse caso, você pode pesquisar os [eShopOnContainers bifurcados usando o NServiceBus](#) (amostra derivada adicional implementada pelo Software Particular).

Claro, você sempre pode construir seus próprios recursos de barramento de serviço em cima de tecnologias de nível inferior como RabbitMQ e Docker, mas o trabalho necessário para "reinventar a roda" pode ser muito caro para um aplicativo corporativo personalizado.

Para reiterar: as abstrações do barramento de evento de exemplo e a implementação apresentada no exemplo do eShopOnContainers devem ser usados apenas como uma prova de conceito. Quando decidir que deseja ter comunicação assíncrona e controlada por evento, conforme explicado na seção atual, você deve escolher o produto de barramento de serviço que melhor atenda às suas necessidades de produção.

## Eventos de integração

Eventos de integração são usados para colocar o estado de domínio em sincronia entre vários microsserviços ou sistemas externos. Isso é feito ao publicar eventos de integração fora do microsserviço. Quando um evento é publicado para vários microsserviços receptores (para tantos microsserviços quantos assinarem o evento de integração), o manipulador de eventos apropriado em cada microsserviço receptor manipulará o evento.

Um evento de integração é basicamente uma classe de retenção de dados, como no exemplo a seguir:

```
public class ProductPriceChangedIntegrationEvent : IntegrationEvent
{
    public int ProductId { get; private set; }
    public decimal NewPrice { get; private set; }
    public decimal OldPrice { get; private set; }

    public ProductPriceChangedIntegrationEvent(int productId, decimal newPrice,
                                                decimal oldPrice)
    {
        ProductId = productId;
        NewPrice = newPrice;
        OldPrice = oldPrice;
    }
}
```

Os eventos de integração podem ser definidos no nível do aplicativo de cada microsserviço, assim, estão separados de outros microsserviços, de modo comparável a como os ViewModels são definidos no cliente e servidor. O que não é recomendável é compartilhar uma biblioteca de eventos de integração comum com vários microsserviços; fazer isso seria acoplar esses microsserviços a uma única biblioteca de dados de definição de evento. Você não deseja fazer isso pelos mesmos motivos que não deseja compartilhar um modelo de domínio

comum entre vários microsserviços: os microsserviços deve ser completamente autônomos.

Há apenas alguns tipos de bibliotecas que você deve compartilhar entre microsserviços. Um são as bibliotecas que são blocos de aplicativo finais, como [API cliente do Barramento de Evento](#), conforme mostrado em eShopOnContainers. Outra vantagem são as bibliotecas que constituem ferramentas que também podem ser compartilhadas como componentes do NuGet, como serializadores JSON.

## O barramento de evento

Um barramento de evento permite comunicação no estilo publicar/assinar entre microsserviços sem a necessidade de os componentes explicitamente estarem cientes uns dos outros, como mostra a Figura 6-19.

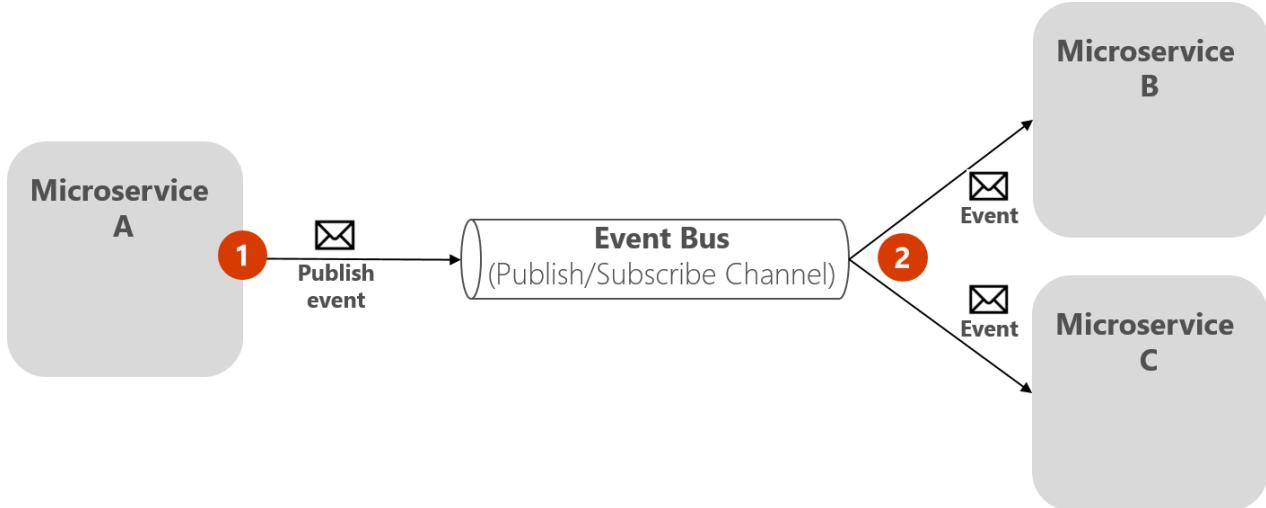


Figura 6-19. Noções básicas sobre publicação/assinatura com um barramento de evento

O diagrama acima mostra que o microserviço A publica para event bus, que distribui para subscrevendo microsserviços B e C, sem que o editor precise conhecer os assinantes. O barramento de evento está relacionado ao padrão Observador e ao padrão de publicação/assinatura.

### Padrão do observador

No [padrão Observador](#), seu objeto primário (conhecido como o Observável) notifica outros objetos de interessados (conhecidos como Observadores) com informações relevantes (eventos).

### Padrão Pub/Sub (Publicar/Assinar)

O objetivo do [padrão Publicar/Assinar](#) é o mesmo que o padrão Observador: você deseja notificar outros serviços quando determinados eventos ocorrem. Mas há uma diferença importante entre os padrões de Observador e Pub/Sub. No padrão do observador, a transmissão é realizada diretamente do observável para os observadores, para que eles "se conheçam" uns aos outros. Porém, ao usar um padrão Pub/Sub, há um terceiro componente, chamado de agente, agente de mensagem ou barramento de evento, que é conhecido tanto pelo publicador quanto pelo assinante. Portanto, ao usar o padrão Pub/Sub, o publicador e os assinantes são precisamente desacoplados graças ao agente de mensagem ou barramento de evento mencionado.

### O barramento de evento ou intermediário

Como você obtém anonimato entre publicador e assinante? Uma maneira fácil é permitir que um intermediário cuide de toda a comunicação. Um barramento de evento é um intermediário.

Um barramento de evento normalmente é composto por duas partes:

- A abstração ou interface.
- Uma ou mais implementações.

Na Figura 6-19, você pode ver como, de um ponto de vista de aplicativo, o barramento de evento é nada mais que

um canal Pub/Sub. A maneira como você implementa essa comunicação assíncrona pode variar. Eles podem ter várias implementações, assim, você pode alternar entre elas, dependendo dos requisitos do ambiente (por exemplo, produção versus ambientes de desenvolvimento).

Na Figura 6-20, você pode ver uma abstração de um barramento de evento com várias implementações baseadas nas tecnologias de mensagem de infraestrutura, como RabbitMQ, Barramento de Serviço do Azure ou outro agente de mensagem/evento.

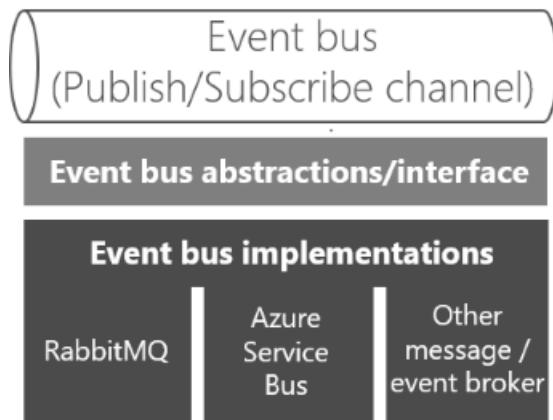


Figura 6- 20. Várias implementações de um barramento de evento

É bom ter o barramento de evento definido por meio de uma interface para que ele possa ser implementado com várias tecnologias, como o Barramento de Serviço do Azure, o RabbitMQ ou outros. No entanto, conforme mencionado anteriormente, usar suas próprias abstrações (a interface de barramento do evento) será bom somente se você precisar de recursos de barramento de evento básicos compatíveis com as suas abstrações. Se você precisar de recursos mais avançados de barramento de serviço, provavelmente deverá usar a API e as abstrações fornecidas pelo seu barramento de serviço comercial preferido, em vez das suas próprias abstrações.

### Definir uma interface de barramento de evento

Vamos começar com algum código de implementação para a interface de barramento de eventos e possíveis implementações para fins de exploração. A interface deve ser genérica e simples, como na interface a seguir.

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

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

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

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

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

O método `Publish` é simples. O barramento de evento difundirá o evento de integração passado a ele para qualquer microsserviço ou até mesmo um aplicativo externo, que assine esse evento. Esse método é usado pelo microsserviço que está publicando o evento.

Os métodos `Subscribe` (você pode ter várias implementações, dependendo dos argumentos) são usados pelos microsserviços que desejam receber eventos. Esse método tem dois argumentos. O primeiro é o evento de

integração a assinar (`IntegrationEvent`). O segundo argumento é o manipulador de eventos de integração (ou método de retorno de chamada), denominado `IIntegrationEventHandler<T>`, a ser executado quando o microsserviço receptor obtiver essa mensagem de evento de integração.

## Recursos adicionais

Algumas soluções de mensagens prontas para a produção:

- **Ônibus de serviço azure**  
</azure/service-bus-messaging/>
- **NServiceBus**  
<https://particular.net/nservicebus>
- **Transporte coletivo**  
<https://masstransit-project.com/>

[PRÓXIMO](#)

[ANTERIOR](#)

# Implementando um barramento de eventos com o RabbitMQ para o ambiente de desenvolvimento ou de teste

10/09/2020 • 6 minutes to read • [Edit Online](#)

Devemos começar dizendo que, se você criar seu barramento de eventos personalizado com base no RabbitMQ em execução em um contêiner, como o aplicativo eShopOnContainers faz, ele deverá ser usado apenas para seus ambientes de desenvolvimento e teste. Não use-o para seu ambiente de produção, a menos que esteja criando-o como parte de um barramento de serviço pronto para produção. Um barramento de eventos personalizado simples talvez não tenha muitos recursos críticos prontos para produção que um barramento de serviço comercial tem.

Uma das implementações personalizadas do barramento de evento no eShopOnContainers é basicamente uma biblioteca que usa a API RabbitMQ. (Há outra implementação baseada no barramento de serviço do Azure.)

A implementação do barramento de eventos com RabbitMQ permite que os microserviços assinem, publiquem e recebam eventos, conforme mostrado na Figura 6-21.

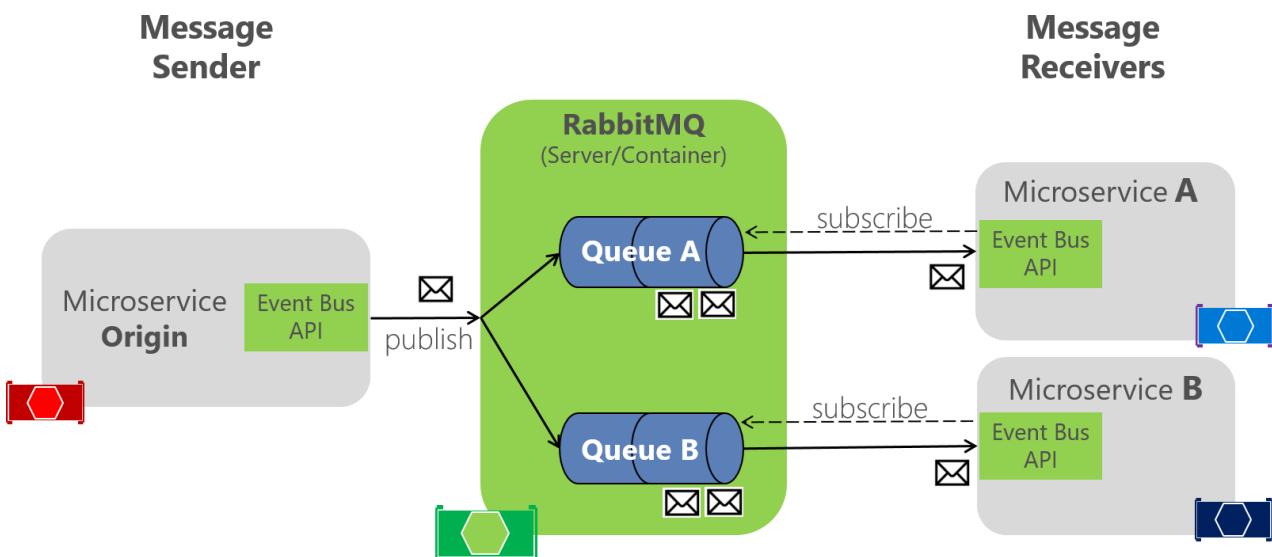


Figura 6-21. Implementação de um barramento de eventos do RabbitMQ

O RabbitMQ é um intermediário entre o publicador da mensagem e os assinantes, para tratar da distribuição. No código, a classe EventBusRabbitMQ implementa a interface IEventBus genérica. Isso se baseia na Injeção de dependência para que você possa trocar dessa versão de Desenvolvimento/Teste para uma versão de produção.

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

A implementação do RabbitMQ de um barramento de eventos de Desenvolvimento/Teste de exemplo é um código clichê. Ele precisa lidar com a conexão com o servidor do RabbitMQ e fornecer o código para a publicação de um evento de mensagem nas filas. Ele também precisa implementar um dicionário de coleções de manipuladores de eventos de integração para cada tipo de evento; esses tipos de evento podem ter uma instanciação diferente e

diferentes assinaturas para cada microsserviço receptor, conforme mostrado na Figura 6-21.

## Implementando um método de publicação simples com RabbitMQ

O código a seguir é uma versão *simplificada* de uma implementação de barramento de eventos do RabbitMQ para demonstrar o cenário completo. Na prática, a conexão não é tratada dessa maneira. Para ver a implementação completa, confira o código real no repositório [dotnet-architecture/eShopOnContainers](#).

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

    public void Publish(IntegrationEvent @event)
    {
        var eventName = @event.GetType().Name;
        var factory = new ConnectionFactory() { HostName = _connectionString };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: _brokerName,
                type: "direct");
            string message = JsonConvert.SerializeObject(@event);
            var body = Encoding.UTF8.GetBytes(message);
            channel.BasicPublish(exchange: _brokerName,
                routingKey: eventName,
                basicProperties: null,
                body: body);
        }
    }
}
```

O [código real](#) do método de publicação no aplicativo eShopOnContainers é aprimorado usando uma política de repetição [Polly](#), que repete a tarefa um determinado número de vezes caso o contêiner do RabbitMQ não esteja pronto. Isso pode ocorrer quando docker-compose está iniciando os contêineres; por exemplo, o contêiner do RabbitMQ pode ser iniciado mais lentamente do que outros contêineres.

Conforme mencionado anteriormente, há muitas configurações possíveis no RabbitMQ. Portanto, esse código deve ser usado apenas para ambientes de Desenvolvimento/Teste.

## Implementando o código de assinatura com a API do RabbitMQ

Como acontece com o código de publicação, o código a seguir é uma simplificação de parte da implementação do barramento de eventos para RabbitMQ. Mais uma vez, geralmente não é necessário alterá-lo, a menos que você o esteja aprimorando.

```

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

    public void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIIntegrationEventHandler<T>
    {
        var eventName = _subsManager.GetEventKey<T>();

        var containsKey = _subsManager.HasSubscriptionsForEvent(eventName);
        if (!containsKey)
        {
            if (!_persistentConnection.IsConnected)
            {
                _persistentConnection.TryConnect();
            }

            using (var channel = _persistentConnection.CreateModel())
            {
                channel.QueueBind(queue: _queueName,
                    exchange: BROKER_NAME,
                    routingKey: eventName);
            }
        }

        _subsManager.AddSubscription<T, TH>();
    }
}

```

Cada tipo de evento tem um canal relacionado para obter eventos do RabbitMQ. Em seguida, é possível ter tantos manipuladores de eventos por tipo de evento e de canal conforme necessário.

O método `Subscribe` aceita um objeto `IIIntegrationEventHandler`, que é como um método de retorno de chamada no microsserviço atual, além de seu objeto `IntegrationEvent` relacionado. O código, então, adiciona o manipulador de eventos à lista de manipuladores de eventos que cada tipo de evento de integração pode ter por microsserviço do cliente. Se o código do cliente ainda não tiver assinado o evento, o código criará um canal para o tipo de evento para que ele possa receber eventos em um estilo de push do RabbitMQ quando esse evento for publicado de qualquer outro serviço.

Conforme mencionado acima, o barramento de evento implementado em `eShopOnContainers` tem apenas e finalidade `Education`, já que trata apenas dos principais cenários, portanto, ele não está pronto para produção.

Para cenários de produção, verifique os recursos adicionais abaixo, específicos para RabbitMQ e a seção [implementando a comunicação baseada em evento entre os microserviços](#).

## Recursos adicionais

Uma solução pronta para produção com suporte para RabbitMQ.

- **EasyNetQ** -cliente de API .net de código aberto para RabbitMQ  
<https://easynetq.com/>
- **MassTransit**  
<https://masstransit-project.com/>

# Assinando eventos

08/04/2020 • 33 minutes to read • [Edit Online](#)

A primeira etapa para usar o barramento de eventos é fazer com que os microsserviços assinem os eventos que eles desejam receber. Isso deve ser feito nos microsserviços destinatários.

O código simples a seguir mostra o que cada destinatário de microsserviço precisa implementar ao iniciar o serviço (ou seja, a classe `Startup`) para assinar os eventos que precisa. Nesse caso, o microsserviço `basket-api` precisa assinar as mensagens `ProductPriceChangedIntegrationEvent` e `OrderStartedIntegrationEvent`.

Por exemplo, ao assinar `ProductPriceChangedIntegrationEvent` o evento, isso torna o microserviço da cesta cliente de qualquer alteração no preço do produto e permite que ele avise o usuário sobre a alteração se esse produto estiver na cesta do usuário.

```
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();

eventBus.Subscribe<ProductPriceChangedIntegrationEvent,
    ProductPriceChangedIntegrationEventHandler>();

eventBus.Subscribe<OrderStartedIntegrationEvent,
    OrderStartedIntegrationEventHandler>();
```

Depois que esse código for executado, o microsserviço assinante escutará por meio de canais RabbitMQ. Quando qualquer mensagem do tipo `ProductPriceChangedIntegrationEvent` chegar, o código invocará o manipulador de eventos, que será passado a ele e processará o evento.

## Publicando eventos por meio do barramento de eventos

Por fim, o remetente da mensagem (microsserviço de origem) publica os eventos de integração com um código semelhante ao exemplo a seguir. (Este é um exemplo simplificado que não leva em conta a atômicaidade.) Você implementaria código semelhante sempre que um evento deve ser propagado em vários microsserviços, geralmente logo após o cometimento de dados ou transações do microserviço de origem.

Primeiro, o objeto de implementação do barramento de eventos (baseado no RabbitMQ ou em um barramento de serviço) seria injetado no construtor do controlador, como no código a seguir:

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _context;
    private readonly IOptionsSnapshot<Settings> _settings;
    private readonly IEventBus _eventBus;

    public CatalogController(CatalogContext context,
        IOptionsSnapshot<Settings> settings,
        IEventBus eventBus)
    {
        _context = context;
        _settings = settings;
        _eventBus = eventBus;
    }
    // ...
}
```

Em seguida, você usá-lo a partir dos métodos do seu controlador, como no método `UpdateProduct`:

```
[Route("items")]
[HttpPost]
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem product)
{
    var item = await _context.CatalogItems.SingleOrDefaultAsync(
        i => i.Id == product.Id);
    // ...
    if (item.Price != product.Price)
    {
        var oldPrice = item.Price;
        item.Price = product.Price;
        _context.CatalogItems.Update(item);
        var @event = new ProductPriceChangedIntegrationEvent(item.Id,
            item.Price,
            oldPrice);
        // Commit changes in original transaction
        await _context.SaveChangesAsync();
        // Publish integration event to the event bus
        // (RabbitMQ or a service bus underneath)
        _eventBus.Publish(@event);
        // ...
    }
    // ...
}
```

Nesse caso, como o microsserviço de origem é um microsserviço CRUD simples, o código é colocado diretamente em um controlador de API Web.

Em microsserviços mais avançados, como ao usar abordagens de CQRS, ele pode ser implementado na classe `CommandHandler`, dentro do método `Handle()`.

### Criando atomicidade e resiliência ao publicar no barramento de eventos

Ao publicar eventos de integração por meio de um sistema de mensagens distribuído como o barramento de eventos, surge o problema da atualização do banco de dados original e da publicação de um evento de forma atômica (ou seja, ambas as operações são concluídas ou nenhuma delas). Por exemplo, no exemplo simplificado mostrado anteriormente, o código confirma os dados no banco de dados quando o preço do produto é alterado e, em seguida, publica uma mensagem `ProductPriceChangedIntegrationEvent`. Inicialmente, pode parecer essencial que essas duas operações sejam executadas atomicamente. No entanto, se você estivesse usando uma transação distribuída que envolvesse o banco de dados e o agente de mensagens, assim como faria em sistemas mais antigos, como o [MSMQ \(Enfileiramento de Mensagens da Microsoft\)](#), isso não seria recomendado pelos motivos descritos pelo [Teorema CAP](#).

Basicamente, você usa microsserviços para criar sistemas escalonáveis e altamente disponíveis. Simplificando um pouco, o teorema do CAP diz que você não pode construir um banco de dados (distribuído) (ou um microsserviço que possua seu modelo) que esteja continuamente disponível, fortemente consistente e tolerante a qualquer partição. Você deve escolher duas dessas três propriedades.

Em arquiteturas baseadas em microsserviços, você deve escolher disponibilidade e tolerância, e você deve desenfatizar uma forte consistência. Portanto, na maioria dos aplicativos modernos baseados em microsserviços, geralmente não é interessante usar transações distribuídas no sistema de mensagens, como ao implementar [transações distribuídas](#) com base no DTC (Coordenador de Transações Distribuídas) do Windows com o [MSMQ](#).

Vamos voltar à questão inicial e ao seu exemplo. Se o serviço falhar após a atualização do banco de dados `_context.SaveChangesAsync()` (neste caso, logo após a linha de código com ), mas antes do evento de integração ser publicado, o sistema global pode se tornar inconsistente. Isso pode ser crítico, dependendo da operação de negócios específica com a qual você está lidando.

Como mencionado anteriormente na seção de arquitetura, você pode ter várias abordagens para lidar com esse problema:

- Usando o padrão [Event Sourcing](#) completo.
- Usando [mineração do log de transações](#).
- Usando o [padrão Outbox](#). Essa é uma tabela transacional para armazenar os eventos de integração (estendendo a transação local).

Neste cenário, o uso do padrão ES (Event Sourcing) completo é uma das melhores abordagens, se não for a melhor. No entanto, em muitos cenários de aplicativo, pode ser impossível implementar um sistema completo de ES. O ES significa o armazenamento somente dos eventos de domínio em seu banco de dados transacional, em vez de armazenar dados do estado atual. Armazenar apenas os eventos de domínio pode ter grandes benefícios, como ter o histórico do sistema disponível e poder determinar o estado do sistema em qualquer momento no passado. No entanto, implementar um sistema de ES completo exige que você refaça a arquitetura da maior parte do seu sistema e também pode apresentar muitas outras complexidades e requisitos. Por exemplo, vai ser interessante usar um banco de dados criado especificamente para fornecimento de eventos, como o [Event Store](#), ou um banco de dados orientado a documentos, como Azure Cosmos DB, MongoDB, Cassandra, CouchDB ou RavenDB. O ES é uma ótima abordagem para esse problema, mas não é a solução mais fácil, a menos que você já esteja familiarizado com fornecimento de eventos.

A opção de usar a mineração de log de transações inicialmente parece transparente. No entanto, para usar essa abordagem, o microserviço deverá ser acoplado ao seu log de transações do RDBMS, como o log de transações do SQL Server. Isso provavelmente não é interessante. Outra desvantagem é que as atualizações de baixo nível registradas no log de transações podem não estar no mesmo nível que seus eventos de integração de alto nível. Nesse caso, o processo de engenharia reversa dessas operações do log de transações poderá ser difícil.

Uma abordagem equilibrada é uma combinação de uma tabela de banco de dados transacional e um padrão de ES simplificado. Você pode usar um estado como "pronto para publicar o evento", que você define no evento original quando o compromete na tabela de eventos de integração. Em seguida, você tenta publicar o evento no barramento de eventos. Se a ação de publicação-evento for bem sucedida, você inicia outra transação no serviço de origem e move o estado de "pronto para publicar o evento" para "evento já publicado".

Se a ação de evento de publicação no ônibus de caso falhar, os dados ainda não serão inconsistentes dentro do microserviço de origem — ele ainda está marcado como "pronto para publicar o evento", e com relação ao resto dos serviços, ele eventualmente será consistente. Você sempre fazer com que trabalhos em segundo plano verifiquem o estado das transações ou dos eventos de integração. Se o trabalho encontrar um evento no estado "pronto para publicar o evento", ele pode tentar republicar esse evento para o ônibus do evento.

Observe que, com essa abordagem, você estará persistindo apenas os eventos de integração de cada microserviço de origem e somente os eventos que você deseja comunicar para outros microserviços ou sistemas externos. Por outro lado, em um sistema completo de ES, você também armazena todos os eventos de domínio.

Portanto, essa abordagem equilibrada é um sistema de ES simplificado. Você precisa de uma lista de eventos de integração com seu estado atual ("pronto para publicar" versus "publicado"). Mas você só precisa implementar esses estados para os eventos de integração. E, nessa abordagem, não é necessário armazenar todos os dados de domínio como eventos no banco de dados transacional, como você faria em um sistema completo de ES.

Se já está usando um banco de dados relacional, você pode usar uma tabela transacional para armazenar eventos de integração. Para obter a atomicidade em seu aplicativo, você usa um processo de duas etapas com base nas transações locais. Basicamente, você tem uma tabela `IntegrationEvent` no mesmo banco de dados em que estão as entidades de domínio. Essa tabela funciona como um seguro para a obtenção de atomicidade, para que você inclua eventos de integração persistidos nas mesmas transações que estão confirmado seus dados de domínio.

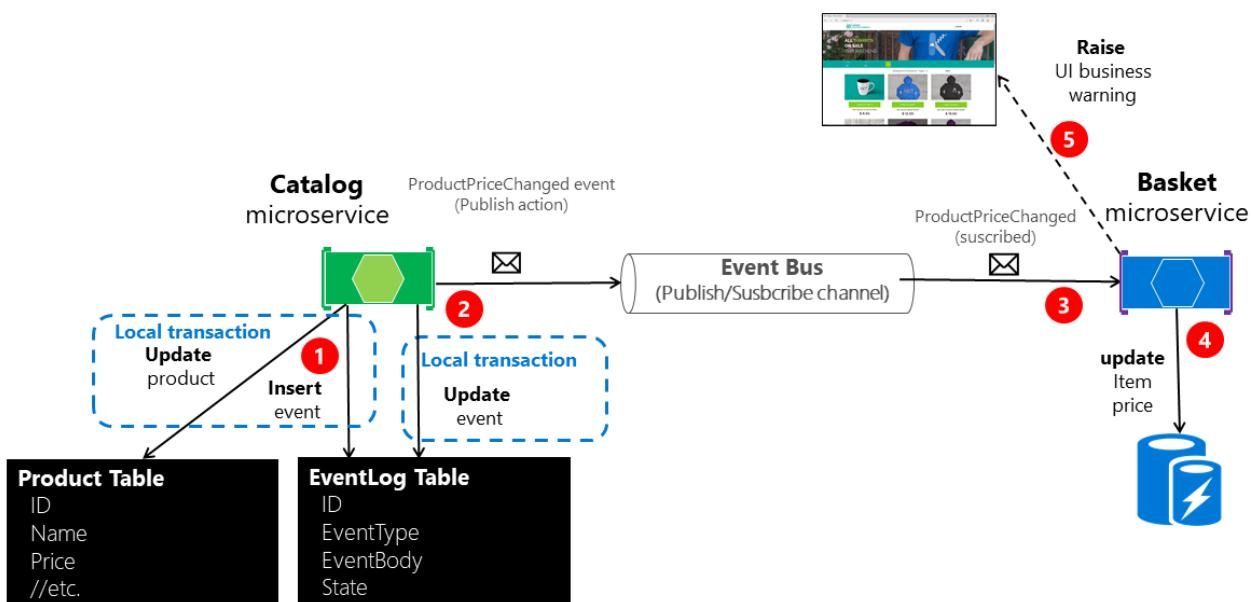
Passo a passo, o processo é o seguinte:

1. O aplicativo inicia uma transação de banco de dados local.
2. Em seguida, ele atualiza o estado das suas entidades de domínio e insere um evento na tabela de eventos de integração.
3. Por fim, ele confirma a transação, de modo que você obtenha a atomicidade desejada e
4. Você publica o evento de alguma forma (a seguir).

Ao implementar as etapas de publicação dos eventos, você tem estas opções:

- Publicar o evento de integração logo após a confirmação da transação e usar outra transação local para marcar os eventos na tabela como sendo publicados. Em seguida, usar a tabela apenas como um artefato para controlar os eventos de integração, em caso de problemas com os microsserviços remotos, e executar ações compensatórias com base nos eventos de integração armazenados.
- Usar a tabela como um tipo de fila. Um thread ou processo do aplicativo separado consulta a tabela de eventos de integração, publica os eventos no barramento de eventos e, em seguida, usa uma transação local para marcar os eventos como publicados.

A Figura 6-22 mostra a arquitetura da primeira dessas abordagens.



**Figura 6-22.** Atomicidade ao publicar eventos no barramento de eventos

A abordagem ilustrada na Figura 6-22 não tem um microsserviço de trabalho adicional, que é responsável por verificar e confirmar o êxito dos eventos de integração publicados. Em caso de falha, esse microsserviço de trabalho verificador adicional pode ler os eventos na tabela e publicá-los novamente, ou seja, repetir a etapa 2.

Em relação à segunda abordagem: você usa a tabela EventLog como uma fila e sempre usa um microsserviço de trabalho para publicar as mensagens. Nesse caso, o processo é como o mostrado na Figura 6-23. Ela mostra um microsserviço adicional, e a tabela é a única fonte durante a publicação de eventos.

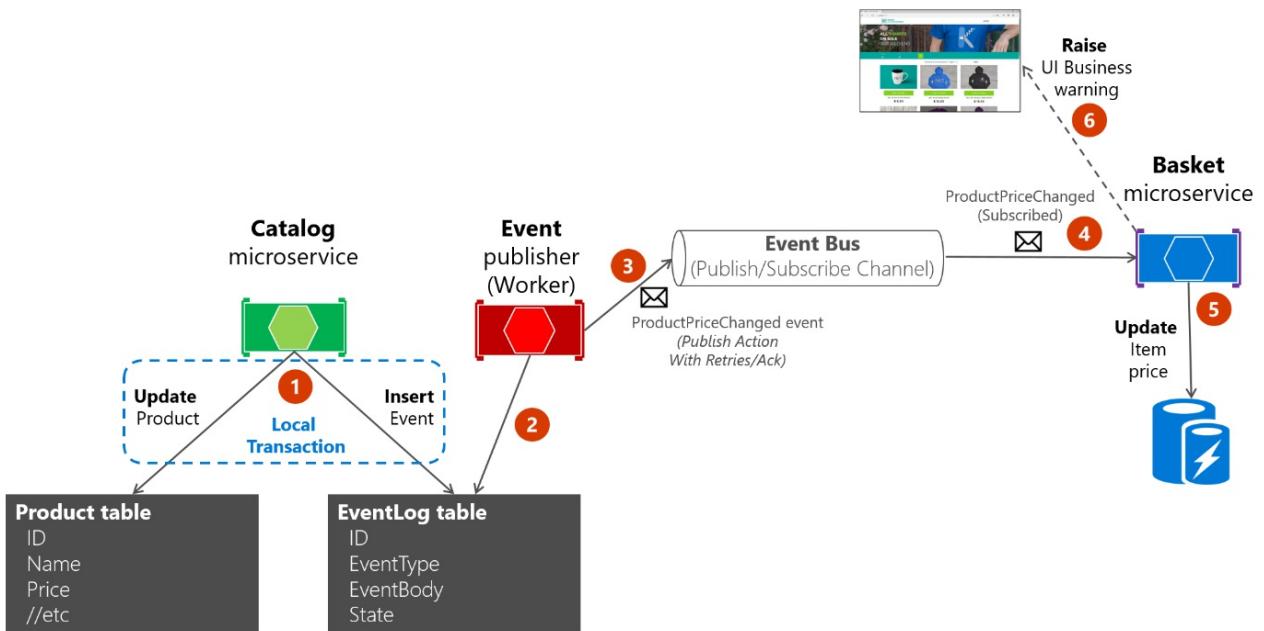


Figura 6-23. Atomicidade ao publicar eventos no barramento de eventos com um microsserviço de trabalho

Para simplificar, o exemplo eShopOnContainers usa a primeira abordagem (sem processos adicionais nem microsserviços verificadores) e o barramento de eventos. No entanto, a amostra eShopOnContainers não está lidando com todos os casos de falha possíveis. Em um aplicativo real implantado na nuvem, você deve aceitar o fato de que os problemas surgirão eventualmente, e você deverá implementar essa lógica de verificação e reenvio. O uso da tabela como uma fila pode ser mais eficiente do que a primeira abordagem se você tem essa tabela como uma única fonte de eventos ao publicá-los (com o trabalho) por meio do barramento de eventos.

### Implementando atomicidade ao publicar eventos de integração por meio do barramento de eventos

O código a seguir mostra como criar uma única transação que envolva vários objetos DbContext: um contexto relacionados aos dados originais que estão sendo atualizados e o segundo contexto relacionado à tabela IntegrationEventLog.

A transação no código de exemplo abaixo não será resiliente se as conexões com o banco de dados tiverem algum problema no momento em que o código estiver em execução. Isso pode ocorrer em sistemas baseados em nuvem, como o BD SQL do Azure, que pode mover bancos de dados entre servidores. Para implementar transações resilientes em vários contextos, consulte a seção [Implementando conexões de SQL do Entity Framework Core resilientes](#) mais adiante, neste guia.

Para maior clareza, o exemplo a seguir mostra todo o processo em um único segmento de código. No entanto, a implementação do eShopOnContainers é refatorada e divide essa lógica em várias classes para que seja mais fácil de manter.

```

// Update Product from the Catalog microservice
//
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem productToUpdate)
{
    var catalogItem =
        await _catalogContext.CatalogItems.SingleOrDefaultAsync(i => i.Id ==
            productToUpdate.Id);

    if (catalogItem == null) return NotFound();

    bool raiseProductPriceChangedEvent = false;
    IntegrationEvent priceChangedEvent = null;

    if (catalogItem.Price != productToUpdate.Price)
        raiseProductPriceChangedEvent = true;

    if (raiseProductPriceChangedEvent) // Create event if price has changed
    {
        var oldPrice = catalogItem.Price;
        priceChangedEvent = new ProductPriceChangedIntegrationEvent(catalogItem.Id,
            productToUpdate.Price,
            oldPrice);
    }

    // Update current product
    catalogItem = productToUpdate;

    // Just save the updated product if the Product's Price hasn't changed.
    if (!raiseProductPriceChangedEvent)
    {
        await _catalogContext.SaveChangesAsync();
    }
    else // Publish to event bus only if product price changed
    {
        // Achieving atomicity between original DB and the IntegrationEventLog
        // with a local transaction
        using (var transaction = _catalogContext.Database.BeginTransaction())
        {
            _catalogContext.CatalogItems.Update(catalogItem);
            await _catalogContext.SaveChangesAsync();

            // Save to EventLog only if product price changed
            if(raiseProductPriceChangedEvent)
                await _integrationEventLogService.SaveEventAsync(priceChangedEvent);

            transaction.Commit();
        }

        // Publish the integration event through the event bus
        _eventBus.Publish(priceChangedEvent);

        _integrationEventLogService.MarkEventAsPublishedAsync(
            priceChangedEvent);
    }
}

return Ok();
}

```

Depois que o evento de integração `ProductPriceChangedIntegrationEvent` é criado, a transação que armazena a operação de domínio original (atualizar o item de catálogo) também inclui a persistência do evento na tabela `EventLog`. Isso faz com que ele se torne uma única transação, e sempre será possível verificar se as mensagens de evento foram enviadas.

A tabela do log de eventos é atualizada atomicamente com a operação do banco de dados original, usando uma transação local no mesmo banco de dados. Se uma das operações falha, uma exceção é gerada e a transação reverte qualquer operação concluída, mantendo a consistência entre as operações de domínio e as mensagens de

evento salvas na tabela.

## Recebendo mensagens de assinaturas: manipuladores de eventos em microsserviços destinatários

Além da lógica de assinatura de evento, você precisa implementar o código interno para os manipuladores de eventos de integração (como um método de retorno de chamada). O manipulador de eventos é o local em que você especifica o local em que as mensagens de eventos de um determinado tipo serão recebidas e processadas.

Primeiro, um manipulador de eventos recebe uma instância de evento do barramento de eventos. Em seguida, ele localiza o componente a ser processado, relacionado a esse evento de integração, propagando e persistindo o evento como uma alteração no estado no microsserviço destinatário. Por exemplo, se um evento ProductPriceChanged se origina no microsserviço catálogo, ele é tratado no microsserviço carrinho de compras e também altera o estado nesse microsserviço destinatário de carrinho de compras, conforme mostrado no código a seguir.

```
namespace Microsoft.eShopOnContainers.Services.Basket.API.IntegrationEvents.EventHandling
{
    public class ProductPriceChangedIntegrationEventHandler :
        IIntegrationEventHandler<ProductPriceChangedIntegrationEvent>
    {
        private readonly IBasketRepository _repository;

        public ProductPriceChangedIntegrationEventHandler(
            IBasketRepository repository)
        {
            _repository = repository;
        }

        public async Task Handle(ProductPriceChangedIntegrationEvent @event)
        {
            var userIds = await _repository.GetUsers();
            foreach (var id in userIds)
            {
                var basket = await _repository.GetBasket(id);
                await UpdatePriceInBasketItems(@event.ProductId, @event.NewPrice, basket);
            }
        }

        private async Task UpdatePriceInBasketItems(int productId, decimal newPrice,
            CustomerBasket basket)
        {
            var itemsToUpdate = basket?.Items?.Where(x => int.Parse(x.ProductId) ==
                productId).ToList();
            if (itemsToUpdate != null)
            {
                foreach (var item in itemsToUpdate)
                {
                    if(item.UnitPrice != newPrice)
                    {
                        var originalPrice = item.UnitPrice;
                        item.UnitPrice = newPrice;
                        item.OldUnitPrice = originalPrice;
                    }
                }
                await _repository.UpdateBasket(basket);
            }
        }
    }
}
```

O manipulador de eventos deve verificar se o produto existe em qualquer uma das instâncias do carrinho de compras. Ele também atualiza o preço do item para cada item de linha relacionado ao carrinho. Por fim, ele cria um alerta sobre a alteração de preço, que será exibido para o usuário, conforme mostrado na Figura 6-24.

The screenshot shows a shopping cart page from 'eShop onCONTAINERS'. The cart contains three items:

- .NET Black & White Mug**: Price \$8.50, Quantity 1, Cost \$8.50.
- .NET Blue Hoodie**: Price \$21.00, Quantity 1, Cost \$21.00. This item has a red box around its price and quantity input field.
- .NET Bot Black Hoodie**: Price \$19.50, Quantity 1, Cost \$19.50.

A yellow callout box with a red arrow points to the .NET Blue Hoodie row, containing the text: "Note that the price of this article changed in our Catalog. The old price when you originally added it to the basket was 12.00 \$".

At the bottom right of the cart area, there are two buttons: [ UPDATE ] and [ CHECKOUT ].

The footer of the page includes the eShop onContainers logo and the text "e-ShoponContainers. All right reserved".

Figura 6-24. Exibindo uma alteração de preço de item em um carrinho de compras, conforme comunicado pelos eventos de integração

## Idempotência em eventos de mensagem de atualização

Um aspecto importante dos eventos de mensagem de atualização é que uma falha em qualquer ponto da comunicação deverá fazer com que a mensagem seja repetida. Caso contrário, uma tarefa em segundo plano poderá tentar publicar um evento que já tenha sido publicado, criando uma condição de corrida. Certifique-se de que as atualizações são idempotentes ou que fornecem informações suficientes para garantir que você possa detectar uma duplicata, descartá-la e enviar de volta apenas uma resposta.

Conforme observado anteriormente, a idempotência significa que uma operação poderá ser executada várias vezes sem alterar o resultado. Em um ambiente de sistema de mensagens, como ao comunicar eventos, um evento será idempotente se ele puder ser entregue várias vezes sem alterar o resultado para o microsserviço destinatário. Isso pode ser necessário devido à natureza do evento em si ou devido à maneira como o sistema manipula o evento. A idempotência de mensagem é importante em qualquer aplicativo que use o sistema de mensagens, e não apenas em aplicativos que implementam o padrão do barramento de eventos.

Um exemplo de uma operação idempotente é uma instrução SQL que insere dados em uma tabela somente se os dados ainda não estiverem na tabela. Não importa quantas vezes você executa essa instrução insert do SQL; o resultado será o mesmo: a tabela conterá esses dados. Essa idempotência também poderá ser necessária ao lidar com mensagens se houver chances de que elas sejam enviadas e processadas mais de uma vez. Por exemplo, se a lógica de repetição fizer com que um remetente envie exatamente a mesma mensagem mais de uma vez, você

precisará verificar se ela é idempotente.

É possível projetar mensagens idempotentes. Por exemplo, crie um evento que indica "definir o preço do produto como US\$ 25" em vez de "adicionar US\$ 5 ao preço do produto". A primeira mensagem pode ser processada com segurança qualquer número de vezes e o resultado será o mesmo. Isso não é válido para a segunda mensagem. Mas, mesmo no primeiro caso, talvez não seja interessante processar o primeiro evento, porque o sistema também pode ter enviado um evento de alteração de preço mais recente, e o novo preço seria substituído.

Outro exemplo pode ser um evento concluído por ordem que é propagado para vários assinantes. O aplicativo tem que garantir que as informações do pedido sejam atualizadas em outros sistemas apenas uma vez, mesmo que haja eventos de mensagens duplicadas para o mesmo evento concluído por ordem.

É conveniente ter algum tipo de identidade por evento, para que você possa criar uma lógica que imponha que cada evento seja processado apenas uma vez em cada destinatário.

Alguns processamentos de mensagens são inherentemente idempotentes. Por exemplo, se um sistema gerar miniaturas de imagem, não importará quantas vezes a mensagem sobre a miniatura gerada vai ser processada; o resultado será que as miniaturas foram geradas e elas serão sempre as mesmas. Por outro lado, operações como chamar um gateway de pagamento para cobrar um cartão de crédito jamais poderão ser idempotentes. Nesses casos, você precisa garantir que o processamento de uma mensagem várias vezes tenha o efeito que você espera.

## Recursos adicionais

- Honrando a idempotência da mensagem

[/previous-versions/msp-n-p/jj591565\(v=pandp.10\)#honoring-message-idempotency](/previous-versions/msp-n-p/jj591565(v=pandp.10)#honoring-message-idempotency)

## Eliminando a duplicação de mensagens de eventos de integração

Você pode garantir que os eventos de mensagem sejam enviados e processados apenas uma vez por assinante em diferentes níveis. Uma maneira é usar um recurso de eliminação de duplicação oferecido pela infraestrutura do sistema de mensagens que você está usando. Outra é implementar uma lógica personalizada no microserviço de destino. A melhor opção é ter validações no nível de transporte e no nível do aplicativo.

### Eliminando a duplicação de eventos de mensagem no nível do EventHandler

Uma maneira de garantir que um evento seja processado apenas uma vez por qualquer receptor é implementando certa lógica ao processar os eventos de mensagem em manipuladores de eventos. Por exemplo, essa é a abordagem usada no aplicativo eShopOnContainers, como você pode ver no [código-fonte da classe UserCheckoutAcceptedIntegrationEventHandler](#) quando recebe um evento de

`UserCheckoutAcceptedIntegrationEvent` integração. (Neste caso, `CreateOrderCommand` o é `IdentifiedCommand` embrulhado com `eventMsg.RequestID` um , usando o como um identificador, antes de enviá-lo para o manipulador de comando).

### Eliminando a duplicação de mensagens ao usar RabbitMQ

Quando ocorrem falhas intermitentes na rede, as mensagens podem ser duplicadas e o destinatário da mensagem deve estar preparado para lidar com essas mensagens duplicadas. Sempre que possível, os destinatários devem lidar com as mensagens de maneira idempotente, pois isso é melhor que tratá-las explicitamente com a eliminação de duplicação.

De acordo com a documentação do [RabbitMQ](#), "Se uma mensagem for entregue a um consumidor e depois for remanerada (porque não foi reconhecida antes da conexão com o consumidor cair, por exemplo) então a RabbitMQ definirá a bandeira reentregue quando for entregue novamente (seja para o mesmo consumidor ou outra).

Se o sinalizador "reentregue" estiver definido, o receptor deve levar isso em conta, porque a mensagem já pode ter sido processada. Mas isso não é garantido; a mensagem pode nunca ter alcançado o destinatário depois de deixar o agente de mensagens, talvez por causa de problemas de rede. Por outro lado, se a bandeira "reentregue" não for

definida, é garantido que a mensagem não tenha sido enviada mais de uma vez. Portanto, o receptor precisa deduplicar mensagens ou processar mensagens de forma idempotente somente se a bandeira "reentregue" for definida na mensagem.

## Recursos adicionais

- EShopOnContainers forked usando NServiceBus (Software Particular)  
<https://go.particular.net/eShopOnContainers>
- Mensagens orientadas a eventos  
[https://patterns.arcitura.com/soa-patterns/design\\_patterns/event\\_driven.messaging](https://patterns.arcitura.com/soa-patterns/design_patterns/event_driven.messaging)
- Jimmy Bogard. Refatoração para a resiliência: Avaliando o acoplamento  
<https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/>
- Publique-Inscreva-se canal  
<https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- Comunicação entre contextos limitados  
[/previous-versions/msp-n-p/jj591572\(v=pandp.10\)](/previous-versions/msp-n-p/jj591572(v=pandp.10))
- Consistência eventual  
[https://en.wikipedia.org/wiki/Eventual\\_consistency](https://en.wikipedia.org/wiki/Eventual_consistency)
- Philip Brown. Estratégias para integrar contextos limitados  
<https://www.culttt.com/2014/11/26/strategies-integrating-bounded-contexts/>
- Chris Richardson. Desenvolvimento de Microserviços Transacionais Utilizando Agregados, Sourcing de Eventos e CQRS - Parte 2  
<https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson>
- Chris Richardson. Padrão de Sourcing de Eventos  
<https://microservices.io/patterns/data/event-sourcing.html>
- Introdução ao Sourcing de Eventos  
[/previous-versions/msp-n-p/jj591559\(v=pandp.10\)](/previous-versions/msp-n-p/jj591559(v=pandp.10))
- Banco de dados Event Store. Site oficial.  
<https://geteventstore.com/>
- Patrick Nommensen. Gerenciamento de dados orientado a eventos para microserviços  
<https://dzone.com/articles/event-driven-data-management-for-microservices-1>
- Teorema do CAP  
[https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem)
- O que é o Teorema de CAP?  
<https://www.quora.com/What-Is-CAP-Theorem-1>
- Primer de consistência de dados  
[/previous-versions/msp-n-p/dn589800\(v=pandp.10\)](/previous-versions/msp-n-p/dn589800(v=pandp.10))
- Rick Saling. O Teorema do CAP: Por que "Tudo é Diferente" com a Nuvem e internet  
[/archive/blogs/rickatmicrosoft/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/](https://archive/blogs/rickatmicrosoft/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/)
- Eric Brewer. CAP Doze anos depois: como as "regras" mudaram  
<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>
- Ônibus de Serviço Azure. Mensagens Intermediadas: Detecção de duplicatas  
<https://code.msdn.microsoft.com/Brokered-Messaging-c0acea25>

- Guia de Confiabilidade (documentação do RabbitMQ)

<https://www.rabbitmq.com/reliability.html#consumer>

PRÓXIMO

ANTERIOR

# Testar serviços e aplicativos Web do ASP.NET Core

10/09/2020 • 13 minutes to read • [Edit Online](#)

Os controladores são uma parte central de qualquer serviço de API do ASP.NET Core e aplicativo Web ASP.NET MVC. Assim, você precisa estar confiante de que eles se comportarão conforme o esperado para o aplicativo. Testes automatizados podem fornecer essa confiança e detectar erros antes que eles atinjam a produção.

É necessário testar como o controlador se comporta com base em entradas válidas ou inválidas, bem como testar suas respostas com base nos resultados da operação de negócios realizada. No entanto, você deveria ter esses tipos de testes nos seus microsserviços:

- Testes de unidade. Eles garantem que os componentes individuais do aplicativo funcionam como esperado. Instruções assert testam a API do componente.
- Testes de integração. Garantem que as interações do componente funcionam como esperado em artefatos externos, como bancos de dados. Instruções assert podem testar a API do componente, a interface do usuário ou os efeitos colaterais de ações, como E/S de banco de dados, registros em log etc.
- Testes funcionais para cada microsserviço. Eles garantem que o aplicativo funcione conforme o esperado da perspectiva do usuário.
- Testes de serviço. Garantem que os casos de uso do serviço de ponta a ponta sejam testados, incluindo testes de vários serviços ao mesmo tempo. Para esse tipo de teste, é necessário preparar o ambiente primeiro. Nesse caso, significa iniciar os serviços (por exemplo, usando o docker-compose up).

## Implementar testes de unidade para APIs Web do ASP.NET Core

O teste de unidade envolve o teste de uma parte de um aplicativo em isolamento de sua infraestrutura e suas dependências. Ao executar a lógica do controlador de teste de unidade, somente o conteúdo de uma única ação ou método é testado, e não o comportamento de suas dependências ou da estrutura. Os testes de unidade não detectam problemas na interação entre componentes, o que é a finalidade dos testes de integração.

Ao executar o teste de unidade nas ações do controlador, concentre-se apenas em seu comportamento. Um teste de unidade de controlador evita itens como filtros, roteamento ou model binding (o mapeamento dos dados de solicitação a um ViewModel ou DTO). Ao se concentrarem no teste de apenas um item, os testes de unidade geralmente são simples de serem gravados e rápidos de serem executados. Um conjunto bem escrito de testes de unidade pode ser executado com frequência sem muita sobrecarga.

Testes de unidade são implementados com base em estruturas de teste como xUnit.net, MSTest, Moq ou NUnit. Para o aplicativo de exemplo eShopOnContainers, estamos usando o xUnit.

Ao gravar um teste de unidade de um controlador de API Web, você cria instâncias da classe do controlador usando diretamente a nova palavra-chave no C#, para que o teste seja executado o mais rápido possível. O exemplo a seguir mostra como fazer isso usando o [xUnit](#) como a estrutura de teste.

```

[Fact]
public async Task Get_order_detail_success()
{
    //Arrange
    var fakeOrderId = "12";
    var fakeOrder = GetFakeOrder();

    //...

    //Act
    var orderController = new OrderController(
        _orderServiceMock.Object,
        _basketServiceMock.Object,
        _identityParserMock.Object);

    orderController.ControllerContext.HttpContext = _contextMock.Object;
    var actionResult = await orderController.Detail(fakeOrderId);

    //Assert
    var viewResult = Assert.IsType<ViewResult>(actionResult);
    Assert.IsAssignableFrom<Order>(viewResult.ViewData.Model);
}

```

### **Implementar testes funcionais e de integração em cada microsserviço**

Conforme observado, testes funcionais e de integração têm finalidades e objetivos diferentes. No entanto, a maneira de implementar ambos ao testar os controladores do ASP.NET Core é semelhante. Nesta seção, o foco é o teste de integração.

Testes de integração garantem que os componentes de um aplicativo funcionem corretamente quando montados. O ASP.NET Core é compatível com testes de integração por meio de estruturas de teste de unidade e um host Web de testes interno que pode ser utilizado para lidar com solicitações sem sobrecarga de rede.

Ao contrário do teste de unidade, os testes de integração frequentemente envolvem questões relacionadas com a infraestrutura do aplicativo, como banco de dados, sistema de arquivos, recurso de rede ou solicitações e respostas da Web. Testes de unidade usam objetos falsos ou fictícios em vez disso. Porém, a finalidade dos testes de integração é confirmar que o sistema funciona como o esperado com esses sistemas. Então, nos testes de integração não se usam objetos falsos ou fictícios. Em vez disso, você inclui a infraestrutura, como acesso ao banco de dados ou invocação de serviço de outros serviços.

Como os testes de integração exercitam segmentos de código maiores que os testes de unidade e contam com elementos de infraestrutura, eles tendem a ter ordens de magnitude mais lentas que os testes de unidade. Portanto, é uma boa ideia limitar a quantidade de testes de integração gravados e executados.

O ASP.NET Core inclui um host Web de teste interno que pode ser usado para lidar com solicitações HTTP sem sobrecarga de rede, o que significa que você pode executar esses testes mais rápido do que ao usar um host Web real. O host Web de testes (`TestServer`) está disponível em um componente NuGet como `Microsoft.AspNetCore.TestHost`. Ele pode ser adicionado a projetos de teste de integração e usados para hospedar aplicativos do ASP.NET Core.

Conforme mostrado no código a seguir, ao criar testes de integração para controladores do ASP.NET Core, você cria instâncias para os controladores por meio do host de teste. Isso é equivalente a uma solicitação HTTP, mas ele é executado mais rapidamente.

```

public class PrimeWebDefaultRequestShould
{
    private readonly TestServer _server;
    private readonly HttpClient _client;

    public PrimeWebDefaultRequestShould()
    {
        // Arrange
        _server = new TestServer(new WebHostBuilder()
            .UseStartup<Startup>());
        _client = _server.CreateClient();
    }

    [Fact]
    public async Task ReturnHelloWorld()
    {
        // Act
        var response = await _client.GetAsync("/");
        response.EnsureSuccessStatusCode();
        var responseString = await response.Content.ReadAsStringAsync();
        // Assert
        Assert.Equal("Hello World!", responseString);
    }
}

```

#### Recursos adicionais

- **Steve Smith.** [Testando controladores](#) (ASP.NET Core)  
<https://docs.microsoft.com/aspnet/core/mvc/controllers/testing>
- **Steve Smith.** [Testes de integração](#) (ASP.NET Core)  
<https://docs.microsoft.com/aspnet/core/test/integration-tests>
- [Teste de unidade no .NET Core usando o teste dotnet](#)  
<https://docs.microsoft.com/dotnet/core/testing/unit-testing-with-dotnet-test>
- [xUnit.net](#). Site oficial.  
<https://xunit.github.io/>
- [Noções básicas do teste de unidade.](#)  
<https://docs.microsoft.com/visualstudio/test/unit-test-basics>
- [Moq](#). Repositório do GitHub.  
<https://github.com/moq/moq>
- [NUnit](#). Site oficial.  
<https://www.nunit.org/>

#### Implementar testes de serviço em um aplicativo com vários contêineres

Conforme mencionado anteriormente, ao testar aplicativos com vários contêineres, todos os microsserviços precisam ser executados no host do Docker ou no cluster do contêiner. Testes de serviço de ponta a ponta que incluem várias operações envolvendo diversos microsserviços exigem a implantação e inicialização do aplicativo inteiro no host do Docker por meio da execução do docker-compose up (ou um mecanismo semelhante se você estiver usando um orquestrador). Quando o aplicativo inteiro e todos os seus serviços estiverem em execução, você pode executar testes funcionais e de integração de ponta a ponta.

Existem algumas abordagens que você pode usar. No arquivo docker-compose.yml usado para implantar o aplicativo, é possível expandir o ponto de entrada no nível da solução para usar o [dotnet test](#). Também é possível usar outro arquivo do Compose para executar testes na imagem de destino. Ao utilizar outro arquivo do Compose para testes de integração que inclui microsserviços e bancos de dados em contêineres, você garante que dados relacionados sempre sejam redefinidos para seu estado original antes de executar os testes.

Se o Visual Studio estiver em execução, será possível aproveitar pontos de interrupção e exceções após o aplicativo do Compose entrar em funcionamento. Outra opção é executar os testes de integração automaticamente no pipeline de CI no Azure DevOps Services ou em qualquer outro sistema CI/CD compatível com contêineres do Docker.

## Testando em eShopOnContainers

Os testes do aplicativo (eShopOnContainers) de referência foram recentemente restrukturados e agora há quatro categorias:

1. **Unidade** testa, os antigos testes de unidade regulares, contidos nos projetos `{MicroserviceName}.UnitTests`
2. **Testes funcionais/integração de microsserviço**, com casos de teste que envolvem a infraestrutura para cada microsserviço, mas isolado dos outros; estão contidos nos projetos `{MicroserviceName}.FunctionalTests`.
3. **Testes funcionais/de integração do aplicativo**, que se concentram na integração de microsserviços, com casos de teste que exercem vários microsserviços. Esses testes estão localizados no projeto `Application.FunctionalTests`.

Os testes de unidade e de integração por microsserviço estão contidos em uma pasta de teste em cada microsserviço e o testes de carga e de aplicativo estão contidos na pasta teste da pasta da solução, conforme mostrado na Figura 6-25.

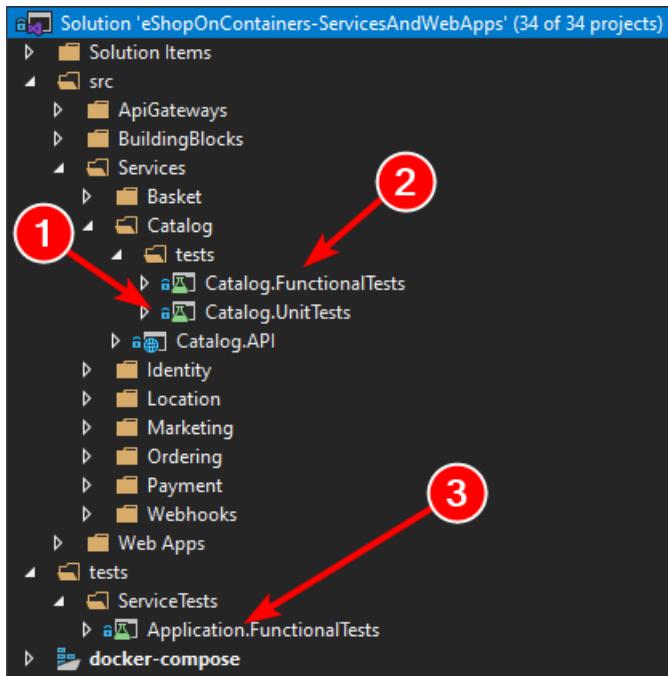


Figura 6-25. Estrutura da pastas de teste em eShopOnContainers

Testes funcionais/integração de aplicativos e microsserviços são executados no Visual Studio, usando o executor de testes regular, mas primeiro você precisa iniciar os serviços de infraestrutura necessários, por meio de um conjunto de arquivos docker-compose contidos na pasta de teste da solução:

`docker-compose-test.yml`

```
version: '3.4'

services:
  redis.data:
    image: redis:alpine
  rabbitmq:
    image: rabbitmq:3-management-alpine
  sqldata:
    image: mcr.microsoft.com/mssql/server:2017-latest
  nosqldata:
    image: mongo
```

### docker-compose-test.override.yml

```
version: '3.4'

services:
  redis.data:
    ports:
      - "6379:6379"
  rabbitmq:
    ports:
      - "15672:15672"
      - "5672:5672"
  sqldata:
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
  nosqldata:
    ports:
      - "27017:27017"
```

Portanto, para executar os testes funcionais/integração, primeiro você deve executar esse comando na pasta de teste da solução:

```
docker-compose -f docker-compose-test.yml -f docker-compose-test.override.yml up
```

Como você pode ver, esses arquivos docker-compose só iniciam os microsserviços Redis, RabbitMQ, SQL Server e MongoDB.

### Recursos adicionais

- **Teste de integração de & de unidade no eShopOnContainers**  
<https://github.com/dotnet-architecture/eShopOnContainers/wiki/Unit-and-integration-testing>
- **Teste de carga no eShopOnContainers**

[ANTERIOR](#)

[AVANÇAR](#)

# Implementar tarefas em segundo plano em microserviços com `IHostedService` e a classe `BackgroundService`

10/09/2020 • 15 minutes to read • [Edit Online](#)

Tarefas em segundo plano e trabalhos agendados são algo que talvez você precise usar em qualquer aplicativo, seja ou não após o padrão de arquitetura de microserviços. A diferença ao usar uma arquitetura de microserviços é que você pode implementar a tarefa em segundo plano em um processo/contêiner separado para hospedagem, de modo que você possa dimensioná-la verticalmente com base em sua necessidade.

De um ponto de vista genérico, no .NET Core, chamamos esses tipos de tarefas de *Serviços Hospedados*, porque são serviços/lógica que você hospeda em seu host/aplicativo/microserviço. Observe que, neste caso, o serviço hospedado simplesmente significa uma classe com a lógica de tarefa em segundo plano.

Desde o .NET Core 2.0, a estrutura fornece uma nova interface chamada `IHostedService`, ajudando você a implementar serviços hospedados facilmente. A ideia básica é que você pode registrar várias tarefas em segundo plano (serviços hospedados) que são executadas em segundo plano enquanto o host ou o servidor Web está em execução, conforme mostrado na imagem 6-26.

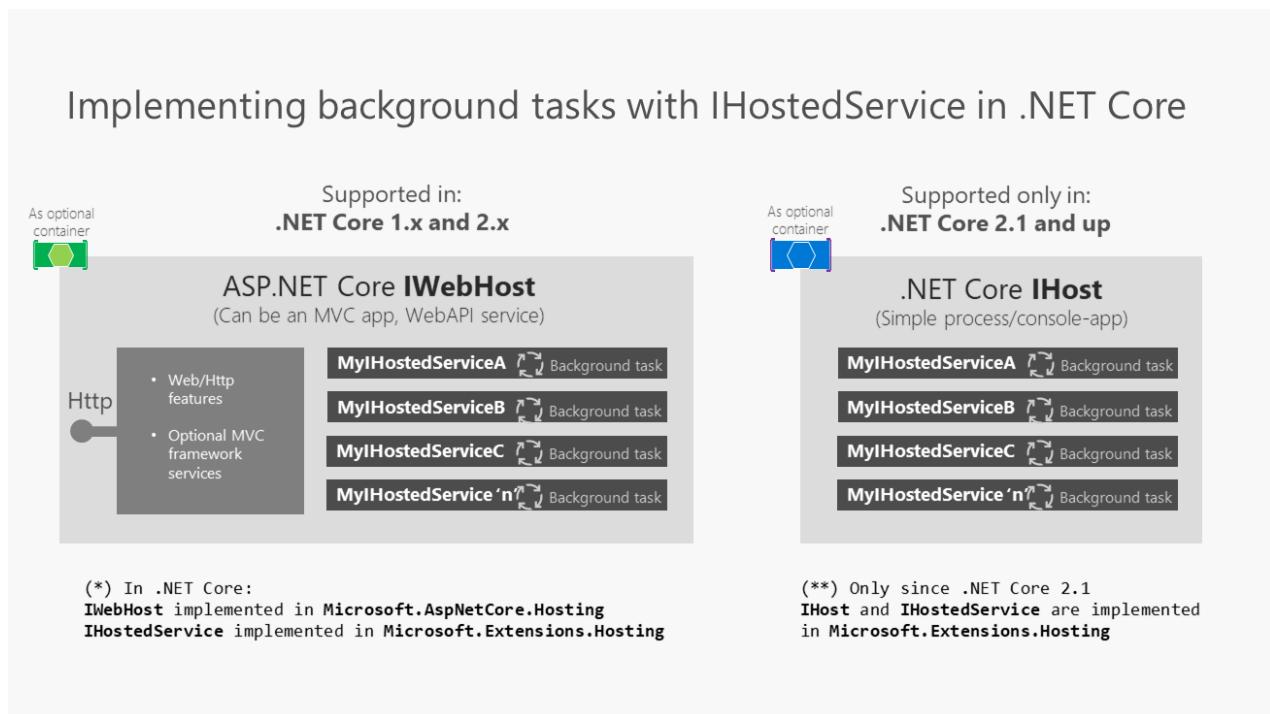


Figura 6-26. Usando `IHostedService` em `WebHost` versus um `Host`

Suporte a ASP.NET Core 1.x e 2.x `IWebHost` para processos em segundo plano em aplicativos Web. O .NET Core 2,1 e versões posteriores dão suporte a `IHost` processos em segundo plano com aplicativos de console simples. Observe a diferença entre `WebHost` e `Host`.

R `WebHost` (classe base implementando `IWebHost`) no ASP.NET Core 2,0 é o artefato de infraestrutura que você usa para fornecer recursos de servidor http para seu processo, como quando você está implementando um aplicativo Web MVC ou serviço de API Web. Ele fornece toda a nova qualidade de infraestrutura no ASP.NET Core, permitindo que você use a injeção de dependência, insira middleware no pipeline de solicitação e semelhante. O `WebHost` usa esses mesmos `IHostedServices` para tarefas em segundo plano.

Um `Host` (classe base que implementa `IHost`) foi introduzido no .NET Core 2.1. Basicamente, um `Host` permite que você tenha uma infraestrutura semelhante àquela que você tem com `WebHost` (injeção de dependência, serviços hospedados etc.), mas, nesse caso, você quer apenas ter um processo simples e mais leve como o host, sem nada relacionado ao MVC, à API da Web ou aos recursos de servidor HTTP.

Portanto, você pode escolher e criar um processo de host especializado com `IHost` o para lidar com os serviços hospedados e nada mais, como um microserviço feito apenas para hospedar o `IHostedServices`, ou então você pode estender uma ASP.NET Core existente `WebHost`, como uma API Web existente ASP.NET Core ou um aplicativo MVC.

Cada abordagem tem vantagens e desvantagens, dependendo de suas necessidades de negócios e escalabilidade. A linha inferior é basicamente que se as tarefas em segundo plano não têm nada a ver com HTTP (`IWebHost`), você deve usar `IHost`.

## Registro de serviços hospedados em seu `WebHost` ou `Host`

Vamos detalhar mais detalhadamente a `IHostedService` interface, pois seu uso é bastante semelhante em um `WebHost` ou em um `Host`.

SignalR é um exemplo de um artefato usando serviços hospedados, mas você também pode usá-lo para itens muito mais simples, como:

- Uma tarefa em segundo plano sondando um banco de dados em busca de alterações.
- Uma tarefa agendada atualizando algum cache periodicamente.
- Uma implementação de `QueueBackgroundWorkItem` que permite que uma tarefa seja executada em um thread em segundo plano.
- Processamento de mensagens de uma fila de mensagens em segundo plano de um aplicativo Web enquanto se compartilham serviços comuns como `ILogger`.
- Uma tarefa em segundo plano iniciada com `Task.Run()`.

Basicamente, você pode descarregar qualquer uma dessas ações em uma tarefa em segundo plano que implementa o `IHostedService`.

A maneira como você adiciona um ou vários `IHostedServices` em seu `WebHost` ou `Host` está registrando-os por meio do `AddHostedService` método de extensão em um ASP.NET Core `WebHost` (ou em um `Host` no .NET Core 2,1 e posterior). Basicamente, você deve registrar os serviços hospedados no método `ConfigureServices()` familiar da classe `Startup`, como no código a seguir de um `WebHost` ASP.NET típico.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    //Other DI registrations;

    // Register Hosted Services
    services.AddHostedService<GracePeriodManagerService>();
    services.AddHostedService<MyHostedServiceB>();
    services.AddHostedService<MyHostedServiceC>();
    //...
}
```

Nesse código, o serviço hospedado `GracePeriodManagerService` é o código real do microsserviço de negócios de Ordenação em eShopOnContainers, enquanto os outros dois são apenas dois exemplos adicionais.

A execução da tarefa em segundo plano `IHostedService` é coordenada com o tempo de vida do aplicativo (ou seja, host ou microsserviço). Você registra tarefas quando o aplicativo é iniciado e você tem a oportunidade de fazer alguma ação normal ou limpeza quando o aplicativo está sendo desligado.

Sem usar `IHostedService`, você sempre pode iniciar um thread em segundo plano para executar qualquer tarefa.

A diferença é precisamente no tempo de desligamento do aplicativo quando esse thread simplesmente fosse encerrado sem a oportunidade de executar ações de limpeza normais.

## A interface `IHostedService`

Quando você registra um `IHostedService`, o .NET Core chamará os métodos `StartAsync()` e `StopAsync()` de seu tipo `IHostedService` durante o início e a parada do aplicativo, respectivamente. Para obter mais detalhes, consulte a [interface `IHostedService`](#)

Como você pode imaginar, é possível criar várias implementações de `IHostedService` e registrá-las no método `ConfigureService()` no contêiner de ID, como mostrado anteriormente. Todos esses serviços hospedados serão iniciados e interrompidos junto com o aplicativo/microsserviço.

Como desenvolvedor, você é responsável por lidar com a ação de parada de seus serviços quando o método `StopAsync()` é disparado pelo host.

## Implementando `IHostedService` com uma classe de serviço hospedado personalizado derivado da classe base `BackgroundService`

Vá em frente e crie sua classe de serviço hospedado personalizada do zero e implemente o `IHostedService`, como você precisa fazer ao usar o .NET Core 2.0.

No entanto, como a maioria das tarefas em segundo plano terá necessidades semelhantes em relação ao gerenciamento de tokens de cancelamento e outras operações típicas, há uma classe base abstrata conveniente da qual você pode derivar, chamada `BackgroundService` (disponível no .NET Core 2.1).

Essa classe fornece o trabalho principal necessário para configurar a tarefa em segundo plano.

O próximo código é a classe base abstrata `BackgroundService`, conforme implementada no .NET Core.

```

// Copyright (c) .NET Foundation. Licensed under the Apache License, Version 2.0.
/// <summary>
/// Base class for implementing a long running <see cref="IHostedService"/>.
/// </summary>
public abstract class BackgroundService : IHostedService, IDisposable
{
    private Task _executingTask;
    private readonly CancellationTokenSource _stoppingCts =
        new CancellationTokenSource();

    protected abstract Task ExecuteAsync(CancellationToken stoppingToken);

    public virtual Task StartAsync(CancellationToken cancellationToken)
    {
        // Store the task we're executing
        _executingTask = ExecuteAsync(_stoppingCts.Token);

        // If the task is completed then return it,
        // this will bubble cancellation and failure to the caller
        if (_executingTask.IsCompleted)
        {
            return _executingTask;
        }

        // Otherwise it's running
        return Task.CompletedTask;
    }

    public virtual async Task StopAsync(CancellationToken cancellationToken)
    {
        // Stop called without start
        if (_executingTask == null)
        {
            return;
        }

        try
        {
            // Signal cancellation to the executing method
            _stoppingCts.Cancel();
        }
        finally
        {
            // Wait until the task completes or the stop token triggers
            await Task.WhenAny(_executingTask, Task.Delay(Timeout.Infinite,
                cancellationToken));
        }
    }

    public virtual void Dispose()
    {
        _stoppingCts.Cancel();
    }
}

```

Ao derivar da classe base abstrata anterior, graças àquela implementação herdada, você só precisa implementar o método `ExecuteAsync()` em sua própria classe de serviço hospedado personalizada, como no seguinte código simplificado eShopOnContainers que está sondando um banco de dados e publicando eventos de integração no Barramento de Eventos quando necessário.

```

public class GracePeriodManagerService : BackgroundService
{
    private readonly ILogger<GracePeriodManagerService> _logger;
    private readonly OrderingBackgroundSettings _settings;

    private readonly IEventBus _eventBus;

    public GracePeriodManagerService(IOptions<OrderingBackgroundSettings> settings,
                                    IEventBus eventBus,
                                    ILogger<GracePeriodManagerService> logger)
    {
        // Constructor's parameters validations...
    }

    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogDebug($"GracePeriodManagerService is starting.");

        stoppingToken.Register(() =>
            _logger.LogDebug($" GracePeriod background task is stopping."));

        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogDebug($"GracePeriod task doing background work.");

            // This eShopOnContainers method is querying a database table
            // and publishing events into the Event Bus (RabbitMQ / ServiceBus)
            CheckConfirmedGracePeriodOrders();

            await Task.Delay(_settings.CheckUpdateTime, stoppingToken);
        }

        _logger.LogDebug($"GracePeriod background task is stopping.");
    }

    .../...
}

```

Neste caso específico para eShopOnContainers, que está executando um método de aplicativo que está consultando uma tabela de banco de dados procurando pedidos com um estado específico e ao aplicar as alterações, ele está publicando eventos de integração por meio de um barramento de evento (sob ele, pode estar usando RabbitMQ ou Barramento de Serviço do Azure).

Obviamente, você pode executar qualquer outra tarefa em segundo plano de negócios em vez disso.

Por padrão, o token de cancelamento é definido com um tempo limite de 5 segundos, embora você possa alterar esse valor ao compilar o `WebHost` usando a `UseShutdownTimeout` extensão do `IWebHostBuilder`. Isso significa que o nosso serviço deve cancelar dentro de 5 segundos, caso contrário, ele será encerrado mais abruptamente.

O código a seguir alteraria esse tempo para 10 segundos.

```

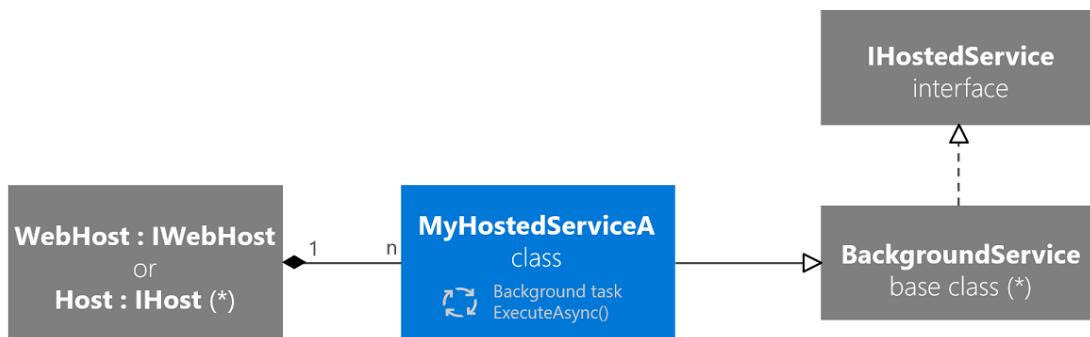
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...

```

## Diagrama de classe de resumo

A imagem a seguir mostra um resumo visual das classes e interfaces envolvidas ao implementar `IHostedServices`.

## Class diagram with a custom `IHostedService` and related classes and interfaces



(\*\*) `IHost` and `BackgroundService` are implemented in `Microsoft.Extensions.Hosting` only since .NET Core 2.1

Figura 6-27. Diagrama de classe mostrando as várias classes e interfaces relacionadas ao `IHostedService`

Diagrama de classe: o `IWebHost` e o `IHost` podem hospedar muitos serviços, herdados de `BackgroundService`, que implementa o `IHostedService`.

### Pontos importantes e considerações sobre implantação

É importante observar que a maneira como você implanta o ASP.NET Core `WebHost` ou .NET Core `Host` pode afetar a solução final. Por exemplo, se você implantar o `WebHost` no IIS ou em um Serviço de Aplicativo do Azure normal, o host poderá desligar devido a reciclagens do pool de aplicativos. Mas se você estiver implantando o host como um contêiner em um orquestrador como kubernetes, poderá controlar o número garantido de instâncias ao vivo do seu host. Além disso, poderá considerar outras abordagens na nuvem feitas especialmente para esses cenários, como Azure Functions. Por fim, se você precisar que o serviço esteja em execução o tempo todo e estiver implantando em um Windows Server, você poderá usar um Serviço Windows.

Mas mesmo para uma `WebHost` implantação em um pool de aplicativos, há cenários como repopular ou liberar o cache na memória do aplicativo que ainda seria aplicável.

A `IHostedService` interface fornece uma maneira conveniente de iniciar tarefas em segundo plano em um aplicativo web ASP.NET Core (no .net core 2,0 e em versões posteriores) ou em qualquer processo/host (a partir do .net core 2,1 com `IHost`). Seu principal benefício é a oportunidade que você obtém com o cancelamento normal para limpar o código das tarefas em segundo plano quando o próprio host está sendo desligado.

## Recursos adicionais

- Criando uma tarefa agendada no ASP.NET Core/Standard 2,0  
<https://blog.maartenballiauw.be/post/2017/08/01/building-a-scheduled-cache-updater-in-aspnet-core-2.html>
- Implementando `IHostedService` no ASP.NET Core 2,0  
<https://www.stevejgordon.co.uk/asp-net-core-2-ihostedservice>
- Exemplo de `GenericHost` usando o ASP.NET Core 2,1  
<https://github.com/aspnet/Hosting/tree/release/2.1/samples/GenericHostSample>

# Implementar Gateways de API com o Ocelot

10/09/2020 • 39 minutes to read • [Edit Online](#)

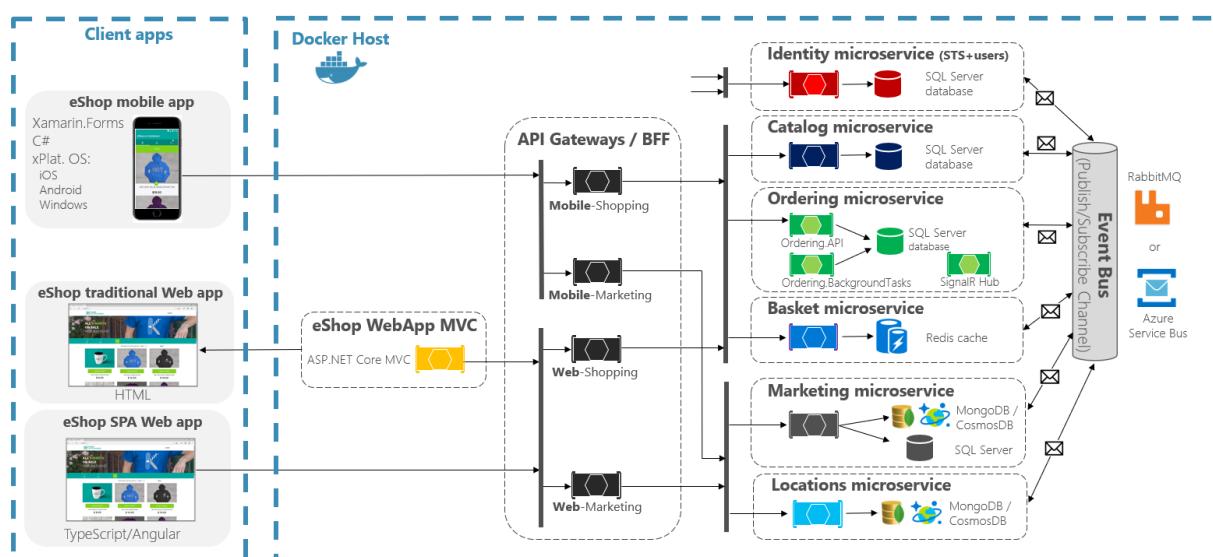
## IMPORTANT

O aplicativo de desserviço de referência [eShopOnContainers](#) está usando atualmente os recursos fornecidos pelo [Envoy](#) para implementar o gateway de API em vez do [Ocelot](#) referenciado anterior. Fizemos essa escolha de design devido ao suporte interno do Envoy para o protocolo WebSocket, exigido pelas novas comunicações entre serviços gRPC implementadas no eShopOnContainers. No entanto, guardamos esta seção no guia para que você possa considerar o Ocelot como um gateway de API simples, compatível e leve, adequado para cenários de nível de produção. Além disso, a versão mais recente do Ocelot contém uma alteração significativa em seu esquema JSON. Considere usar o Ocelot < v 16.0.0 ou usar as rotas de chave em vez de redirecionar.

## Arquitetar e projetar seus Gateways de API

O diagrama de arquitetura a seguir mostra como os gateways de API foram implementados com Ocelot em eShopOnContainers.

**eShopOnContainers reference application**  
(Development environment architecture)



**Figura 6-28.** Arquitetura do eShopOnContainers com Gateways de API

Esse diagrama mostra como todo o aplicativo é implantado em um único host do Docker ou em um PC de desenvolvimento com "Docker for Windows" ou "Docker para Mac". No entanto, a implantação em qualquer orquestrador seria semelhante, mas qualquer container no diagrama poderia ser escalado horizontalmente no orquestrador.

Além disso, os ativos de infraestrutura, como bancos de dados, cache e agentes de mensagens devem ser descarregados do orquestrador e implantados em sistemas altamente disponíveis para a infraestrutura, como o Banco de Dados SQL do Azure, o Azure Cosmos DB, o Redis do Azure, o Barramento de Serviço do Azure ou qualquer solução de cluster de HA local.

Como você também pode observar no diagrama, ter vários gateways de API permite que várias equipes de desenvolvimento sejam autônomas (neste caso, recursos de marketing versus recursos de compra) ao

desenvolver e implantar seus microserviços, além de seus próprios gateways de API relacionados.

Se você tivesse um único Gateway de API monolítico, isso significaria um único ponto a ser atualizado por várias equipes de desenvolvimento, o que poderia vincular todos os microsserviços com uma única parte do aplicativo.

Avançado bastante no design, às vezes, um Gateway de API refinado também pode ser limitado a um único microsserviço de negócios, dependendo da arquitetura escolhida. Ter os limites do gateway de API ditados pelo negócio ou pelo domínio ajudará você a obter um design melhor.

Por exemplo, a granularidade refinada na camada do Gateway de API pode ser útil principalmente para aplicativos de interface do usuário compostos, mais avançados e baseados em microsserviços, porque o conceito de Gateway de API refinado é semelhante ao de serviço de composição de interface do usuário.

Apresentamos mais detalhes na seção anterior [Criando uma interface do usuário composta baseada em microsserviços](#).

Como principal vantagem para muitos aplicativos de médio e grande porte, o uso de um produto de Gateway de API personalizado costuma ser uma boa abordagem, mas não como um único agregador monolítico ou um Gateway de API personalizado central exclusivo, a menos que esse Gateway de API permita várias áreas de configuração independentes para as diversas equipes de desenvolvimento que criam microsserviços autônomos.

#### **Microsserviços/contêineres de amostra a serem reencaminhados por meio dos Gateways de API**

Por exemplo, o eShopOnContainers tem cerca de seis tipos de microsserviços internos que precisam ser publicados por meio dos Gateways de API, conforme mostrado na imagem a seguir.

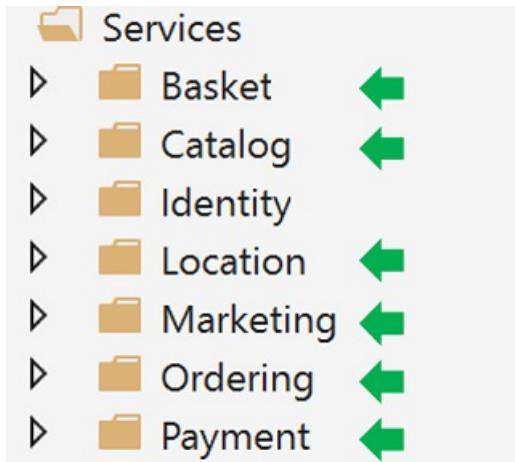
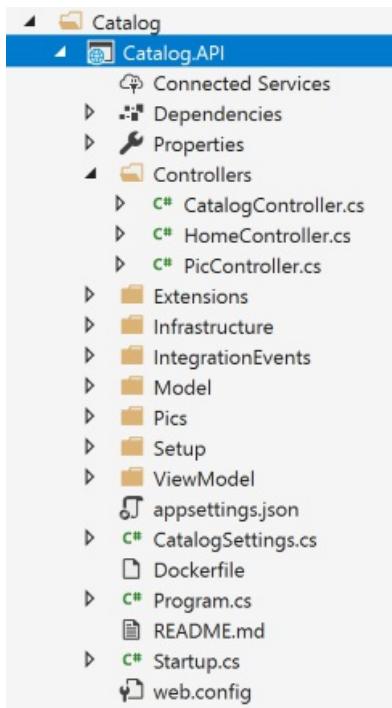


Figura 6-29. Pastas do microsserviço na solução eShopOnContainers no Visual Studio

Sobre o serviço de Identidade, no design, ele fica fora do roteamento do Gateway de API porque é o único interesse paralelo no sistema, embora com o Ocelot também seja possível incluí-lo como parte das listas de reencaminhamento.

Todos esses serviços são implementados atualmente como serviços de API Web do ASP.NET Core, como você pode observar no código. Vamos nos concentrar em um dos microsserviços como o código de microsserviço de catálogo.



**Figura 6-30.** Microsserviço de API Web de exemplo (microsserviço Catálogo)

Você pode ver que o microsserviço Catálogo é um projeto de API Web ASP.NET Core típico com vários controladores e métodos como no código a seguir.

```
[HttpGet]
[Route("items/{id:int}")]
[ProducesResponseType((int) HttpStatusCode.BadRequest)]
[ProducesResponseType((int) HttpStatusCode.NotFound)]
[ProducesResponseType(typeof(CatalogItem), (int) HttpStatusCode.OK)]
public async Task<IActionResult> GetItemById(int id)
{
    if (id <= 0)
    {
        return BadRequest();
    }
    var item = await _catalogContext.CatalogItems.
        SingleOrDefaultAsync(ci => ci.Id == id);
    //...

    if (item != null)
    {
        return Ok(item);
    }
    return NotFound();
}
```

A solicitação HTTP acabará executando esse tipo de código C# acessando o banco de dados de microsserviço e qualquer ação adicional necessária.

Em relação à URL do Microservice, quando os contêineres são implantados em seu PC de desenvolvimento local (host do Docker local), cada contêiner de microsserviço sempre tem uma porta interna (geralmente a porta 80) especificada em seu dockerfile, como no seguinte dockerfile:

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1 AS base
WORKDIR /app
EXPOSE 80
```

A porta 80 mostrada no código é interna no host do Docker e, portanto, não pode ser acessada por aplicativos

cliente.

Os aplicativos cliente poderão acessar apenas as portas externas (se houver) publicadas durante a implantação com `docker-compose`.

Essas portas externas não devem ser publicadas durante a implantação em um ambiente de produção. Esse é exatamente o motivo para usar o Gateway de API, ou seja, para evitar a comunicação direta entre os aplicativos clientes e os microsserviços.

No entanto, durante o desenvolvimento, é necessário acessar o microsserviço/contêiner diretamente e executá-lo por meio do Swagger. É por isso que, em eShopOnContainers, as portas externas ainda são especificadas mesmo quando não serão usadas pelo gateway de API ou pelos aplicativos cliente.

Aqui está um exemplo do `docker-compose.override.yml` arquivo para o microsserviço de catálogo:

```
catalog-api:  
  environment:  
    - ASPNETCORE_ENVIRONMENT=Development  
    - ASPNETCORE_URLS=http://0.0.0.0:80  
    - ConnectionString=YOUR_VALUE  
    - ... Other Environment Variables  
  ports:  
    - "5101:80"  # Important: In a production environment you should remove the external port (5101) kept  
here for microservice debugging purposes.  
          # The API Gateway redirects and access through the internal port (80).
```

Você pode ver como na configuração do `docker-compose.override.yml` a porta interna do contêiner de catálogo é a porta 80, mas a porta para acesso externo é a 5101. Mas essa porta não deve ser usada pelo aplicativo ao usar um gateway de API, somente para depurar, executar e testar apenas o microsserviço de catálogo.

Normalmente, você não estará implantando com Docker-Compose em um ambiente de produção porque o ambiente de implantação de produção certo para microsserviços é um orquestrador como Kubernetes ou Service Fabric. Ao implantar nesses ambientes, você usa arquivos de configuração diferentes em que você não publicará diretamente nenhuma porta externa para os microsserviços, mas, sempre usará o proxy reverso do gateway de API.

Execute o microsserviço de catálogo no host do Docker local. Execute a solução eShopOnContainers completa do Visual Studio (ela executa todos os serviços nos arquivos do Docker-Compose) ou inicie o microsserviço de catálogo com o comando Docker-compote a seguir no CMD ou no PowerShell posicionado na pasta em que o `docker-compose.yml` e `docker-compose.override.yml` são colocados.

```
docker-compose run --service-ports catalog-api
```

Esse comando só executa o contêiner do serviço Catalog-API, além das dependências especificadas no Docker-Compose. yml. Nesse caso, o contêiner do SQL Server e o contêiner do RabbitMQ.

Em seguida, você pode acessar diretamente o microsserviço de catálogo e ver seus métodos por meio da interface do usuário do Swagger acessando diretamente por meio dessa porta "externa", neste caso

`http://localhost:5101/swagger` :

**Figura 6-31.** Testando o microsserviço Catálogo com sua interface do usuário do Swagger

Neste ponto, você pode definir um ponto de interrupção no código C# no Visual Studio, testar o microsserviço com os métodos expostos na interface do usuário do Swagger e, por fim, limpar tudo com o comando `docker-compose down`.

No entanto, a comunicação de acesso direto com o microsserviço, nesse caso, pela porta 5101 externa, é exatamente o que você deseja evitar em seu aplicativo. E você pode evitar isso definindo o nível adicional de indireção do Gateway de API (o Ocelot, neste caso). Dessa forma, o aplicativo cliente não acessará diretamente o Microservice.

## Implementando Gateways de API com o Ocelot

O Ocelot é basicamente um conjunto de middleware que você pode aplicar em uma ordem específica.

O Ocelot foi projetado para funcionar apenas com o ASP.NET Core. A versão mais recente do pacote é direcionada para `.NETCoreApp 3.1` e, portanto, não é adequada para aplicativos .NET Framework.

Instale o Ocelot e suas dependências no projeto ASP.NET Core com o [pacote NuGet do Ocelot](#), por meio do Visual Studio.

```
Install-Package Ocelot
```

No eShopOnContainers, sua implementação de gateway de API é um projeto Webhost simples ASP.NET Core, e o

middleware da Ocelot lida com todos os recursos do gateway de API, conforme mostrado na imagem a seguir:

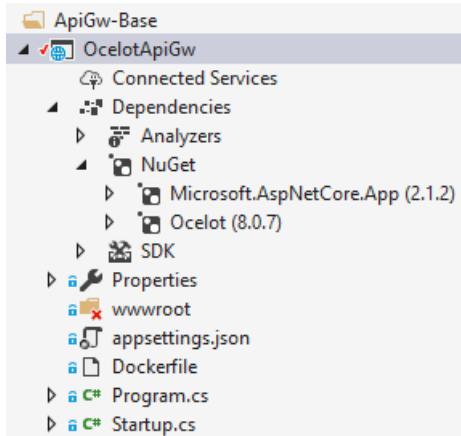


Figura 6-32. Projeto base OcelotApiGw no eShopOnContainers

Basicamente, este projeto WebHost do ASP.NET Core é criado com dois arquivos simples: `Program.cs` e `Startup.cs`.

O `Program.cs` precisa apenas criar e configurar o `BuildWebHost` típico do ASP.NET Core.

```
namespace OcelotApiGw
{
    public class Program
    {
        public static void Main(string[] args)
        {
            BuildWebHost(args).Run();
        }

        public static IWebHost BuildWebHost(string[] args)
        {
            var builder = WebHost.CreateDefaultBuilder(args);

            builder.ConfigureServices(s => s.AddSingleton(builder))
                .ConfigureAppConfiguration(
                    ic => ic.AddJsonFile(Path.Combine("configuration",
                        "configuration.json")))
                .UseStartup<Startup>();

            var host = builder.Build();
            return host;
        }
    }
}
```

O ponto importante para Ocelot é o arquivo `configuration.json` que você precisa fornecer para o construtor pelo método `AddJsonFile()`. Esse `configuration.json` é onde você especifica todos os reencaminhamentos do Gateway de API, o que significa os pontos de extremidade externos com portas específicas e os pontos de extremidade internos correlacionados, geralmente usando portas diferentes.

```
{
    "ReRoutes": [],
    "GlobalConfiguration": {}
}
```

Há duas seções para a configuração. Uma matriz de redirecionamentos e um `GlobalConfiguration`. As rotas são os objetos que dizem ao Ocelot como tratar uma solicitação upstream. A configuração global permite substituições de redirecionar configurações específicas. É útil se você não quiser gerenciar muitas configurações específicas de redirecionamento.

Aqui está um exemplo simplificado de [redirecionar o arquivo de configuração](#) de um dos gateways de API do eShopOnContainers.

```
{
  "ReRoutes": [
    {
      "DownstreamPathTemplate": "/api/{version}/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "catalog-api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/c/{everything}",
      "UpstreamHttpMethod": [ "POST", "PUT", "GET" ]
    },
    {
      "DownstreamPathTemplate": "/api/{version}/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "basket-api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/b/{everything}",
      "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "AllowedScopes": []
      }
    }
  ],
  "GlobalConfiguration": {
    "RequestIdKey": "OcRequestId",
    "AdministrationPath": "/administration"
  }
}
```

A funcionalidade principal de um Gateway de API Ocelot é obter solicitações HTTP de entrada e encaminhá-las para um serviço downstream, simultaneamente à outra solicitação HTTP. Ocelot descreve o roteamento de uma solicitação para outra como uma redirecionamento.

Por exemplo, vamos nos concentrar em uma das rotas na configuration.json acima, a configuração do microserviço basket.

```
{
    "DownstreamPathTemplate": "/api/{version}/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
        {
            "Host": "basket-api",
            "Port": 80
        }
    ],
    "UpstreamPathTemplate": "/api/{version}/b/{everything}",
    "UpstreamHttpMethod": [ "POST", "PUT", "GET" ],
    "AuthenticationOptions": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "AllowedScopes": []
    }
}
```

O `DownstreamPathTemplate`, o esquema e o `DownstreamHostAndPorts` compõem a URL interna do microserviço ao qual essa solicitação será encaminhada.

A porta é a porta interna usada pelo serviço. Ao usar contêineres, é a porta especificada no `dockerfile`.

O `Host` é um nome de serviço que depende da resolução de nomes de serviço que você está usando. Quando o `docker-compose` é usado, os serviços de nomes são fornecidos pelo Host do Docker, que está usando os nomes de serviço fornecidos nos arquivos `docker-compose`. Quando é usado um orquestrador, como o Kubernetes ou o Service Fabric, esse nome deve ser resolvido pelo DNS ou pela resolução de nomes fornecida por cada orquestrador.

`DownstreamHostAndPorts` é uma matriz que contém o host e a porta de todos os serviços downstream para os quais você deseja encaminhar solicitações. Normalmente, ela conterá apenas uma entrada, mas, às vezes, convém balancear a carga de solicitações para os serviços downstream e, para isso, o Ocelot permite que você adicione mais de uma entrada e, em seguida, selecione um平衡ador de carga. Mas se você estiver usando o Azure e algum orquestrador, provavelmente, será melhor balancear a carga com a infraestrutura de nuvem e do orquestrador.

O `UpstreamPathTemplate` é a URL que o Ocelot usará para identificar qual `DownstreamPathTemplate` será usado para uma determinada solicitação do cliente. Por fim, o `UpstreamHttpMethod` é usado para que Ocelot possa distinguir entre diferentes solicitações (GET, POST, PUT) para a mesma URL.

Neste ponto, você poderia usar um único Gateway de API do Ocelot (`WebHost` do ASP.NET Core), com um ou vários arquivos `configuration.json` mesclados, ou armazenar a [configuração em um repositório KV Consul](#).

Mas, conforme apresentado nas seções de arquitetura e design, se você realmente deseja usar microserviços autônomos, talvez seja melhor dividir esse único Gateway de API monolítico em vários Gateways de API e/ou BFFs (back-end para front-end). Para essa finalidade, vamos ver como implementar essa abordagem com contêineres do Docker.

## Usando uma única imagem de contêiner do Docker para executar o vários Gateway de API diferentes ou tipos de BFF contêineres

No eShopOnContainers, estamos usando uma única imagem de contêiner do Docker com o gateway de API do Ocelot, mas então, em tempo de execução, criamos diferentes serviços/contêineres para cada tipo de API-Gateway/BFF fornecendo um `configuration.json` diferente no arquivo, usando um volume do Docker para acessar uma pasta de computador diferente para cada serviço.

---

## Containers API Gateways / BFF

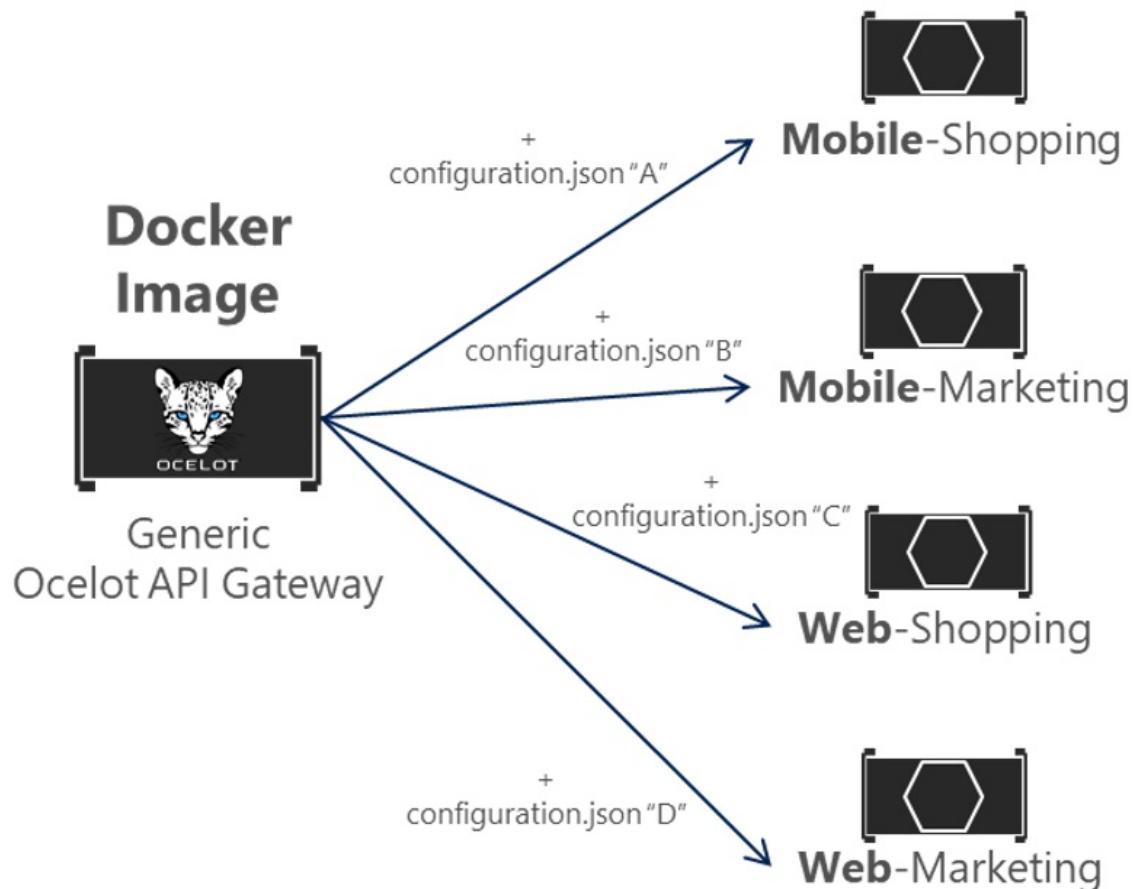


Figura 6-33. Reutilizando uma única imagem do Docker do Ocelot em vários tipos de Gateway de API

No eShopOnContainers, a "imagem do Docker do gateway de API Ocelot genérica" é criada com o projeto chamado 'OcelotApiGw' e o nome da imagem 'eshop/OcelotApiGw' que é especificado no arquivo Docker-Compose.yml. Em seguida, ao implantar no Docker, haverá quatro contêineres de Gateway de API criados com essa mesma imagem do Docker, conforme é mostrado na seguinte extração do arquivo docker-compose.yml.

```

mobileshoppingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

mobilemarketingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

webshoppingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

webmarketingapigw:
  image: eshop/ocelotapigw:${TAG:-latest}
  build:
    context: .
    dockerfile: src/ApiGateways/ApiGw-Base/Dockerfile

```

Além disso, como você pode ver no arquivo docker-compose.override.yml a seguir, a única diferença entre esses contêineres de Gateway de API é o arquivo de configuração do Ocelot, que é diferente para cada contêiner de serviço e é especificado em runtime por meio de um volume do Docker.

```

mobileshoppingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5200:80"
  volumes:
    - ./src/ApiGateways/Mobile.Bff.Shopping/apigw:/app/configuration

mobilemarketingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5201:80"
  volumes:
    - ./src/ApiGateways/Mobile.Bff.Marketeting/apigw:/app/configuration

webshoppingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5202:80"
  volumes:
    - ./src/ApiGateways/Web.Bff.Shopping/apigw:/app/configuration

webmarketingapigw:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - IdentityUrl=http://identity-api
  ports:
    - "5203:80"
  volumes:
    - ./src/ApiGateways/Web.Bff.Marketeting/apigw:/app/configuration

```

Devido ao código anterior e, como é mostrado no Gerenciador do Visual Studio abaixo, o único arquivo necessário para definir cada Gateway de API de negócios/BFF é apenas um arquivo configuration.json, porque os quatro Gateways de API são baseados na mesma imagem do Docker.

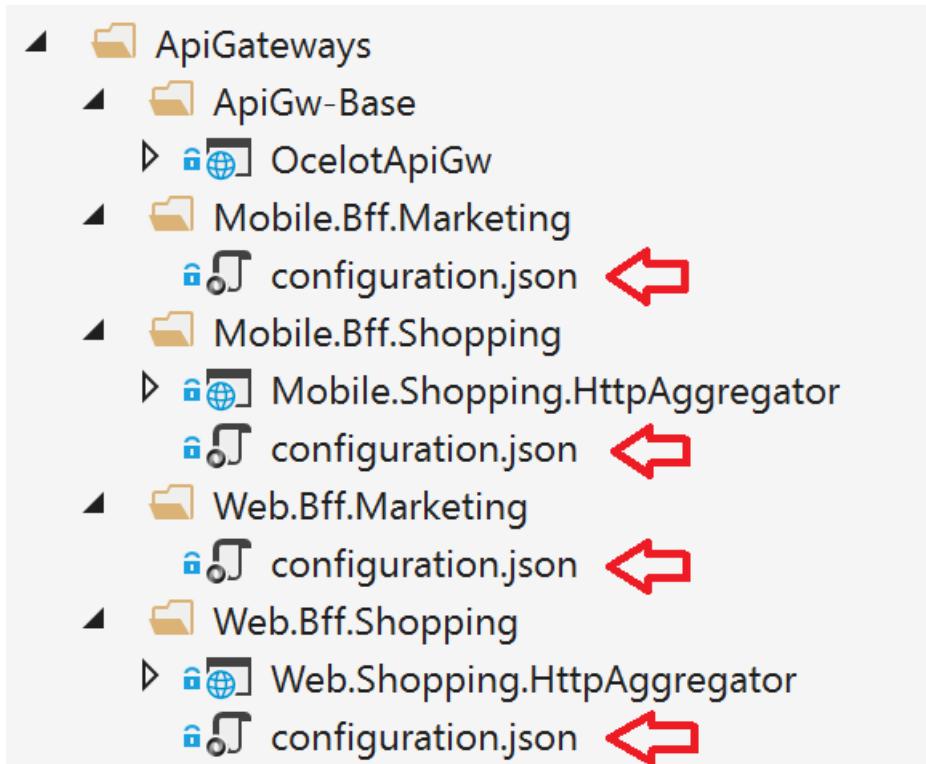


Figura 6-34. O único arquivo necessário para definir cada Gateway de API/BFF com o Ocelot é um arquivo de configuração

Dividindo o Gateway de API em vários Gateways de API, diferentes equipes de desenvolvimento concentrando-se em diferentes subconjuntos de microsserviços podem gerenciar seus próprios Gateways de API usando arquivos de configuração do Ocelot independentes. Além disso, ao mesmo tempo, elas podem reutilizar a mesma imagem do Docker do Ocelot.

Agora, se você executar o eShopOnContainers com os gateways de API (incluídos por padrão no VS ao abrir a solução eShopOnContainers-ServicesAndWebApps.sln ou se estiver executando "Docker-compor"), as rotas de exemplo a seguir serão executadas.

Por exemplo, ao visitar a URL upstream <http://localhost:5202/api/v1/c/catalog/items/2> atendida pelo Gateway de API webshoppingapigw, você obterá o mesmo resultado da URL Downstream interna

<http://catalog-api/api/v1/2> dentro do host do Docker, como no navegador a seguir.

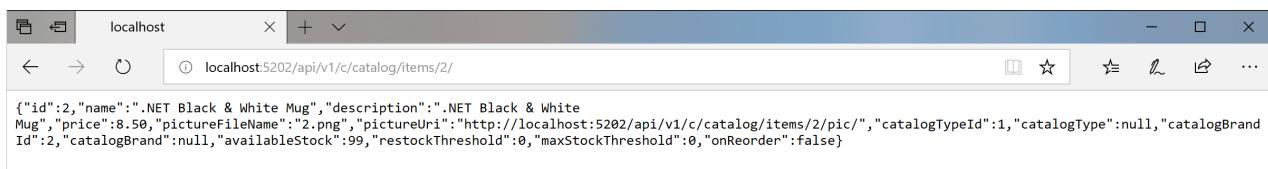
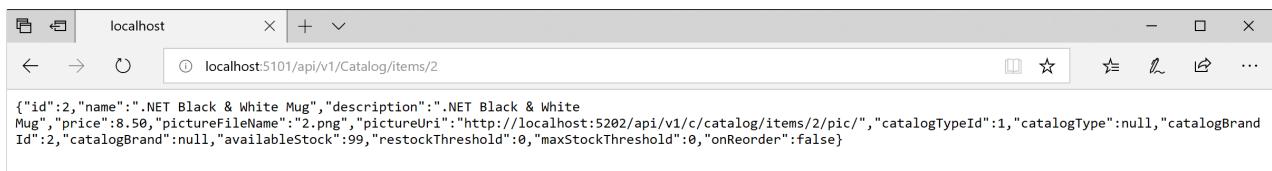


Figura 6-35. Acessando um microsserviço por meio de uma URL fornecida pelo Gateway de API

Devido a motivos de teste ou depuração, se você quisesse acessar diretamente o contêiner do Docker do catálogo (somente no ambiente de desenvolvimento) sem passar pelo gateway de API, como 'Catalog-API' é uma resolução de DNS interna ao host do Docker (descoberta de serviço manipulada por nomes de serviço do Docker-Compose), a única maneira de acessar diretamente o contêiner é por meio da porta externa publicada no Docker-Compose. Override. yml, que é fornecida somente para testes de desenvolvimento, como

<http://localhost:5101/api/v1/Catalog/items/1> no navegador a seguir.



**Figura 6-36.** Acesso direto a um microsserviço para fins de teste

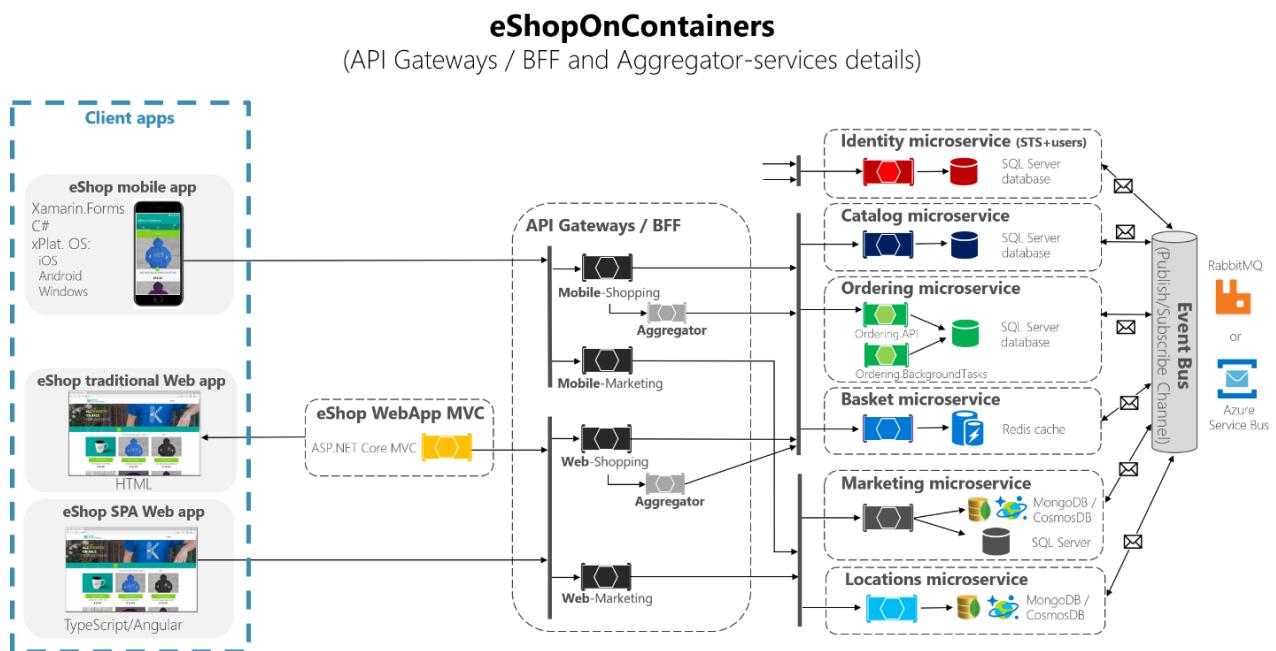
Mas o aplicativo está configurado para que ele acesse todos os microsserviços por meio dos gateways de API, não apesar da porta direta "atalhos".

### O padrão de agregação do Gateway em eShopOnContainers

Conforme apresentado anteriormente, uma maneira flexível de implementar a agregação de solicitações é com os serviços personalizados, por código. Você também pode implementar a agregação de solicitação com o [recurso de agregação de solicitação no Ocelot](#), mas talvez não seja tão flexível quanto necessário. Portanto, a maneira selecionada de implementar a agregação em eShopOnContainers é com um serviço de API Web explícito ASP.NET Core para cada agregador.

De acordo com essa abordagem, o diagrama de composição do Gateway de API é, na realidade, um pouco mais amplo ao considerar os serviços de agregador que não são mostrados no diagrama de arquitetura global simplificado mostrado anteriormente.

No diagrama a seguir, você também poderá ver como os serviços do agregador funcionam com seus Gateways de API relacionados.



**Figura 6-37.** Arquitetura de eShopOnContainers com serviços do agregador

Ampliando ainda mais, na área de negócios de "compras" na imagem a seguir, você pode ver que a informação entre os aplicativos cliente e os microsserviços é reduzida ao usar os serviços de agregador nos gateways de API.

# eShopOnContainers

(API Gateways / BFF and Aggregator-services zoom-in)

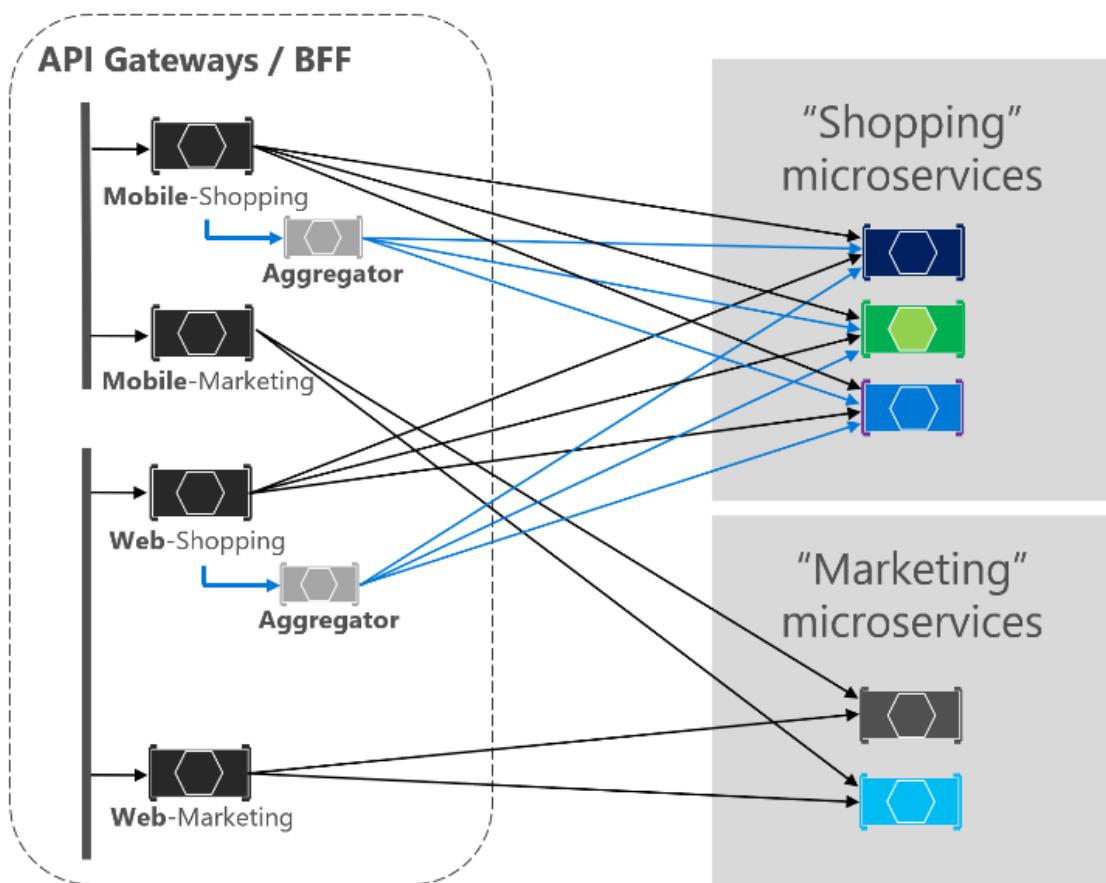


Figura 6-38. Visão ampliada do agregador de serviços

Você pode observar como quando o diagrama mostra as possíveis solicitações provenientes dos gateways de API que podem ser complexas. Apesar disso, veja como as setas em azul seriam simplificadas, da perspectiva dos aplicativos clientes, ao usar o padrão de agregador reduzindo o excesso de comunicação e a latência na comunicação, por fim, melhorando significativamente a experiência do usuário para os aplicativos remotos (aplicativos móveis e SPA), principalmente.

No caso da área de negócios e dos microserviços de "marketing", é um caso de uso simples, portanto, não há necessidade de usar agregadores, mas também pode ser possível, se necessário.

## Autenticação e autorização em Gateways de API do Ocelot

Em um Gateway de API do Ocelot, você pode usar o serviço de autenticação, como um serviço de API Web ASP.NET Core, usando [IdentityServer](#) e fornecendo o token de autenticação dentro ou fora do Gateway de API.

Como o eShopOnContainers está usando vários Gateways de API com limites baseados em BFF e em áreas de negócios, o serviço de identidade/autenticação é deixado de fora dos Gateways de API, conforme está realçado em amarelo no diagrama a seguir.

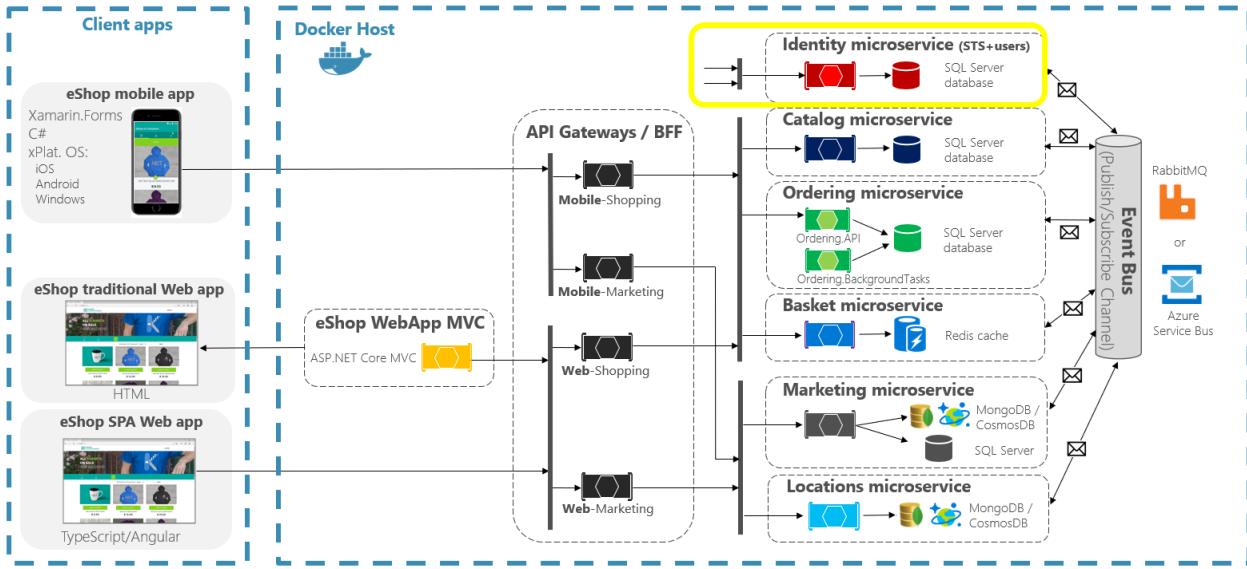


Figura 6-39. A posição do serviço de identidade em eShopOnContainers

No entanto, o Ocelot também dá suporte ao uso do microsserviço de Identidade/Autenticação dentro do limite do Gateway de API, como neste outro diagrama.

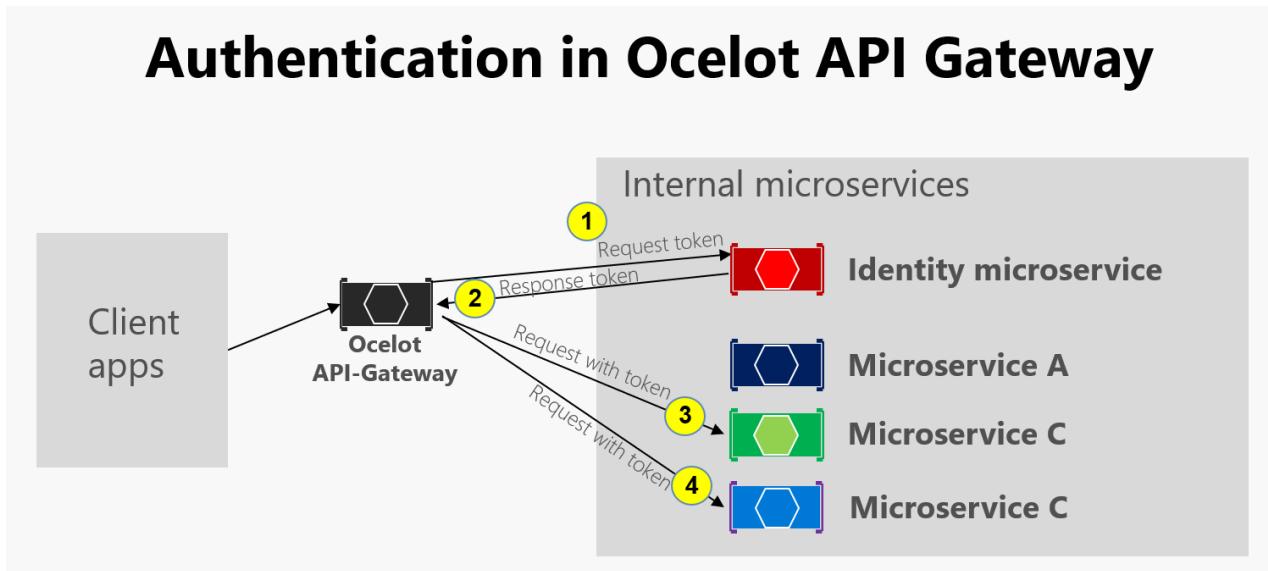


Figura 6-40. Autenticação no Ocelot

Como mostra o diagrama anterior, quando o microserviço de identidade está abaixo do gateway de API (AG): 1) o AG solicita um token de autenticação do microserviço de identidade, 2) o microserviço de identidade retorna token para AG, 3-4) solicitações AG de microserviços usando o token de autenticação. Como o aplicativo eShopOnContainers dividiu o gateway de API em vários BFF (backend para front-end) e gateways de API de áreas de negócios, outra opção seria criar um gateway de API adicional para preocupações abrangentes. Essa opção seria razoável em uma arquitetura de microsserviço bem mais complexa com vários microsserviços de interesses paralelos. Como há apenas uma preocupação cruzada em eShopOnContainers, foi decidido simplesmente lidar com o serviço de segurança fora do realm do gateway de API, para simplificar a simplicidade.

Em qualquer caso, se o aplicativo estiver protegido no nível do Gateway de API, o módulo de autenticação do Gateway de API do Ocelot será acessado primeiro quando houver uma tentativa de usar qualquer microsserviço protegido. Isso redireciona a solicitação HTTP para visitar o microserviço de identidade ou autenticação para obter o token de acesso para que você possa visitar os serviços protegidos com o `access_token`.

É a maneira de proteger com autenticação qualquer serviço no nível do Gateway de API é definir o `AuthenticationProviderKey` em suas configurações relacionadas no `configuration.json`.

```
{
    "DownstreamPathTemplate": "/api/{version}/{everything}",
    "DownstreamScheme": "http",
    "DownstreamHostAndPorts": [
        {
            "Host": "basket-api",
            "Port": 80
        }
    ],
    "UpstreamPathTemplate": "/api/{version}/b/{everything}",
    "UpstreamHttpMethod": [],
    "AuthenticationOptions": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "AllowedScopes": []
    }
}
```

Quando o Ocelot for executado, ele examinará o redirecionamentos. AuthenticationProviderKey e verificará se há um provedor de autenticação registrado com a chave fornecida. Se não houver, o Ocelot não será iniciado. Se houver, o reencaminhamento usará esse provedor quando for executado.

Como o WebHost do Ocelot é configurado com o `authenticationProviderKey = "IdentityApiKey"`, ele exigirá a autenticação sempre que esse serviço tiver alguma solicitação sem nenhum token de autenticação.

```
namespace OcelotApiGw
{
    public class Startup
    {
        private readonly IConfiguration _cfg;

        public Startup(IConfiguration configuration) => _cfg = configuration;

        public void ConfigureServices(IServiceCollection services)
        {
            var identityUrl = _cfg.GetValue<string>("IdentityUrl");
            var authenticationProviderKey = "IdentityApiKey";
            //...
            services.AddAuthentication()
                .AddJwtBearer(authenticationProviderKey, x =>
            {
                x.Authority = identityUrl;
                x.RequireHttpsMetadata = false;
                x.TokenValidationParameters = new
                    Microsoft.IdentityModel.Tokens.TokenValidationParameters()
                {
                    ValidAudiences = new[] { "orders", "basket", "locations", "marketing",
                    "mobileshoppingagg", "webshoppingagg" }
                };
            });
            //...
        }
    }
}
```

Em seguida, você também precisará definir a autorização com o atributo [Authorize] nos recursos a serem acessados, como os microsserviços, como no seguinte controlador do microsserviço Cesta.

```

namespace Microsoft.eShopOnContainers.Services.Basket.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class BasketController : Controller
    {
        //...
    }
}

```

O `ValidAudiences`, como "cesta", está correlacionado ao público definido em cada microserviço com `AddJwtBearer()` em `configurarservices()` da classe de inicialização, como no código a seguir.

```

// prevent from mapping "sub" claim to nameidentifier.
JwtSecurityTokenHandler.DefaultInboundClaimTypeMap.Clear();

var identityUrl = Configuration.GetValue<string>("IdentityUrl");

services.AddAuthentication(options =>
{
    options.DefaultAuthenticateScheme = JwtBearerDefaults.AuthenticationScheme;
    options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;

}).AddJwtBearer(options =>
{
    options.Authority = identityUrl;
    options.RequireHttpsMetadata = false;
    options.Audience = "basket";
});

```

Se você tentar acessar qualquer microserviço protegido, como o microserviço da cesta com uma URL de redirecionamento com base no gateway de API como `http://localhost:5202/api/v1/b/basket/1`, você receberá um 401 não autorizado, a menos que forneça um token válido. Por outro lado, se uma URL de redirecionamento for autenticada, o Ocelot invocará qualquer esquema downstream associado a ele (a URL interna de microatendimento).

**Autorização na camada de redirecionamentos do Ocelot.** O Ocelot dá suporte a autorização baseada em declarações avaliada após a autenticação. Defina a autorização em um nível de rota adicionando as linhas a seguir à configuração de Reencaminhamento.

```

"RouteClaimsRequirement": {
    "UserType": "employee"
}

```

Nesse exemplo, quando o middleware de autorização for chamado, o Ocelot descobrirá se o usuário tem o tipo de declaração 'UserType' no token e se o valor dessa declaração é 'employee'. Se não estiver, o usuário não será autorizado e a resposta será 403 Proibido.

## Usando a entrada do Kubernetes e os Gateways de API do Ocelot

Ao usar o kubernetes (como em um cluster do serviço kubernetes do Azure), você geralmente unificará todas as solicitações HTTP por meio da [camada de entrada do kubernetes](#) com base em *Nginx*.

No kubernetes, se você não usar nenhuma abordagem de entrada, seus serviços e pods terão IPs somente roteáveis pela rede de cluster.

Mas se você usar uma abordagem de entrada, haverá uma camada intermediária entre a Internet e seus serviços (incluindo seus Gateways de API), atuando como um proxy reverso.

Como uma definição, uma entrada é uma coleção de regras que permitem que conexões de entrada acessem os serviços de cluster. Uma entrada é geralmente configurada para fornecer URLs acessadas externamente, balancear a carga do tráfego, terminação SSL e muito mais. Os usuários solicitam a entrada postando o recurso de entrada no servidor de API.

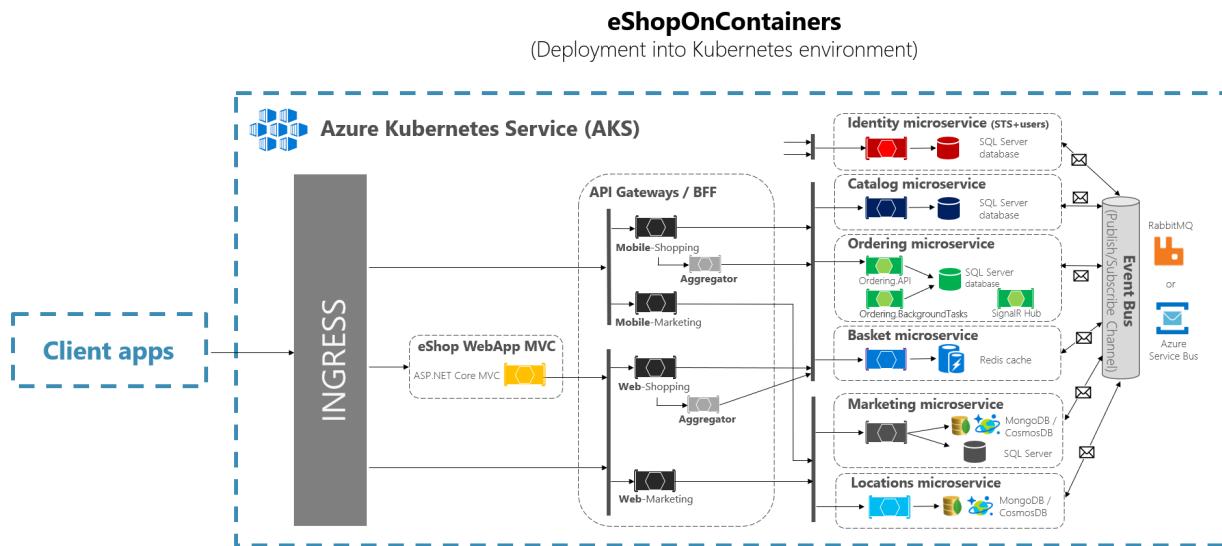
Em eShopOnContainers, ao desenvolver localmente e usar apenas o computador de desenvolvimento como o host do Docker, você não está usando nenhuma entrada, mas apenas vários Gateways de API.

No entanto, ao direcionar um ambiente de "produção" baseado em kubernetes, eShopOnContainers está usando uma entrada na frente dos gateways de API. Dessa forma, os clientes ainda chamam a mesma URL base, mas as solicitações são encaminhadas para vários Gateways de API ou BFFs.

Os gateways de API são front-ends ou as fachadas identificando apenas os serviços, mas não os aplicativos Web que normalmente estão fora de seu escopo. Além disso, os Gateways de API podem ocultar determinados microsserviços internos.

A entrada, no entanto, está apenas redirecionando solicitações HTTP, mas não está tentando ocultar nenhum microsserviço ou aplicativo Web.

A arquitetura ideal é uma camada Nginx de entrada no Kubernetes na frente dos aplicativos Web além de vários Gateways de API/BFF do Ocelot, conforme é mostrado no diagrama a seguir.



**Figura 6-41.** A camada de entrada em eShopOnContainers, quando implantada no Kubernetes

Uma Entrada do Kubernetes funciona como um proxy reverso para todo o tráfego para o aplicativo, incluindo os aplicativos Web, que geralmente estão fora do escopo do gateway de API. Quando você implanta o eShopOnContainers no Kubernetes, ele expõe apenas alguns serviços ou pontos de extremidade via *entrada*, ou seja, basicamente, a seguinte lista de pós-fixados nas URLs:

- `/` para o aplicativo Web cliente SPA
- `/webmvc` para o aplicativo Web cliente MVC
- `/webstatus` para o aplicativo Web cliente mostrando o status e as verificações de integridade
- `/webshoppingapigw` para os processos de negócios Web BFF e de compras
- `/webmarketingapigw` para os processos de negócios Web BFF e de marketing
- `/mobileshoppingapigw` para os processos de negócios móveis BFF e de compras
- `/mobilemarketingapigw` para os processos de negócios móveis BFF e de marketing

Ao implantar no kubernetes, cada gateway de API do Ocelot está usando um arquivo "configuration.json" diferente para cada *Pod* que executa os gateways de API. Esses arquivos "configuration.json" são fornecidos pela montagem (originalmente com o script `deploy.ps1`) de um volume criado com base em um *mapa de configuração*

kubernetes chamado ' Ocelot '. Cada contêiner monta seu arquivo de configuração relacionado na pasta do contêiner chamada `/app/configuration` .

Nos arquivos de código-fonte do eShopOnContainers, os arquivos originais "configuration.jsem" podem ser encontrados na `k8s/ocelot/` pasta. Há um arquivo para cada BFF/APIGateway.

## Recursos de interesses paralelos adicionais em um Gateway de API do Ocelot

Há outros recursos importantes que podem ser pesquisados e usados ao usar um Gateway de API do Ocelot, descritos nos links a seguir.

- Descoberta de serviço no lado do cliente, integrando Ocelot com Consul ou Eureka  
<https://ocelot.readthedocs.io/en/latest/features/servicediscovery.html>
- Caching na camada de gateway de API  
<https://ocelot.readthedocs.io/en/latest/features/caching.html>
- Registro em log na camada de gateway de API  
<https://ocelot.readthedocs.io/en/latest/features/logging.html>
- Qualidade de serviço (repetições e disjuntores de circuito) na camada de gateway de API  
<https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html>
- Limitação de taxa  
<https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html>

[ANTERIOR](#)

[AVANÇAR](#)

# Lidar com a complexidade dos negócios em um microsserviço com padrões DDD e CQRS

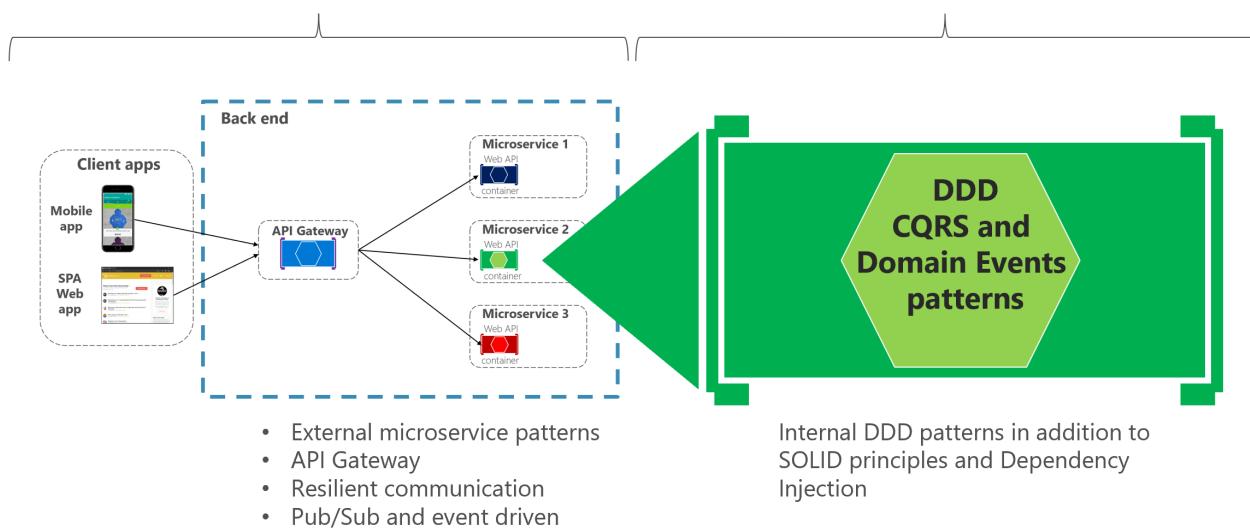
10/09/2020 • 4 minutes to read • [Edit Online](#)

*Crie um modelo de domínio para cada microsserviço ou contexto limitado que reflete o entendimento do domínio da empresa.*

Esta seção concentra-se em microsserviços mais avançados que você implementa quando precisa lidar com subsistemas complexos ou com microsserviços derivados do conhecimento de especialistas no domínio com mudanças constantes nas regras de negócio. Os padrões de arquitetura usados nesta seção são baseados nas abordagens DDD (design controlado por domínio) e CQRS (Segregação de Responsabilidade de Comando e Consulta), conforme é ilustrado na Figura 7-1.

## External architecture per application

## Internal architecture per microservice



Diferença entre arquitetura externa: padrões de microsserviço, gateways de API, comunicação resilientes, pub/sub etc. e a arquitetura interna: controlado por dados/CRUD, padrões de DDD, injeção de dependência, várias bibliotecas etc.

**Figura 7-1.** Arquitetura externa de microsserviço versus padrões de arquitetura interna para cada microsserviço

No entanto, a maioria das técnicas de microsserviços controlados por dados, incluindo como implementar um serviço de API Web do ASP.NET Core ou como expor metadados com Swashbuckle ou NSwag, também se aplica aos microsserviços mais avançados implementados internamente com padrões DDD. Esta seção é uma extensão das seções anteriores, pois a maioria das práticas de explicadas anteriormente também se aplicam aqui ou para qualquer tipo de microsserviço.

Esta seção primeiro fornece detalhes sobre os padrões CQRS simplificados usados no aplicativo de referência eShopOnContainers. Posteriormente, você obterá uma visão geral das técnicas de DDD que permitem encontrar padrões comuns que podem ser reutilizados em seus aplicativos.

O DDD é um tópico grande com um conjunto avançado de recursos de aprendizagem. Você pode iniciar com guias como [Domain-Driven Design](#) (Design controlado por domínio) do Eric Evans e os materiais adicionais dos autores Vernon, Jimmy Nilsson, Greg Young, Udi Dahan, Jimmy Bogard e muitos outros especialistas em

DDD/CQRS. Mas, sobretudo, você precisa tentar aprender como aplicar as técnicas de DDD das sessões de conversa, de quadro de comunicações e de modelagem com os especialistas em seu domínio de negócios concreto.

#### Recursos adicionais

DDD (design controlado por domínio)

- Eric Evans. Idioma do domínio

<https://domainlanguage.com/>

- Martin Fowler. Design controlado por domínio

<https://martinfowler.com/tags/domain%20driven%20design.html>

- Jimmy Bogard. Fortalecendo seu domínio: um primer

<https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/>

Guias sobre DDD

- Eric Evans. Design controlado por domínio: solução de complexidade no coração do software

<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

- Eric Evans. Referência de design controlada por domínio: definições e resumos de padrões

<https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-2014-09-22/dp/B01N8YB4ZO/>

- Vaughn Vernon. Implementando o design controlado por domínio

<https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/>

- Vaughn Vernon. Design controlado por domínio difixado

<https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420/>

- Jimmy Nilsson. Aplicando padrões e design orientados a domínio

<https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202/>

- Cesar de la Torre. Guia de arquitetura orientada a domínio N camadas com o .NET

<https://www.amazon.com/N-Layered-Domain-Oriented-Architecture-Guide-NET/dp/8493903612/>

- Abel Avram e algoritmo Floyd Marinescu. Design controlado por domínio rapidamente

<https://www.amazon.com/Domain-Driven-Design-Quickly-Abel-Avram/dp/1411609255/>

- Scott Millett, ajuste de Nick-padrões, princípios e práticas de design controlado por domínio

<https://www.wiley.com/Patterns%2C+Principles%2C+and+Practices+of+Domain+Driven+Design-p-9781118714706>

Treinamento em DDD

- Julie Lerman e Steve Smith. Conceitos básicos de design controlado por domínio

<https://bit.ly/PS-DDD>

ANTERIOR

AVANÇAR

# Aplicar padrões CQRS e DDD simplificados em um microsserviço

07/04/2020 • 6 minutes to read • [Edit Online](#)

O CQRS é um padrão de arquitetura que separa os modelos para ler e gravar dados. O termo relacionado [CQS \(Separação de Comando-Consulta\)](#) foi originalmente definido por Bertrand Meyer em seu livro *Object Oriented Software Construction* (Construção de software orientada a objeto). A idéia básica é que você pode dividir as operações de um sistema em duas categorias bem separadas:

- Consultas. Essas retornam um resultado, não alteram o estado do sistema e são livres de efeitos colaterais.
- comandos. Esses alteram o estado de um sistema.

CQS é um conceito simples: trata-se de métodos dentro do mesmo objeto sendo consultas ou comandos. Cada método retorna um estado ou muda um estado, mas não ambos. Até mesmo um único objeto de padrão de repositório pode estar em conformidade com o CQS. O CQS pode ser considerado um princípio fundamental para o CQRS.

O [CQRS \(Segregação de Responsabilidade de Consulta e Comando\)](#) foi introduzido por Greg Young e altamente promovido por Udi Dahan e outros. Ele se baseia no princípio do CQS, embora seja mais detalhado. Ele pode ser considerado um padrão com base em comandos e eventos, além de ser opcional em mensagens assíncronas. Em muitos casos, o CQRS está relacionado a cenários mais avançados, como ter um banco de dados físico para leituras (consultas) diferente do banco de dados para gravações (atualizações). Além disso, um sistema CQRS mais evoluído pode implementar [ES \(fonte de eventos\)](#) em seu banco de dados de atualizações. Assim, você deve apenas armazenar eventos no modelo de domínio, em vez de armazenar os dados do estado atual. No entanto, esta abordagem não é usada neste guia. Este guia usa a abordagem CQRS mais simples, que consiste apenas em separar as consultas dos comandos.

O aspecto de separação do CQRS é obtido pelo agrupamento de operações de consulta em uma camada e comandos em outra camada. Cada camada tem seu próprio modelo de dados (observe que dizemos modelo, não necessariamente um banco de dados diferente) e é criada usando sua própria combinação de padrões e tecnologias. Além disso, as duas camadas podem estar dentro do mesmo nível ou microsserviço, como no exemplo (microsserviço de ordenação) usado para este guia. Ou elas podem ser implementadas em diferentes microsserviços ou processos para que possam ser otimizadas e expandidas separadamente sem afetar umas às outras.

CQRS significa ter dois objetos para uma operação de leitura/gravação, em que, em outros, há um. Há motivos para ter um banco de dados de leitura desnormalizado, sobre o qual você pode aprender na literatura sobre CQRS mais avançada. Mas não estamos usando essa abordagem aqui, em que a meta é ter mais flexibilidade nas consultas, em vez de limitá-las com restrições de padrões DDD como agregações.

Um exemplo desse tipo de serviço é o microsserviço de ordenação do aplicativo eShopOnContainers de referência. Este serviço implementa um microsserviço com base em uma abordagem CQRS simplificada. Ele usa uma única fonte de dados ou banco de dados, mas dois modelos lógicos, além de padrões DDD para o domínio transacional, conforme mostrado na Figura 7-2.

# Simplified CQRS and DDD microservice

## High level design

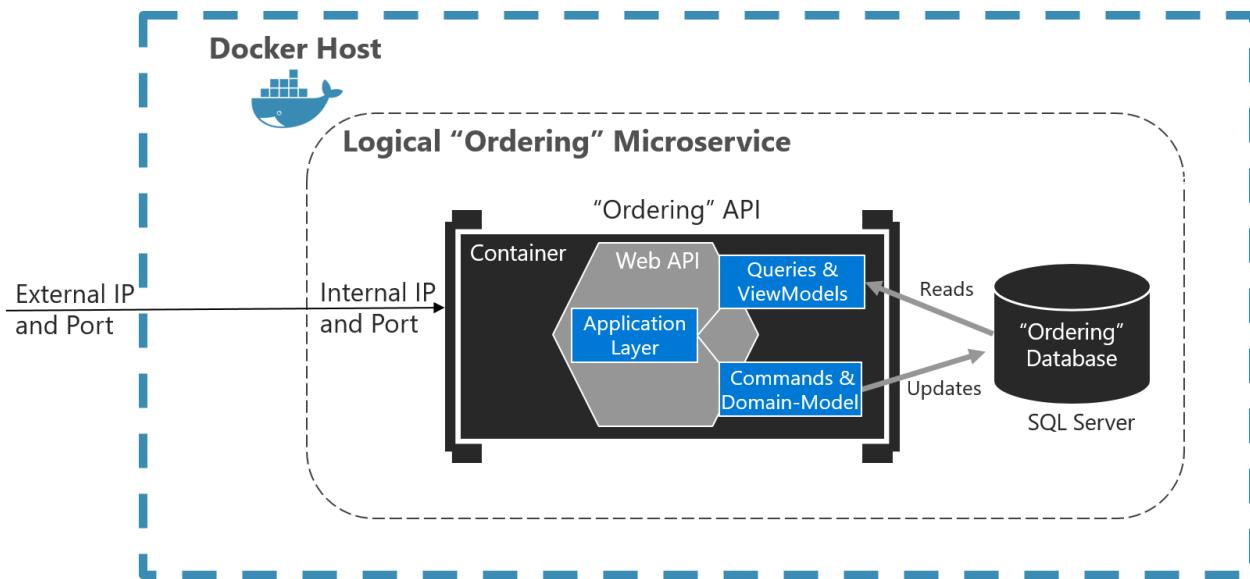


Figura 7-2. Microsserviço baseado em CQRS e DDD simplificado

O Microserviço lógico “Ordering” inclui seu banco de dados de pedidos, que pode ser, mas não precisa ser, o mesmo host Docker. É bom ter o banco de dados no mesmo host do Docker para desenvolvimento, mas não para produção.

A camada de aplicativo pode ser a própria API Web. O aspecto de design importante aqui é que o microsserviço dividiu as consultas e ViewModels (modelos de dados criados especialmente para os aplicativos cliente) dos comandos, do modelo de domínio e das transações que seguem o padrão CQRS. Essa abordagem mantém as consultas independentes de restrições provenientes de padrões DDD que só fazem sentido para transações e atualizações, conforme explicado nas seções posteriores.

## Recursos adicionais

- O Greg Young. *Versão em um sistema de origem de eventos* (livre para ler e-book on-line)  
<https://leanpub.com/esversioning/read>

PRÓXIMO

ANTERIOR

# Aplicar abordagens CQRS e CQS em um microsserviço DDD em eShopOnContainers

10/09/2020 • 6 minutes to read • [Edit Online](#)

O design do microsserviço de ordenação no aplicativo de referência eShopOnContainers é baseado nos princípios CQRS. No entanto, ele usa a abordagem mais simples, que está separando as consultas dos comandos e usando o mesmo banco de dados para ambas as ações.

A essência desses padrões e o ponto importante aqui é que as consultas são idempotentes: não importa quantas vezes você consulte um sistema, o estado desse sistema não será alterado. Em outras palavras, as consultas não têm efeito colateral.

Portanto, você poderia usar um modelo de dados diferente de "leituras" do que o modelo de domínio "gravações" da lógica transacional, mesmo que os microserviços de pedidos estejam usando o mesmo banco de dados.

Portanto, essa é uma abordagem de CQRS simplificada.

Por outro lado, comandos, que disparam transações e atualizações de dados, alteram o estado no sistema. Com os comandos, é necessário ter cuidado ao lidar com a complexidade e com regras de negócio em constante mudança. É o local em que você deseja aplicar as técnicas DDD para ter um sistema modelado melhor.

Os padrões DDD apresentados neste guia não devem ser aplicados universalmente. Eles apresentam restrições em seu design. Essas restrições oferecem benefícios como maior qualidade com o tempo, principalmente em comandos e outro código que modifica o estado do sistema. No entanto, essas restrições adicionam complexidade com menos benefícios para ler e consultar dados.

Um desses padrões é o padrão de agregação, que examinaremos mais nas seções posteriores. Resumidamente, no padrão de agregação, você trata muitos objetos de domínio como uma única unidade como resultado de sua relação no domínio. Você nem sempre pode obter vantagens desse padrão em consultas; ele pode aumentar a complexidade da lógica de consulta. Para consultas somente leitura, você não obtém as vantagens de tratar vários objetos como uma única agregação. Você somente obtém a complexidade.

Como mostra a Figura 7-2 na seção anterior, este guia sugere o uso de padrões DDD somente na área transacional/atualizações do seu microserviço (ou seja, como disparado por comandos). As consultas podem seguir uma abordagem mais simples e devem estar separadas de comandos, seguindo uma abordagem CQRS.

Para implementar o "lado das consultas", você pode escolher entre muitas abordagens, de seu ORM completo como EF Core, projeções do automapeamento, procedimentos armazenados, exibições, exibições materializadas ou um micro ORM.

Neste guia e nos eShopOnContainers (principalmente o microsserviço de ordenação), escolhemos implementar consultas diretas usando um micro ORM como o [Dapper](#). Isso permite que você implemente qualquer consulta baseada em instruções SQL para obter o melhor desempenho, graças a uma estrutura leve com pouca sobrecarga.

Quando você usa essa abordagem, todas as atualizações para seu modelo que afetam o modo como as entidades são persistidas em um banco de dados SQL também precisam de atualizações separadas para consultas SQL usadas por Dapper ou quaisquer outras abordagens separadas (não-EF) para consulta.

## Os padrões CQRS e DDD não são arquiteturas de nível superior

É importante entender que os padrões CQRS e a maioria dos padrões DDD (como camadas DDD ou um modelo de domínio com agregações) não são estilos de arquitetura, mas apenas padrões de arquitetura. Microsserviços, SOA e EDA (arquitetura orientada a eventos) são exemplos de estilos de arquitetura. Eles descrevem um sistema

de muitos componentes, como muitos microsserviços. Os padrões CQRS e DDD descrevem algo dentro de um único sistema ou componente; nesse caso, algo dentro de um microsserviço.

Diferentes BCs (Contextos limitados) empregarão diferentes padrões. Eles têm diferentes responsabilidades, e isso leva a diferentes soluções. Vale destacar que forçar o mesmo padrão em todos os lugares leva a uma falha. Não use padrões CQRS nem DDD em todos os lugares. Muitos subsistemas, BCs ou microsserviços são mais simples e podem ser implementados mais facilmente usando serviços CRUD simples ou outra abordagem.

Há apenas uma arquitetura de aplicativo: a arquitetura do sistema ou o aplicativo de ponta a ponta que você está criando (por exemplo, a arquitetura de microsserviços). No entanto, o design de cada Contexto limitado ou microsserviço dentro desse aplicativo reflete suas próprias compensações e decisões de design interno em um nível de padrões de arquitetura. Não tente aplicar os mesmos padrões de arquitetura como CQRS ou DDD em todos os lugares.

## Recursos adicionais

- **Martin Fowler. CQRS**  
<https://martinfowler.com/bliki/CQRS.html>
- **Greg Young. Documentos CQRS**  
[https://cqrss.files.wordpress.com/2010/11/cqrs\\_documents.pdf](https://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf)
- **Udi Dahan. CQRS esclarecido**  
<https://udidahan.com/2009/12/09/clarified-cqrs/>

[ANTERIOR](#)

[AVANÇAR](#)

# Implementando leituras/consultas em um microsserviço CQRS

10/09/2020 • 15 minutes to read • [Edit Online](#)

Para leituras/consultas, o microsserviço de ordenação do aplicativo de referência eShopOnContainers implementa as consultas independentemente do modelo DDD e da área transacional. Isso foi feito principalmente porque as exigências para consultas e transações são totalmente diferentes. Grava transações de execução que devem estar em conformidade com a lógica do domínio. Consultas, por outro lado, são idempotentes e podem ser separadas das regras de domínio.

A abordagem é simples, conforme mostra a Figura 7-3. A interface de API é implementada pelos controladores de API da Web usando qualquer infraestrutura, como um micro ORM (Mapeador Relacional de Objeto) como Dapper e retornando ViewModels dinâmicos dependendo das necessidades dos aplicativos de interface do usuário.

## High level “Queries-side” in a simplified CQRS

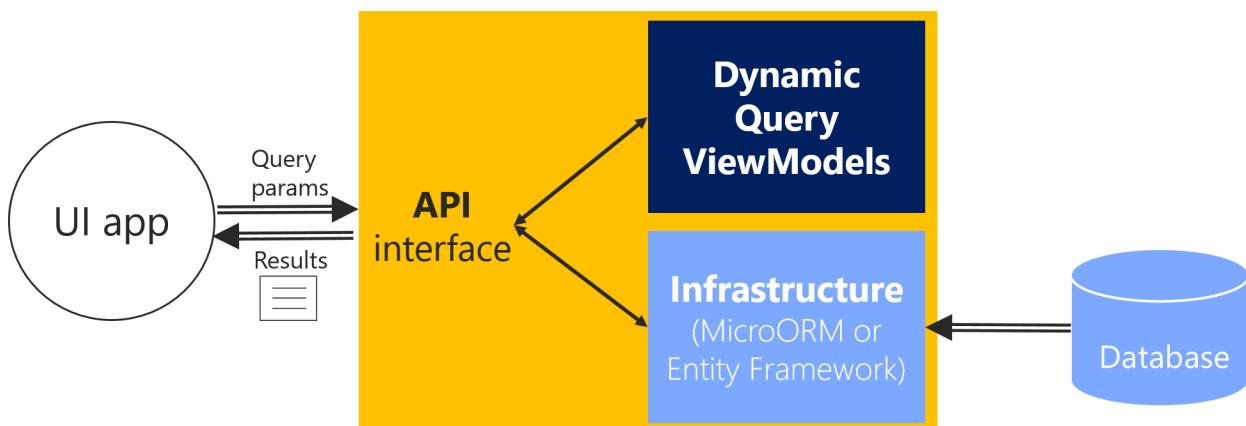


Figura 7-3. A abordagem mais simples para consultas em um microsserviço CQRS

A abordagem mais simples para as consultas no lado de uma abordagem de CQRS simplificada pode ser implementada consultando o banco de dados com um micro ORM como Dapper, retornando ViewModels dinâmicos. As definições de consulta consultam o banco de dados e retornam um ViewModel dinâmico criado dinamicamente para cada consulta. Uma vez que as consultas são idempotentes, elas não alteram os dados, não importa quantas vezes você execute uma consulta. Portanto, você não precisa estar restrito por nenhum padrão DDD usado no lado do transacional, como agregações e outros padrões, e é por isso que as consultas são separadas da área de trabalho transacional. Você consulta o banco de dados em busca de quais são as necessidades da interface do usuário e retorna um ViewModel dinâmico que não precisa ser definido estaticamente em qualquer lugar (nenhuma classe para ViewModels), exceto nas próprias instruções SQL.

Como essa abordagem é simples, o código necessário para o lado das consultas (como código usando um micro ORM como [Dapper](#)) pode ser implementado [dentro do mesmo projeto de API da Web](#). A Figura 7-4 mostra isso. As consultas são definidas no projeto de microsserviço `Ordering.API` dentro da solução eShopOnContainers.

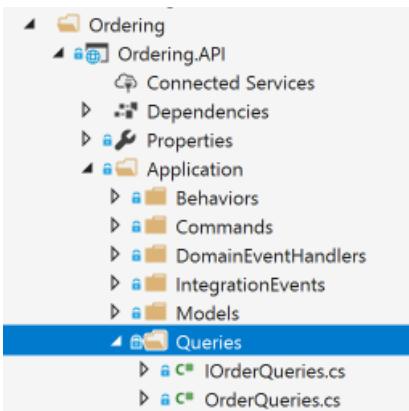


Figura 7-4. Consultas no microsserviço de Ordenação em eShopOnContainers

## Usar ViewModels feitos especificamente para aplicativos cliente, independentemente de restrições do modelo de domínio

Uma vez que as consultas são executadas para obter os dados necessários para os aplicativos cliente, o tipo retornado pode ser feito especificamente para os clientes com base nos dados retornados pelas consultas. Esses modelos ou DTOs (Objetos de Transferência de Dados) são chamados de ViewModels.

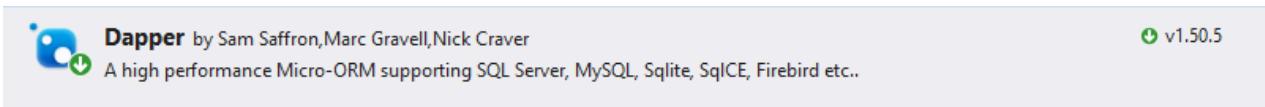
Os dados retornados (ViewModel) podem ser o resultado da associação de dados de várias entidades ou tabelas no banco de dados, ou mesmo entre várias agregações definidas no modelo de domínio para a área transacional. Nesse caso, como você está criando consultas independentes do modelo de domínio, os limites e restrições de agregações são ignorados e você pode consultar qualquer tabela e coluna que você possa precisar. Essa abordagem fornece grande flexibilidade e produtividade para os desenvolvedores criarem ou atualizarem as consultas.

Os ViewModels podem ser tipos estáticos definidos em classes (como é implementado no microserviço de ordenação). Ou podem ser criados dinamicamente com base nas consultas executadas, o que é muito ágil para os desenvolvedores.

## Usar o Dapper como um micro ORM para executar consultas

Você pode usar qualquer micro ORM, Entity Framework Core ou até mesmo ADO.NET simples para a consulta. O aplicativo de exemplo, o Dapper foi selecionado para o microsserviço de ordenação em eShopOnContainers como um bom exemplo de um micro ORM popular. Ele pode executar consultas SQL simples com ótimo desempenho, pois é uma estrutura leve. Usando o Dapper, você pode escrever uma consulta SQL que pode acessar e unir várias tabelas.

O Dapper é um projeto de software livre (original criado por Sam Saffron) e faz parte dos blocos de construção usado no [Stack Overflow](#). Para usar o Dapper, basta instalá-lo por meio do [pacote Dapper NuGet](#), conforme mostra a figura a seguir:



Você também precisa adicionar uma `using` diretiva para que seu código tenha acesso aos métodos de extensão Dapper.

Quando você usa o Dapper em seu código, usa diretamente a classe `SqlConnection` disponível no namespace `System.Data.SqlClient`. Por meio do método `QueryAsync` e de outros métodos de extensão que estendem a `SqlConnection` classe, você pode executar consultas de uma maneira simples e de alto desempenho.

# ViewModels dinâmicos versus estáticos

Ao retornar ViewModels do lado do servidor para aplicativos cliente, você pode pensar sobre esses ViewModels como DTOs (Objetos de Transferência de Dados) que podem ser diferentes para as entidades de domínio internas do seu modelo de entidade porque os ViewModels contêm os dados da maneira como o aplicativo cliente precisa. Portanto, em muitos casos, você pode agregar dados provenientes de várias entidades de domínio e compor os ViewModels exatamente de acordo com a maneira como o aplicativo cliente precisa daqueles dados.

Esses ViewModels ou DTOs podem ser definidos explicitamente (como classes de portador de dados), como a `OrderSummary` classe mostrada em um trecho de código posterior. Ou, você poderia simplesmente retornar ViewModels dinâmicos ou DTOs dinâmicos com base nos atributos retornados por suas consultas como um tipo dinâmico.

## ViewModel como tipo dinâmico

Conforme mostrado no código a seguir, um `viewModel` pode ser diretamente retornado pelas consultas retornando um tipo *dinâmico* internamente baseado nos atributos retornados por uma consulta. Isso significa que o subconjunto de atributos a serem retornados é baseado na própria consulta. Portanto, se você adicionar uma nova coluna à consulta ou junção, esses dados serão adicionados dinamicamente ao `viewModel` retornado.

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;

public class OrderQueries : IOrderQueries
{
    public async Task<IEnumerable<dynamic>> GetOrdersAsync()
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            return await connection.QueryAsync<dynamic>(
                @"SELECT o.[Id] as ordernumber,
                o.[OrderDate] as [date],os.[Name] as [status],
                SUM(oi.units*oi.unitprice) as total
                FROM [ordering].[Orders] o
                LEFT JOIN[ordering].[orderitems] oi ON o.Id = oi.orderid
                LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId = os.Id
                GROUP BY o.[Id], o.[OrderDate], os.[Name]");
        }
    }
}
```

O ponto importante é que, ao usar um tipo dinâmico, a coleção de dados retornada é montada dinamicamente como o ViewModel.

**Prós:** Essa abordagem reduz a necessidade de modificar classes ViewModel estáticas sempre que você atualiza a frase SQL de uma consulta, tornando essa abordagem de design ágil durante a codificação, a simples e a rápida evolução em relação às alterações futuras.

**Contras:** no longo prazo, tipos dinâmicos podem afetar negativamente a clareza e a compatibilidade de um serviço com aplicativos cliente. Além disso, o software middleware como Swashbuckle não poderá fornecer o mesmo nível de documentação em tipos retornados ao usar tipos dinâmicos.

## ViewModel como classes DTO predefinidas

**Prós:** ter classes ViewModel estáticas e predefinidas, como "contratos" com base em classes de DTO explícitas, é definitivamente melhor para APIs públicas, mas também para microserviços de longo prazo, mesmo se elas forem

usadas apenas pelo mesmo aplicativo.

Se você quiser especificar os tipos de resposta para o Swagger, precisará usar as classes DTO explícitas como o tipo de retorno. Portanto, classes DTO predefinidas permitem que você ofereça informações mais sofisticadas do Swagger. Isso melhora a documentação da API e a compatibilidade ao consumir uma API.

**Contras:** conforme mencionado anteriormente, ao atualizar o código, serão necessárias mais algumas etapas para atualizar as classes DTO.

*Dica com base em nossa experiência:* nas consultas implementadas no microserviço de pedidos no eShopOnContainers, começamos a desenvolver usando ViewModels dinâmicos, pois era simples e ágil nos estágios de desenvolvimento antecipados. Mas, depois que o desenvolvimento foi estabilizado, optamos por refatorar as APIs e usar os DTOs estáticos ou predefinidos para os ViewModels, porque ele é mais claro para que os consumidores do microserviço saibam tipos de DTO explícitos, usados como "contratos".

No exemplo a seguir, você pode ver como a consulta está retornando dados usando uma classe DTO ViewModel explícita: a classe OrderSummary.

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;

public class OrderQueries : IOrderQueries
{
    public async Task<IEnumerable<OrderSummary>> GetOrdersAsync()
    {
        using (var connection = new SqlConnection(_connectionString))
        {
            connection.Open();
            return await connection.QueryAsync<OrderSummary>(
                @"SELECT o.[Id] as ordernumber,
                o.[OrderDate] as [date],os.[Name] as [status],
                SUM(oi.units*oi.unitprice) as total
                FROM [ordering].[Orders] o
                LEFT JOIN[ordering].[orderitems] oi ON  o.Id = oi.orderid
                LEFT JOIN[ordering].[orderstatus] os on o.OrderStatusId = os.Id
                GROUP BY o.[Id], o.[OrderDate], os.[Name]
                ORDER BY o.[Id]");
        }
    }
}
```

#### Descrever os tipos de resposta de APIs da Web

Os desenvolvedores que consomem APIs e microserviços da Web estão mais preocupados com o que é retornado — especificamente tipos de resposta e códigos de erro (se não padrão). Os tipos de resposta são tratados nos comentários de XML e nas anotações de dados.

Sem a documentação adequada na interface do usuário do Swagger, o consumidor não tem conhecimento de quais tipos estão sendo retornados ou quais códigos HTTP podem ser retornados. Esse problema é corrigido adicionando o [Microsoft.AspNetCore.Mvc.ProducesResponseTypeAttribute](#), portanto, o Swashbuckle pode gerar informações mais avançadas sobre o modelo e os valores de retorno de API, conforme mostra o código a seguir:

```

namespace Microsoft.eShopOnContainers.Services.Ordering.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class OrdersController : Controller
    {
        //Additional code...
        [Route("")]
        [HttpGet]
        [ProducesResponseType(typeof(IEnumerable<OrderSummary>),
            (int) HttpStatusCode.OK)]
        public async Task<IActionResult> GetOrders()
        {
            var userid = _identityService.GetUserIdentity();
            var orders = await _orderQueries
                .GetOrdersFromUserAsync(Guid.Parse(userid));
            return Ok(orders);
        }
    }
}

```

No entanto, o atributo `ProducesResponseType` não pode usar dinâmica como um tipo, mas requer o uso de tipos explícitos, como o `OrderSummary` ViewModel DTO, mostrado no exemplo a seguir:

```

public class OrderSummary
{
    public int ordernumber { get; set; }
    public DateTime date { get; set; }
    public string status { get; set; }
    public double total { get; set; }
}

```

Essa é outra razão pela qual tipos retornados explícitos são melhores que tipos dinâmicos no longo prazo. Ao usar o atributo `ProducesResponseType`, também é possível especificar qual é o resultado esperado no que diz respeito a possíveis erros/códigos HTTP, como 200, 400, etc.

Na imagem a seguir, você pode ver como a interface do usuário Swagger mostra as informações de `ResponseType`.

**Figura 7-5.** Interface do usuário do Swagger mostrando os tipos de resposta e possíveis códigos de status HTTP de uma API da Web

A imagem mostra alguns valores de exemplo com base nos tipos ViewModel e os possíveis códigos de status HTTP que podem ser retornados.

## Recursos adicionais

- Dapper  
<https://github.com/StackExchange/dapper-dot-net>
- Julie Lerman. Pontos de dados – Dapper, Entity Framework e aplicativos híbridos (artigo da MSDN Magazine)  
[/archive/msdn-magazine/2016/may/data-points-dapper-entity-framework-and-hybrid-apps](https://archive.microsoft.com/MSDN-Magazine/2016/may/data-points-dapper-entity-framework-and-hybrid-apps)
- ASP.NET Core páginas de ajuda da API Web usando o Swagger  
[/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio](https://docs.asp.net/en/latest/tutorials/web-api-help-pages-using-swagger.html?tabs=visual-studio)

[ANTERIOR](#)
[AVANÇAR](#)

# Projetar um microsserviço orientado a DDD

09/04/2020 • 19 minutes to read • [Edit Online](#)

O DDD (design orientado a domínio) defende modelagem com base na realidade dos negócios conforme relevante para os seus casos de uso. No contexto da criação de aplicativos, o DDD fala sobre problemas como domínios. Ele descreve as áreas de problema independentes como Contextos Limitados (cada Contexto Limitado se correlaciona a um microsserviço) e enfatiza uma linguagem comum para falar sobre esses problemas. Também sugere muitos conceitos técnicos e padrões, como entidades de domínio com regras de modelos avançados (nenhum [modelo de domínio anêmico](#)), objetos de valor, agregações e raiz de agregação (ou entidade raiz) para dar suporte à implementação interna. Esta seção apresenta o design e a implementação desses padrões internos.

Às vezes, esses padrões e regras técnicas de DDD são considerados obstáculos que têm uma curva de aprendizado acentuada para implementar abordagens de DDD. Mas a parte importante não são os padrões em si, mas organizar o código para que ele fique alinhado aos problemas de negócios e usar os mesmos termos de negócios (linguagem ubíqua). Além disso, as abordagens DDD deverão ser aplicadas somente se você estiver implementando microsserviços complexos com regras de negócio significativas. Responsabilidades mais simples, como um serviço CRUD, podem ser gerenciadas com abordagens mais simples.

Onde traçar os limites é a tarefa principal ao criar e definir um microsserviço. Padrões DDD ajudam você a compreender a complexidade no domínio. Para o modelo de domínio para cada Contexto Limitado, você pode identificar e definir as entidades, os objetos de valor e agregações que modelem seu domínio. Você cria e refina um modelo de domínio contido dentro de um limite que define o seu contexto. E isso é bastante explícito na forma de um microsserviço. Os componentes dentro desses limites acabam sendo seus microsserviços, embora, em alguns casos, uma continuidade de negócios ou microsserviços de negócios possam ser compostos por vários serviços físicos. DDD é sobre limites, assim como os microsserviços.

## Mantenha os limites de contexto do microsserviço relativamente pequenos

Determinar onde colocar os limites entre Contextos Limitados equilibra dois objetivos concorrentes. Primeiro, você deseja criar inicialmente os menores microsserviços possíveis, embora isso não deva ser a meta principal; você deve criar um limite em torno de itens que precisam de coesão. Segundo, você deseja evitar comunicações verborrágica entre microsserviços. Essas metas podem contradizer umas às outras. Você deve equilibrá-las decompondo o sistema em tantos microsserviços quanto puder até ver esses limites de comunicação crescendo rapidamente a cada tentativa adicional de separar um novo contexto limitado. Coesão é essencial em um único contexto limitado.

É semelhante ao [cheiro de código de Intimidade Inadequada](#) ao implementar classes. Se dois microsserviços precisarem colaborar muito entre si, eles deverão provavelmente ser o mesmo microsserviço.

Outra forma de encarar isso é autonomia. Se um microsserviço precisar confiar em outro serviço para atender diretamente a uma solicitação, ele não será realmente autônomo.

## Camadas em microsserviços de DDD

A maioria dos aplicativos empresariais com complexidade de negócios e técnica significativa é definida por várias camadas. As camadas são um artefato lógico e não estão relacionadas à implantação do serviço. Elas existem para ajudar os desenvolvedores a gerenciar a complexidade no código. Camadas diferentes (como a camada de modelo de domínio versus a camada de apresentação etc.) podem ter tipos diferentes, o que exige conversões entre esses tipos.

Por exemplo, uma entidade pode ser carregada do banco de dados. Em seguida, parte dessas informações ou uma agregação de informações, incluindo dados adicionais de outras entidades, podem ser enviadas para uma interface do usuário do cliente por meio de uma API da Web REST. O ponto aqui é que a entidade de domínio está contida na camada de modelo de domínio e não deve ser propagada para outras áreas às quais ela não pertence, como a camada de apresentação.

Além disso, você precisa ter entidades sempre válidas (consulte a seção [Criar validações na camada de modelo de domínio](#)) controlada por raízes agregadas (entidades raiz). Portanto, as entidades não devem ser associadas aos modos de exibição do cliente pois, no nível da interface do usuário, alguns dados podem ainda não ter sido validados. É para isso que serve o ViewModel. O ViewModel é um modelo de dados exclusivamente para necessidades de camada de apresentação. As entidades de domínio não pertencem diretamente ao ViewModel. Em vez disso, você precisa converter entre entidades de domínio e ViewModels e vice-versa.

Ao lidar com complexidade, é importante ter um modelo de domínio controlado por raízes agregadas que garantam que todas as invariáveis e regras relacionadas àquele grupo de entidades (agregação) sejam executadas por meio de um único ponto de entrada ou porta, a raiz de agregação.

A Figura 7-5 mostra como um design em camadas é implementado no aplicativo eShopOnContainers.

## Layers in a Domain-Driven Design Microservice

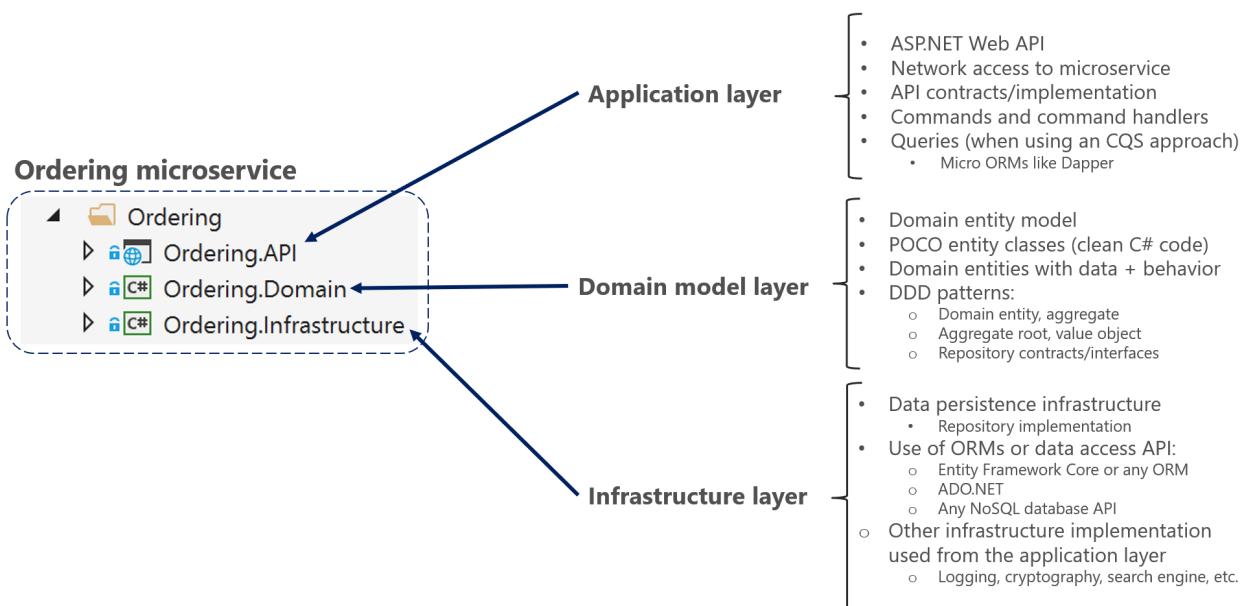
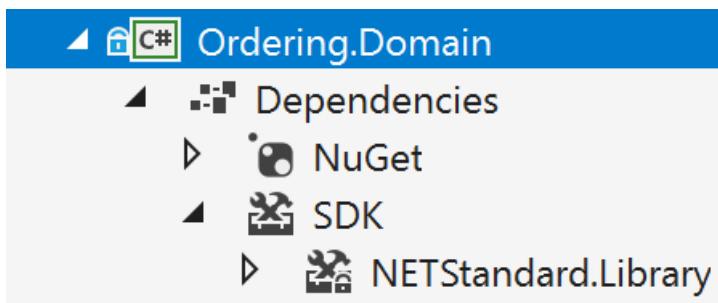


Figura 7-5. Camadas DDD no microsserviço de ordenação em eShopOnContainers

As três camadas em um microsserviço de DDD como o de pedidos. Cada camada é um projeto do VS: a camada de aplicativo é Ordering.API, a camada de domínio é Ordering.Domain e a camada de infraestrutura é Ordering.Infrastructure. Você deseja criar o sistema de modo que cada camada se comunique apenas com determinadas outras camadas. Isso poderá ser mais fácil de impor se as camadas forem implementadas como bibliotecas de classes diferentes, porque você pode identificar claramente que dependências são definidas entre bibliotecas. Por exemplo, a camada de modelo de domínio não deve receber uma dependência de nenhuma outra camada (as classes de modelo de domínio devem ser [POCO](#) ou Plain Old CLR Objects). Conforme mostrado na Figura 7-6, a biblioteca de camada **Ordering.Domain** tem dependências apenas de bibliotecas .NET Core ou pacotes NuGet, mas não de nenhuma outra biblioteca personalizada, como a biblioteca de dados ou a biblioteca de persistência.



**Figura 7-6.** Camadas implementadas como bibliotecas permitem melhor controle de dependências entre as camadas

### A camada de modelo de domínio

O excelente livro de Eric Evans, [Domain Driven Design](#) (Projeto Orientado a Domínio) diz o seguinte sobre a camada de modelo de domínio e a camada de aplicativo.

**Camada de modelo de domínio:** responsável por representar conceitos de negócios, informações sobre a situação de negócios e as regras de negócio. O estado que reflete a situação de negócios é controlado e usado aqui, embora os detalhes técnicos de armazená-lo sejam delegados à infraestrutura. Essa camada é a essência do software de negócios.

A camada de modelo de domínio é onde os negócios é expresso. Quando você implementa uma camada de modelo de domínio de microserviço em .NET, essa camada é codificada como uma biblioteca de classes com as entidades de domínio que capturam dados mais comportamento (métodos com lógica).

Seguindo os princípios de [Ignorância da Persistência](#) e a [Ignorância da Infraestrutura](#), essa camada deve ignorar totalmente os detalhes de persistência de dados. Essas tarefas de persistência devem ser executadas pela camada de infraestrutura. Portanto, essa camada não deveria receber dependências diretas da infraestrutura, o que significa que uma regra importante é que suas classes de entidade do modelo de domínio devem ser [POCOs](#).

Entidades de domínio não devem ter nenhuma dependência direta (como derivar de uma classe base) em nenhuma estrutura de infraestrutura de acesso a dados, como Entity Framework ou NHibernate. Idealmente, suas entidades de domínio não devem derivar nem implementar nenhum tipo definido em nenhuma estrutura de infraestrutura.

As estruturas ORM mais modernas, como Entity Framework Core, permitem essa abordagem, de modo que suas classes de modelo de domínio não estejam ligadas à infraestrutura. No entanto, ter entidades POCO nem sempre é possível ao usar determinados bancos de dados NoSQL e estruturas, como Atores e Coleções Confiáveis no Azure Service Fabric.

Mesmo quando é importante seguir o princípio de Ignorância de Persistência para o modelo de Domínio, você não deve ignorar preocupações de persistência. Ainda é muito importante entender o modelo de dados físicos e como ele é mapeado para o modelo de objeto de entidade. Caso contrário, você pode criar designs impossíveis.

Além disso, isso não significa que você pode pegar um modelo criado para um banco de dados relacional e movê-lo diretamente para um banco de dados orientado por documentos ou NoSQL. Em alguns modelos de entidade, o modelo pode se ajustar, mas geralmente não se ajusta. Ainda há restrições que o modelo de entidade deve cumprir, com base tanto na tecnologia de armazenamento quanto na tecnologia ORM.

### A camada de aplicativo

Passando para a camada de aplicativo, novamente podemos citar o livro de Eric Evans [Domain Driven Design](#) (Projeto Orientado a Domínio):

**Camada de aplicativo:** define os trabalhos que o software deve fazer e direciona os objetos de domínio expressivos para resolver problemas. As tarefas pelas quais esta camada é responsável são significativas para os negócios ou necessárias para a interação com as camadas do aplicativo de outros sistemas. Essa camada é mantida fina. Ele não contém regras de negócio nem conhecimento, mas apenas coordena o trabalho de tarefas e

delegados para colaborações de objetos de domínio na próxima camada abaixo. Ele não tem um estado refletindo a situação de negócios, mas pode ter um estado que reflita o progresso de uma tarefa para o usuário ou o programa.

A camada de aplicação de um microserviço no .NET é comumente codificada como um projeto de API da Web ASP.NET. O projeto implementa a interação do microserviço, o acesso à rede remota e as APIs externas da Web usadas a partir dos aplicativos de interface do cliente ou ui. Ele incluirá consultas se estiver usando uma abordagem CQRS, comandos aceitos pelo microsserviço e até mesmo a comunicação controlada por evento entre microsserviços (eventos de integração). A API Web do ASP.NET Core que representa a camada de aplicativo não deve conter as regras de negócio nem o conhecimento do domínio (especialmente regras de domínio para transações ou atualizações); isso deve ser de propriedade da biblioteca de classes de modelo de domínio. A camada do aplicativo deve apenas coordenar tarefas e não deve reter nem definir qualquer estado de domínio (modelo de domínio). Ela delega a execução de regras de negócio para as classes de modelo de domínio em si (raízes agregadas e entidades de domínio), que, por fim, atualizarão os dados dentro dessas entidades de domínio.

Basicamente, a “lógica de aplicativo” é onde você implementa todos os casos de uso que dependem de um determinado front-end. Por exemplo, a implementação relacionada a um serviço de API da Web.

A meta é que a lógica do domínio na camada de modelo de domínio, suas invariáveis, o modelo de dados e as regras de negócio relacionadas sejam completamente independentes das camadas de apresentação e do aplicativo. Principalmente, a camada de modelo de domínio deve não deve depender diretamente de nenhuma estrutura de infraestrutura.

### **A camada de infraestrutura**

A camada de infraestrutura é como os dados inicialmente mantidos em entidades de domínio (em memória) são mantidos em bancos de dados ou outro repositório persistente. Um exemplo é usar o código do Entity Framework Core para implementar as classes padrão do repositório que usam um DbContext para manter os dados em um banco de dados relacional.

De acordo com os princípios anteriormente mencionados [de ignorância](#) e ignorância [de infra-estrutura](#), a camada de infra-estrutura não deve “contaminar” a camada do modelo de domínio. Você deve manter as classes de entidade de modelo de domínio independentes da infraestrutura que você usa para manter os dados (EF ou qualquer outra estrutura) não obtendo dependências rígidas de estruturas. Sua biblioteca de classes de camada de modelo de domínio deve ter somente o código de domínio, apenas classes de entidade [POCO](#) implementando a essência do seu software e completamente separadas de tecnologias de infraestrutura.

Assim, suas camadas ou bibliotecas e projetos de classes devem, por fim, depender da sua camada de modelo de domínio (biblioteca), não vice-versa, conforme mostra a Figura 7-7.

# Dependencies between Layers in a Domain-Driven Design service

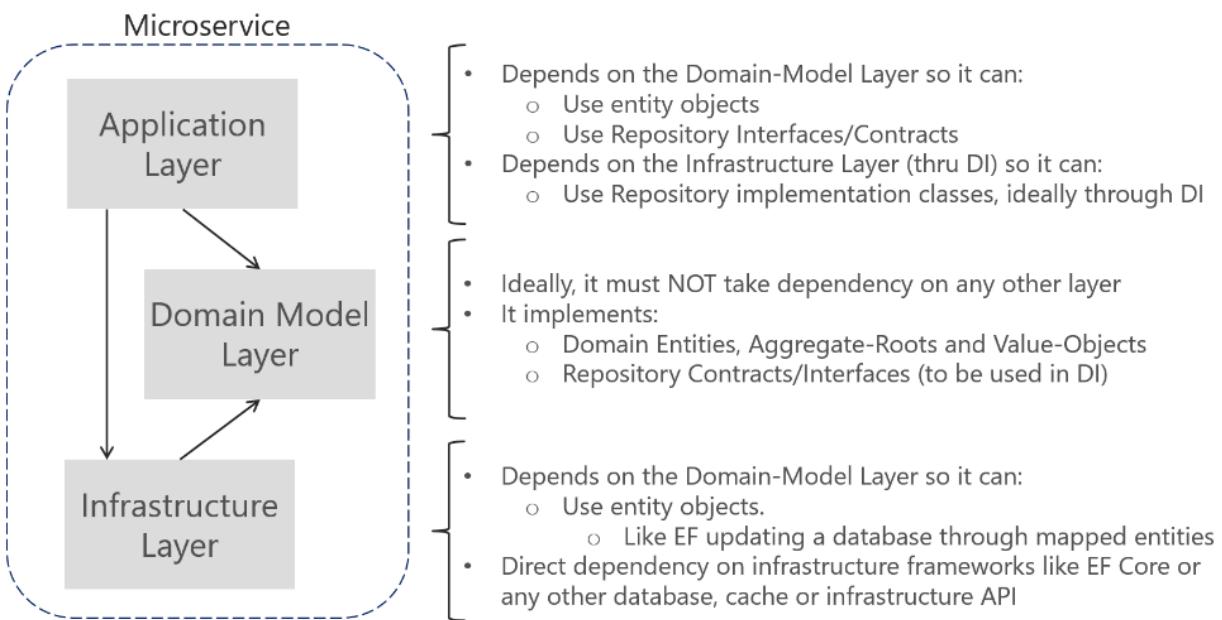


Figura 7-7. Dependências entre camadas em DDD

Dependências em um serviço de DDD, a camada de aplicativo depende do domínio e da infraestrutura e a infraestrutura depende do domínio, mas o domínio não depende de nenhuma camada. Esse design de camada deve ser independente de cada microserviço. Conforme observado anteriormente, você pode implementar os microserviços mais complexos seguindo padrões DDD ao mesmo tempo em que implementa microserviços conduzidos por dados mais simples (CRUD simples em uma única camada) de maneira mais simples.

## Recursos adicionais

- O DevIQ. Princípio da ignorância da persistência  
<https://deviq.com/persistence-ignorance/>
- Oren Eini. Ignorância da infra-estrutura  
<https://ayende.com/blog/3137/infrastructure-ignorance>
- Angel Lopez. Arquitetura em camadas em design orientado por domínio  
<https://ajlopez.wordpress.com/2008/09/12/layered-architecture-in-domain-driven-design/>

PRÓXIMO

ANTERIOR

# Projetar um modelo de domínio de microsserviço

10/09/2020 • 20 minutes to read • [Edit Online](#)

*Definir um modelo de domínio avançado para cada microsserviço empresarial ou Contexto Limitado.*

Sua meta é criar um único modelo de domínio coeso para cada microsserviço de negócios ou BC (Contexto Limitado). No entanto, tenha em mente que um microsserviço de BC ou de continuidade de negócios, às vezes, pode ser composto por vários serviços físicos que compartilham um único modelo de domínio. O modelo de domínio precisa capturar as regras, o comportamento, a linguagem de negócios e as restrições do único Contexto Limitado ou microsserviço de negócios que ele representa.

## O padrão de entidade de domínio

As entidades representam objetos de domínio e são definidas principalmente pela identidade, a continuidade e a persistência ao longo do tempo e não apenas pelos atributos que as compõem. Como Eric Evans diz, "um objeto definido principalmente por sua identidade é chamado de entidade". As entidades são muito importantes no modelo de domínio, pois elas são a base para um modelo. Portanto, você deve identificá-las e criá-las com cuidado.

*A identidade de uma entidade pode cruzar vários microsserviços ou contextos limitados.*

A mesma identidade (ou seja, o mesmo valor `Id`, embora talvez não a mesma entidade de domínio) pode ser modelada em vários Contextos Limitados ou microsserviços. No entanto, isso não significa que a mesma entidade, com os mesmos atributos e a mesma lógica, possa ser implementada em vários Contextos Limitados. Em vez disso, as entidades em cada contexto limitado limitam seus atributos e comportamentos aos necessários no domínio do contexto limitado.

Por exemplo, a entidade comprador pode ter a maioria dos atributos de uma pessoa que são definidos na entidade usuário no microsserviço de perfil ou de identidade, incluindo a identidade. Mas a entidade de comprador no microsserviço de pedidos pode ter menos atributos, porque somente determinados dados do comprador estão relacionados ao processo de pedido. O contexto de cada microsserviço ou o Contexto Limitado afeta o modelo de domínio.

*As entidades de domínio precisam implementar o comportamento, além de implementar os atributos de dados.*

Uma entidade de domínio no DDD precisa implementar a lógica do domínio ou o comportamento relacionado aos dados da entidade (o objeto acessado na memória). Por exemplo, como parte de uma classe de entidade de pedido, você precisa ter a lógica de negócios e as operações implementadas como métodos para tarefas como adicionar um item de pedido, validação de dados e cálculo de total. Os métodos da entidade cuidam das invariáveis e das regras da entidade, em vez de fazer com que essas regras se espalhem pela camada do aplicativo.

A Figura 7-8 mostra uma entidade de domínio que, além de implementar os atributos de dados, também implementa as operações ou os métodos com a lógica de domínio relacionada.

# Domain Entity pattern

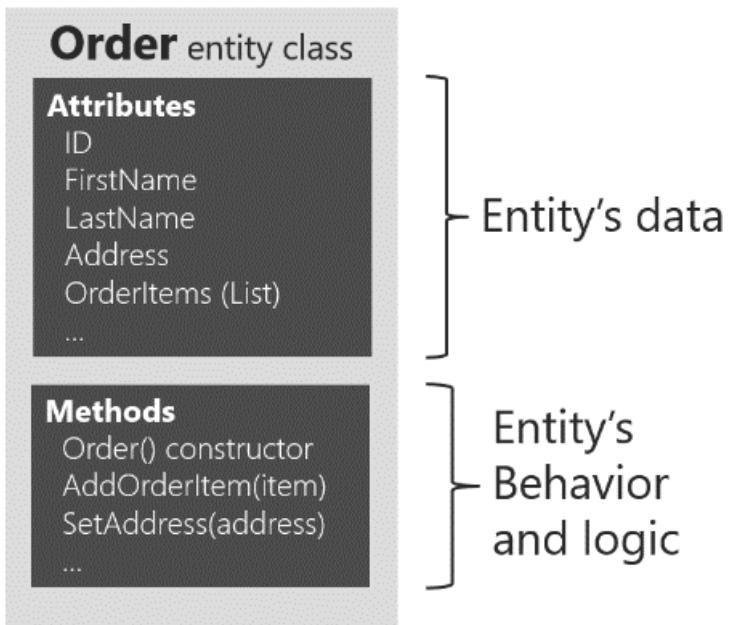


Figura 7-8. Exemplo de um projeto de entidade de domínio implementando dados e também comportamento

Uma entidade de modelo de domínio implementa comportamentos por meio de métodos, ou seja, não é um modelo "anêmico". Obviamente, também é possível haver entidades que não implementam nenhuma lógica como parte da classe da entidade. Isso poderá ocorrer em entidades filhas dentro de uma agregação se a entidade filha não tiver nenhuma lógica especial porque a maioria da lógica está definida na raiz da agregação. Se você tiver um microserviço complexo que tenha uma lógica implementada nas classes de serviço em vez de nas entidades de domínio, poderá estar se enquadrando no modelo de domínio anêmico, explicado na seção a seguir.

## Modelo de domínio avançado em comparação com o modelo de domínio anêmico

Em sua postagem [AnemicDomainModel](#), Martin Fowler descreve um modelo de domínio fraco desta forma:

O sintoma básico de um modelo de domínio anêmico é que, à primeira vista, ele parece real. Existem objetos, muitos nomeados de acordo com os nomes no espaço de domínio, conectados com as relações avançadas e a estrutura que os modelos de domínio verdadeiros têm. A verdade aparece quando você observa o comportamento e percebe que não há praticamente nenhum comportamento nesses objetos, ou seja, eles não passam de pacotes de getters e setters.

É claro que, quando você usar um modelo de domínio anêmico, serão usados os modelos de dados de um conjunto de objetos de serviço (tradicionalmente chamado de *camada comercial*) que captura toda a lógica de negócios ou do domínio. A camada de negócios fica acima do modelo de dados e usa o modelo de dados apenas como dados.

O modelo de domínio anêmico é apenas um design de estilo de procedimento. Os objetos da entidade anêmica não são objetos reais, porque eles não têm comportamento (métodos). Eles apenas mantêm as propriedades dos dados e, portanto, não são de design orientado a objeto. Ao colocar todo o comportamento em objetos de serviço (a camada de negócios), essencialmente você acaba com o [código espaguete](#) ou com os [scripts de transação](#), portanto, perde as vantagens que um modelo de domínio fornece.

De qualquer maneira, se o microsserviço ou o Contexto Limitado for muito simples (um serviço de CRUD), o modelo de domínio anêmico na forma de objetos de entidade apenas com as propriedades dos dados já poderá ser suficiente e talvez não compense implementar os padrões mais complexos de DDD. Nesse caso, ele será simplesmente um modelo de persistência, porque você criou uma entidade intencionalmente apenas com os dados para fins de CRUD.

É por isso que as arquiteturas de microsserviços são perfeitas para uma abordagem de várias arquiteturas, dependendo de cada Contexto Limitado. Por exemplo, no eShopOnContainers, o microsserviço de pedidos implementa os padrões de DDD, mas o microsserviço de catálogo, que é um serviço de CRUD simples, não implementa.

Algumas pessoas dizem que o modelo de domínio anêmico é um antipadrão. Isso realmente depende do que está sendo implementando. Se o microsserviço que você estiver criando for simples o suficiente (por exemplo, um serviço de CRUD), seguir o modelo de domínio anêmico não será um antipadrão. No entanto, se você precisar lidar com a complexidade do domínio de um microsserviço que tem muitas regras de negócio em constante mudança, o modelo de domínio anêmico pode ser um antipadrão para esse microsserviço ou contexto limitado. Nesse caso, criá-lo como um modelo avançado com entidades contendo dados e comportamentos, bem como implementar os padrões de DDD adicionais (agregações, objetos de valor, etc.) pode trazer enormes benefícios para o sucesso a longo prazo desse microsserviço.

#### Recursos adicionais

- **DevIQ. Entidade de domínio**  
<https://deviq.com/entity/>
- **Martin Fowler. O modelo de domínio**  
<https://martinfowler.com/eaaCatalog/domainModel.html>
- **Martin Fowler. O modelo de domínio anêmico**  
<https://martinfowler.com/bliki/AnemicDomainModel.html>

#### O padrão de objeto de valor

Como Eric Evans foi observado, "muitos objetos não têm identidade conceitual. Esses objetos descrevem determinadas características de algo."

Uma entidade requer uma identidade, mas há muitos objetos em um sistema que não requerem, como o padrão de objeto de valor. Um objeto de valor é um objeto sem nenhuma identidade conceitual que descreve um aspecto de domínio. Esses são objetos dos quais uma instância é criada para representar os elementos de design que interessam apenas temporariamente. Importa *o que* eles são e não *quem* são. Exemplos incluem números e cadeias de caracteres, mas também podem ser conceitos de nível superior como grupos de atributos.

Algo que é uma entidade em um microsserviço pode não ser uma entidade em outro microsserviço, porque no segundo caso, o Contexto Limitado pode ter um significado diferente. Por exemplo, um endereço em um aplicativo de comércio eletrônico pode não ter uma identidade, já que ele pode representar apenas um grupo de atributos do perfil do cliente para uma pessoa ou empresa. Nesse caso, o endereço deve ser classificado como um objeto de valor. No entanto, em um aplicativo de uma empresa utilitária de energia elétrica, o endereço do cliente pode ser importante para o domínio de negócios. Portanto, o endereço precisa ter uma identidade para que o sistema de cobrança possa ser vinculado diretamente ao endereço. Nesse caso, o endereço deve ser classificado como uma entidade de domínio.

Uma pessoa com um nome e sobrenome geralmente é uma entidade porque uma pessoa tem identidade, mesmo que o nome e sobrenome coincidam com outro conjunto de valores, como se esses nomes também se referirem a uma pessoa diferente.

Os objetos de valor são difíceis de gerenciar em bancos de dados relacionais e ORMs como Entity Framework (EF), enquanto em bancos de dados orientados a documentos eles são mais fáceis de implementar e usar.

EF Core 2,0 e versões posteriores incluem o recurso de [entidades de propriedade](#) que facilita o tratamento de objetos de valor, como veremos detalhadamente mais adiante.

#### Recursos adicionais

- **Martin Fowler. Padrão de objeto de valor**  
<https://martinfowler.com/bliki/ValueObject.html>

- **Objeto de valor**  
<https://deviq.com/value-object/>
- **Objetos de valor no desenvolvimento controlado por testes**  
<https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects>
- **Eric Evans. Design controlado por domínio: solução de complexidade no coração do software.**  
(Livro; inclui uma discussão sobre objetos de valor)  
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

## O padrão de agregação

Um modelo de domínio contém diferentes clusters de entidades de dados e processos que podem controlar uma área significativa de funcionalidades, como inventário ou execução de pedidos. Uma unidade de DDD mais refinada é a agregação, que descreve um cluster ou o grupo de entidades e os comportamentos que podem ser tratados como uma unidade coesa.

Normalmente, a agregação é definida com base nas transações que são necessárias. Um exemplo clássico é um pedido que também contém uma lista de itens do pedido. Geralmente, um item do pedido será uma entidade. Mas ela será uma entidade filha dentro da agregação de pedido, que também conterá a entidade de pedido como entidade raiz, geralmente chamada de raiz de agregação.

Pode ser difícil identificar agregações. Uma agregação é um grupo de objetos que precisam ser consistentes em conjunto, mas não basta apenas selecionar um grupo de objetos e rotulá-lo como uma agregação. Você precisa começar com um conceito de domínio e pensar sobre as entidades que são usadas nas transações mais comuns relacionadas a esse conceito. As entidades que precisam ser consistentes entre as transações são as que formam uma agregação. Pensar sobre as operações de transação provavelmente é a melhor maneira de identificar as agregações.

## O padrão de raiz de agregação ou de entidade raiz

Uma agregação é composta por pelo menos uma entidade: a raiz de agregação, também chamada de entidade raiz ou entidade principal. Além disso, ela pode ter várias entidades filhas e objetos de valor, com todas as entidades e os objetos trabalhando juntos para implementar as transações e os comportamentos necessários.

A finalidade de uma raiz de agregação é garantir a consistência da agregação. Ela deve ser o único ponto de entrada para as atualizações da agregação por meio de métodos ou operações na classe raiz de agregação. Você deve fazer alterações nas entidades na agregação apenas por meio da raiz de agregação. É o guardião da consistência da agregação, considerando todas as invariáveis e regras de consistência que você pode precisar obedecer em sua agregação. Se você alterar uma entidade filha ou um objeto de valor de forma independente, a raiz de agregação não poderá garantir que a agregação esteja em um estado válido. Isso seria semelhante a uma tabela com um segmento flexível. Manter a consistência é o principal objetivo da raiz de agregação.

Na Figura 7-9, veja as agregações de exemplo, como a agregação de comprador, que contém uma única entidade (a raiz de agregação de comprador). A agregação de pedido contém várias entidades e um objeto de valor.

# Aggregate pattern

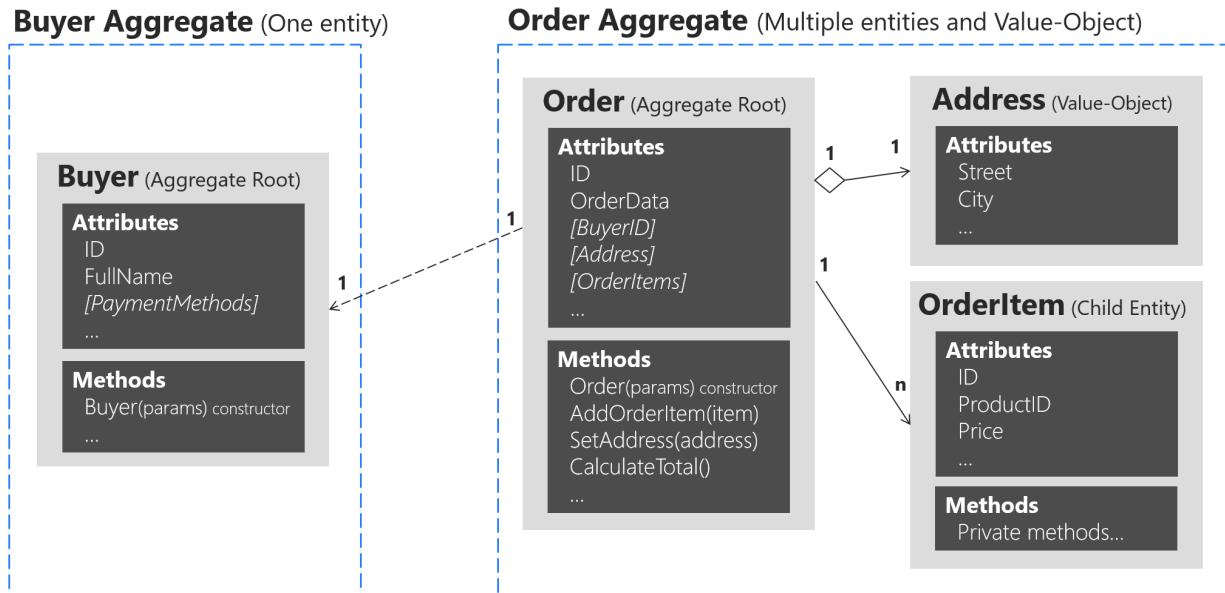


Figura 7-9. Exemplo de agregações com uma única entidade ou com várias entidades

Um modelo de domínio de DDD é composto por agregações, uma agregação pode ter uma única entidade ou mais e pode incluir também objetos de valor. Observe que a agregação de comprador poderá ter entidades filhas adicionais, dependendo do domínio, como ocorre no microsserviço de pedidos no aplicativo eShopOnContainers de referência. A Figura 7-9 apenas ilustra um caso em que o comprador tem uma única entidade, como exemplo de uma agregação que contém somente uma raiz de agregação.

Para manter a separação de agregações e manter limites claros entre elas, uma prática recomendada em um modelo de domínio de DDD é não permitir a navegação direta entre as agregações e ter apenas o campo de FK (chave estrangeira), como foi implementado no [Modelo de domínio do microsserviço de pedidos](#) no eShopOnContainers. A entidade Order tem apenas um campo de chave estrangeira para o comprador, mas não uma EF Core propriedade de navegação, conforme mostrado no código a seguir:

```
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId; // FK pointing to a different aggregate root
    public OrderStatus OrderStatus { get; private set; }
    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;
    // ... Additional code
}
```

Para identificar e trabalhar com agregações é necessário fazer pesquisas e ter experiência. Para obter mais informações, consulte a lista de Recursos adicionais a seguir.

## Recursos adicionais

- Vaughn Vernon. [Design agregado efetivo-parte I: modelando uma única agregação](#) (de <https://dddcommunity.org/>)  
[https://dddcommunity.org/wp-content/uploads/files/pdf\\_articles/Vernon\\_2011\\_1.pdf](https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf)
- Vaughn Vernon. [Design agregado efetivo-parte II: fazer com que as agregações funcionem juntas](#) (de <https://dddcommunity.org/>)  
[https://dddcommunity.org/wp-content/uploads/files/pdf\\_articles/Vernon\\_2011\\_2.pdf](https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf)
- Vaughn Vernon. [Design agregado efetivo-parte III: obtendo informações sobre a descoberta](#)

(de <https://dddcommunity.org/> )

[https://dddcommunity.org/wp-content/uploads/files/pdf\\_articles/Vernon\\_2011\\_3.pdf](https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_3.pdf)

- **Sergey Grybniak.** Padrões de design tático DDD  
<https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part>
- **Chris Richardson.** Desenvolvendo microserviços transacionais usando agregações  
<https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson>
- **DevIQ.** O padrão agregado  
<https://deviq.com/aggregate-pattern/>

[ANTERIOR](#)

[AVANÇAR](#)

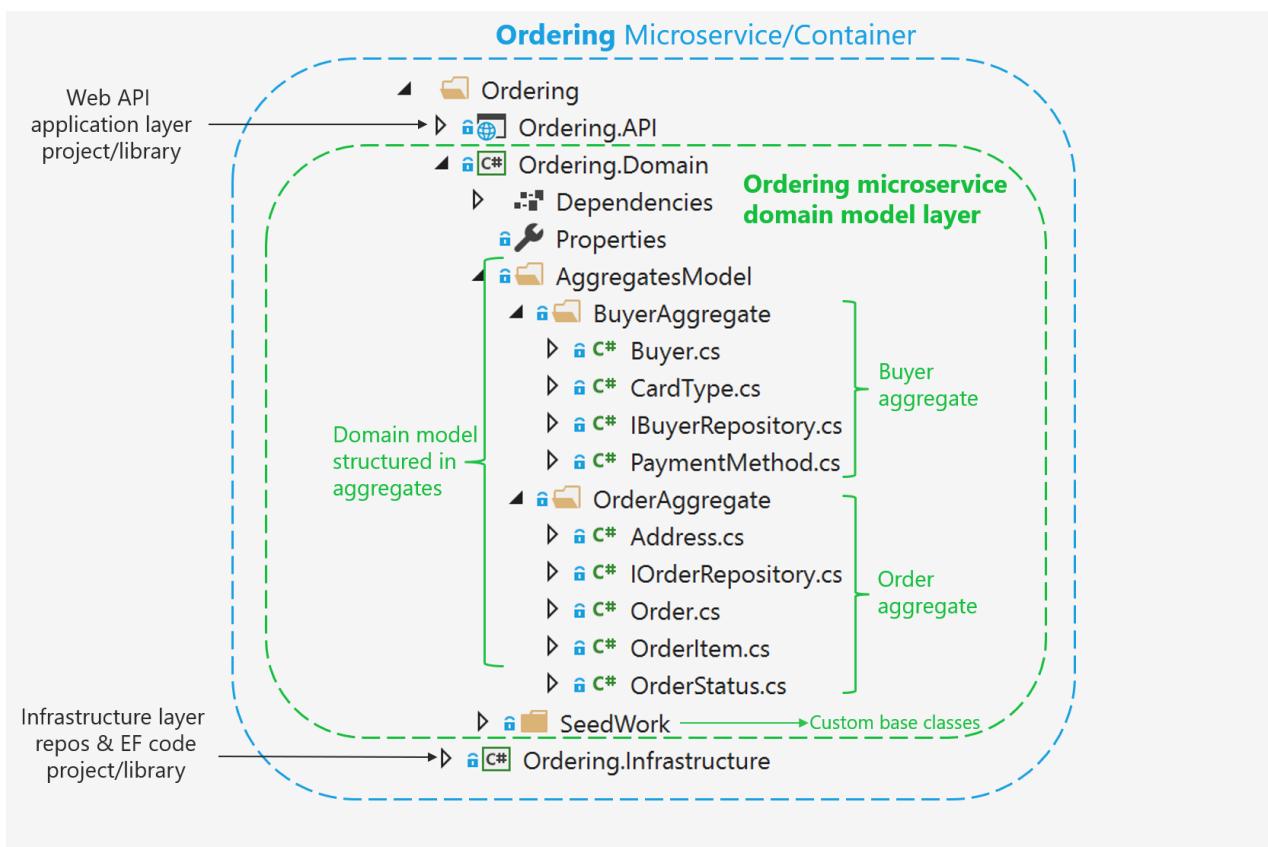
# Implementar um modelo de domínio de microsserviço com o .NET Core

10/09/2020 • 20 minutes to read • [Edit Online](#)

Na seção anterior, foram explicados os princípios de design fundamentais e os padrões para criar um modelo de domínio. Agora é hora de explorar possíveis maneiras de implementar o modelo de domínio usando o .NET Core (código C# simples) e EF Core. Seu modelo de domínio será composto simplesmente pelo seu código. Ele terá apenas os requisitos do modelo EF Core, mas não reais dependências do EF. Você não deve ter dependências rígidas nem referências ao EF Core ou qualquer outro ORM em seu modelo de domínio.

## Estrutura do modelo de domínio em uma biblioteca .NET Standard personalizada

A organização de pastas usada para o aplicativo de referência eShopOnContainers demonstra o modelo DDD para o aplicativo. Você pode considerar que uma organização de pastas diferente comunica mais claramente as escolhas de design feitas para o seu aplicativo. Como é possível ver na Figura 7-10, no modelo de domínio de ordenação, há duas agregações: a agregação de ordem e a agregação de comprador. Cada agregação é um grupo de entidades de domínio e objetos de valor, embora você possa ter uma agregação composta por uma única entidade de domínio (a raiz de agregação ou entidade raiz) também.



O Gerenciador de Soluções exibição para o projeto de ordenação. domínio, mostrando a pasta `AggregatesModel` que contém as pastas `BuyerAggregate` e `OrderAggregate`, cada uma contendo suas classes de entidade, arquivos de objeto de valor e assim por diante.

**Figura 7-10.** Estrutura de modelo de domínio para o microsserviço de ordenação em eShopOnContainers

Além disso, a camada de modelo de domínio inclui os contratos de repositório (interfaces) que são os requisitos

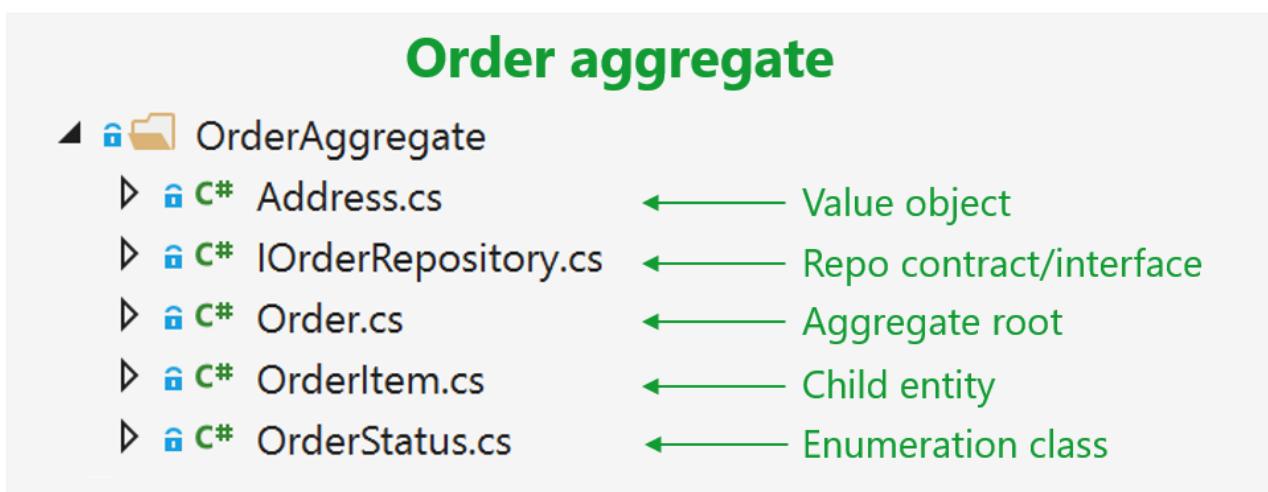
de infraestrutura do seu modelo de domínio. Em outras palavras, essas interfaces expressam quais repositórios e os métodos que a camada de infraestrutura deve implementar. É fundamental que a implementação dos repositórios seja colocada fora da camada de modelo de domínio, na biblioteca de camadas de infraestrutura, de modo que a camada de modelo de domínio não seja "contaminada" por API ou classes de tecnologias de infraestrutura, como Entity Framework.

Você também pode ver uma pasta de tarefa de [propagação](#) que contém classes base personalizadas que você pode usar como base para suas entidades de domínio e objetos de valor, para que você não tenha código redundante na classe de objeto de cada domínio.

## Estruturar agregações em uma biblioteca .NET Standard personalizada

Uma agregação refere-se a um cluster de objetos de domínio agrupados de acordo com a consistência transacional. Esses objetos podem ser instâncias de entidades (uma das quais é a raiz de agregação ou uma entidade raiz) além de qualquer objeto de valor adicional.

Consistência transacional significa que uma agregação está garantidamente consistente e atualizada ao final de uma ação de negócios. Por exemplo, a agregação de ordem do modelo de domínio de microsserviços de pedidos eShopOnContainers é composta conforme mostra a Figura 7-11.



Uma exibição detalhada da pasta OrderAggregate: Address.cs é um objeto de valor, IOrderRepository é uma interface de repositório, Order.cs é uma raiz de agregação, OrderItem.cs é uma entidade filho e OrderStatus.cs é uma classe de enumeração.

Figura 7-11. A agregação de ordem na solução do Visual Studio

Se você abrir qualquer um dos arquivos em uma pasta de agregação, verá como ele está marcado como classe base personalizada ou interface, como objeto de entidade ou valor, conforme implementado na pasta [SeedWork](#).

## Implementar entidades de domínio como classes POCO

Você implementa um modelo de domínio no .NET criando classes POCO que implementam suas entidades de domínio. No exemplo a seguir, a classe Ordem é definida como uma entidade e como uma raiz de agregação. Porque a classe Ordem deriva da classe base da Entidade, ela pode reutilizar o código comum relacionado a entidades. Tenha em mente que essas interfaces e classes base são definidas por você no projeto de modelo de domínio, portanto, é o seu código, não o código de infraestrutura de um ORM, como EF.

```

// COMPATIBLE WITH ENTITY FRAMEWORK CORE 2.0
// Entity is a custom base class with the ID
public class Order : Entity, IAggregateRoot
{
    private DateTime _orderDate;
    public Address Address { get; private set; }
    private int? _buyerId;

    public OrderStatus OrderStatus { get; private set; }
    private int _orderStatusId;

    private string _description;
    private int? _paymentMethodId;

    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    public Order(string userId, Address address, int cardTypeId, string cardNumber, string cardSecurityNumber,
                string cardHolderName, DateTime cardExpiration, int? buyerId = null, int? paymentMethodId = null)
    {
        _orderItems = new List<OrderItem>();
        _buyerId = buyerId;
        _paymentMethodId = paymentMethodId;
        _orderStatusId = OrderStatus.Submitted.Id;
        _orderDate = DateTime.UtcNow;
        Address = address;

        // ...Additional code ...
    }

    public void AddOrderItem(int productId, string productName,
                            decimal unitPrice, decimal discount,
                            string pictureUrl, int units = 1)
    {
        //...
        // Domain rules/logic for adding the OrderItem to the order
        // ...

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

        _orderItems.Add(orderItem);
    }
    // ...
    // Additional methods with domain rules/logic related to the Order aggregate
    // ...
}

```

É importante observar que essa é uma entidade de domínio implementada como uma classe POCO. Ela não tem nenhuma dependência direta do Entity Framework Core nem qualquer outra estrutura de infraestrutura. Essa implementação é como deve estar no DDD, apenas código C# que implementa um modelo de domínio.

Além disso, a classe é decorada com uma interface denominada `IAggregateRoot`. Essa interface é uma interface vazia, às vezes chamada de uma *interface de marcador*, que é usada apenas para indicar que essa classe de entidade também é uma raiz de agregação.

Uma interface de marcador às vezes é considerada como um antipadrão; no entanto, também é uma maneira simples de marcar uma classe, especialmente quando essa interface pode estar em evolução. Um atributo pode ser outra escolha para o marcador, mas é mais rápido ver a classe base (`Entity`) ao lado da interface `IAggregate`, em vez de colocar um marcador de atributo `Aggregate` acima da classe. É uma questão de preferências, de qualquer forma.

Ter uma raiz agregada significa que a maior parte do código relacionado às regras de consistência e de negócios

das entidades da agregação deve ser implementada como métodos na classe raiz de ordem agregada (por exemplo, AddOrderItem ao adicionar um objeto OrderItem à agregação). Você não deve criar nem atualizar objetos OrderItems de modo independente ou direto; a classe AggregateRoot deve manter o controle e a consistência de qualquer operação de atualização com relação às suas entidades filho.

## Encapsular dados nas Entidades de Domínio

Um problema comum em modelos de entidade é que eles expõem propriedades de navegação da coleção como tipos de lista publicamente acessíveis. Isso permite que qualquer desenvolvedor do colaborador manipule o conteúdo desses tipos de coleção, o que pode ignorar importantes regras de negócio relacionadas à coleção, possivelmente deixando o objeto em um estado inválido. A solução para isso é expor o acesso somente leitura a coleções relacionadas e fornecer explicitamente métodos que definem maneiras como os clientes podem manipulá-los.

No código anterior, observe que muitos atributos são somente leitura ou privados e só são atualizáveis pelos métodos de classe, assim, qualquer atualização considera invariáveis de domínio empresarial da conta e lógica especificada dentro do método de classe.

Por exemplo, após os padrões DDD, você **não deve fazer o seguinte** de nenhum método de manipulador de comando ou classe de camada de aplicativo (na verdade, deve ser impossível você fazer isso):

```
// WRONG ACCORDING TO DDD PATTERNS - CODE AT THE APPLICATION LAYER OR
// COMMAND HANDLERS
// Code in command handler methods or Web API controllers
//... (WRONG) Some code with business logic out of the domain classes ...
OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName,
    pictureUrl, unitPrice, discount, units);

//... (WRONG) Accessing the OrderItems collection directly from the application layer // or command handlers
myOrder.OrderItems.Add(myNewOrderItem);
//...
```

Neste caso, o método Add é meramente uma operação para adicionar dados, com acesso direto à coleção OrderItems. Portanto, a maioria da lógica, das regras ou das validações de domínio relacionada àquela operação com as entidades filho será espalhada pela camada de aplicativo (manipuladores de comandos e controladores de API da Web).

Se você usar uma solução alternativa para a raiz de agregação, a raiz de agregação não assegurará suas invariáveis, sua validade nem sua consistência. Por fim, você terá código espaguete ou código de script transacional.

Para seguir padrões DDD, as entidades não devem ter setters públicos em nenhuma propriedade de entidade. Alterações em uma entidade devem ser controladas pelos métodos explícitos com linguagem ubíqua explícita sobre a alteração que estão sendo executadas na entidade.

Além disso, coleções na entidade (como os itens do pedido) devem ser propriedades somente leitura (o método AsReadOnly explicado posteriormente). Você deve ser capaz de atualizá-lo somente de dentro de métodos da classe raiz agregada ou de métodos de entidade filho.

Como você pode ver no código da raiz de agregação Order, todos os setters devem ser privados ou, pelo menos, somente leitura externamente, para que qualquer operação em relação aos dados da entidade ou suas entidades filho precise ser executada por meio de métodos na classe de entidade. Isso mantém a consistência de maneira controlada e orientada a objeto, em vez de implementar o código de script transacional.

O snippet de código a seguir mostra a maneira adequada de codificar a tarefa de adicionar um objeto OrderItem para a agregação de ordem de código.

```

// RIGHT ACCORDING TO DDD--CODE AT THE APPLICATION LAYER OR COMMAND HANDLERS
// The code in command handlers or WebAPI controllers, related only to application stuff
// There is NO code here related to OrderItem object's business logic
myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units);

// The code related to OrderItem params validations or domain rules should
// be WITHIN the AddOrderItem method.

//...

```

Neste snippet, a maioria das validações ou da lógica relacionada à criação de um objeto OrderItem estará sob o controle da raiz de agregação de Ordem (no método AddOrderItem), especialmente validações e lógica relacionadas a outros elementos no agregado. Por exemplo, você pode obter o mesmo item de produto como resultado de várias chamadas para AddOrderItem. Nesse método, você pode examinar os itens de produto e consolidar os mesmos itens de produto em um único objeto OrderItem com várias unidades. Além disso, se houver valores de desconto diferentes, mas a ID do produto (product ID) for a mesma, você provavelmente aplicará o desconto maior. Esse princípio se aplica a qualquer outra lógica do domínio para o objeto OrderItem.

Além disso, a nova operação OrderItem(params) também será controlada e executada pelo método AddOrderItem da raiz de agregação de Ordem. Portanto, a maioria da lógica ou das validações relacionadas àquela operação (especialmente tudo o que afeta a consistência entre outras entidades filho) estará em um único lugar na raiz da agregação. Esta é a principal finalidade do padrão de raiz de agregação.

Quando você usar o Entity Framework Core 1.1 ou posterior, uma entidade DDD pode ser melhor expressada porque ela permite [mapear para os campos](#) além das propriedades. Isso é útil ao proteger as coleções de entidades filho ou objetos de valor. Com esse aprimoramento, você pode usar os campos privados, em vez das propriedades, e pode implementar qualquer atualização à coleção de campo nos métodos públicos e fornecer acesso somente leitura por meio do método AsReadOnly.

No DDD, você deseja atualizar a entidade somente por meio de métodos na entidade (ou do Construtor) para controlar qualquer invariável e a consistência dos dados, para que as propriedades sejam definidas somente com um acessador get. As propriedades têm o respaldo de campos privados. Membros privados só pode ser acessados de dentro da classe. No entanto, há uma exceção: o EF Core precisa definir esses campos também (para poder retornar o objeto com os valores adequados).

### **Mapear propriedades com apenas acessadores get para os campos na tabela de banco de dados**

Mapeamento de propriedades para colunas de tabela do banco de dados não é uma responsabilidade do domínio, mas parte da camada de infraestrutura e persistência. Mencionamos isso aqui apenas para que você esteja ciente das novas funcionalidades no EF Core 1.1 ou posterior relacionadas a como você pode modelar entidades. Detalhes adicionais sobre este tópico são explicados na seção de infraestrutura e persistência.

Quando você usa o EF Core 1.0 ou posterior, no DbContext, precisa mapear as propriedades definidas somente com getters para os campos reais na tabela de banco de dados. Isso é feito com o método HasField da classe PropertyBuilder.

### **Mapear campos sem propriedades**

Com o recurso no EF Core 1.1 ou posterior para mapear colunas para campos, também é possível não usar propriedades. Em vez disso, você pode apenas mapear as colunas de uma tabela para campos. Um caso de uso comum para isso são campos privados para um estado interno que não precisam ser acessados de fora da entidade.

Por exemplo, no exemplo de código anterior, OrderAggregate, há vários campos privados, como o campo `_paymentMethodId`, que não têm nenhuma propriedade relacionada para um setter ou um getter. Esse campo também pode ser calculado dentro da lógica de negócios do pedido e usado a partir dos métodos do pedido, mas ele também precisa ser persistido no banco de dados. Assim, no EF Core (desde a v1.1), existe uma maneira de mapear um campo sem uma propriedade relacionada a uma coluna no banco de dados. Isso também é explicado

na seção [Camada de infraestrutura](#) deste guia.

## Recursos adicionais

- **Vaughn Vernon.** A modelagem agrega com DDD e Entity Framework. Observe que este *não* é um Entity Framework Core.  
<https://kalele.io/blog-posts/modeling-aggregates-with-ddd-and-entity-framework/>
- **Julie Lerman.** Pontos de dados-codificação para o design controlado por domínio: dicas para desenvolvedores focados em dados  
[/archive/msdn-magazine/2013/august/data-points-coding-for-domain-driven-design-tips-for-data-focused-devs](https://archive/msdn-magazine/2013/august/data-points-coding-for-domain-driven-design-tips-for-data-focused-devs)
- **Udi Dahan.** Como criar modelos de domínio totalmente encapsulados  
<https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>

[ANTERIOR](#)

[AVANÇAR](#)

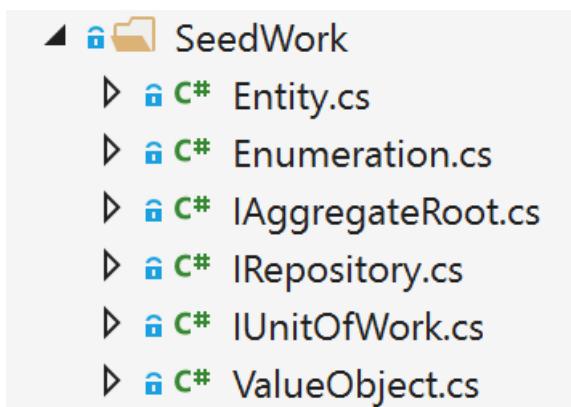
# Seedwork (classes e interfaces base reutilizáveis para seu modelo de domínio)

09/04/2020 • 6 minutes to read • [Edit Online](#)

A pasta de solução contém uma pasta *SeedWork*. Essa pasta contém classes base personalizadas que podem ser usadas como uma base para suas entidades de domínio e objetos de valor. Use essas classes base para que você não tenha código redundante na classe de objeto de cada domínio. A pasta para esses tipos de classes é chamada *SeedWork* e não algo como *Framework*. Chama-se *SeedWork* porque a pasta contém apenas um pequeno subconjunto de classes reutilizáveis que não podem realmente ser consideradas uma estrutura.

*Seedwork* é um termo introduzido por [Michael Feathers](#) e popularizado por [Martin Fowler](#), mas você também pode nomear essa pasta Common, SharedKernel, ou semelhantes.

A figura 7-12 mostra as classes que formam o seedwork do modelo de domínio no microsserviço de ordenação. Ele tem algumas classes base personalizadas como Entity, ValueObject e Enumeration, além de algumas interfaces. Essas interfaces ( IRepository e IUnitOfWork) informam a camada de infraestrutura sobre o que precisa ser implementado. Essas interfaces também são usadas por meio de Injeção de dependência da camada de aplicativo.



O conteúdo detalhado da pasta SeedWork, contendo classes e interfaces básicas: Entity.cs, Enumeration.cs, IAggregateRoot.cs, IRepository.cs, IUnitOfWork.cs e ValueObject.cs.

**Figura 7-12.** Um conjunto de exemplo de classes e interfaces base do modelo de domínio "seedwork"

Esse é o tipo de reutilização de copiar e colar que muitos desenvolvedores compartilham entre projetos, não uma estrutura formal. É possível ter seedworks em qualquer camada ou biblioteca. No entanto, se o conjunto de classes e interfaces ficar grande o suficiente, você pode querer criar uma única biblioteca de classes.

## A classe base Entity personalizada

O código a seguir é um exemplo de uma classe base Entity em que é possível inserir o código que pode ser usado da mesma maneira por qualquer entidade de domínio, como a ID da entidade, [operadores de igualdade](#), uma lista de eventos de domínio por entidade, etc.

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE (1.1 and later)
public abstract class Entity
{
    int? _requestedHashCode;
    int _Id;
    private List<INotification> _domainEvents;
    public virtual int Id
    {
        get
        {
            return _Id;
        }
        set
        {
            if (value != _Id)
            {
                _requestedHashCode = null;
                _domainEvents.Add(new EntityDomainEvent(this, value));
                _Id = value;
            }
        }
    }
}
```

```

        {
            return _Id;
        }
        protected set
        {
            _Id = value;
        }
    }

    public List<INotification> DomainEvents => _domainEvents;
    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }
    public void RemoveDomainEvent(INotification eventItem)
    {
        if (_domainEvents is null) return;
        _domainEvents.Remove(eventItem);
    }

    public bool IsTransient()
    {
        return this.Id == default(Int32);
    }

    public override bool Equals(object obj)
    {
        if (obj == null || !(obj is Entity))
            return false;
        if (Object.ReferenceEquals(this, obj))
            return true;
        if (this.GetType() != obj.GetType())
            return false;
        Entity item = (Entity)obj;
        if (item.IsTransient() || this.IsTransient())
            return false;
        else
            return item.Id == this.Id;
    }

    public override int GetHashCode()
    {
        if (!IsTransient())
        {
            if (!_requestedHashCode.HasValue)
                _requestedHashCode = this.Id.GetHashCode() ^ 31;
            // XOR for random distribution. See:
            // https://docs.microsoft.com/archive/blogs/ericlippert/guidelines-and-rules-for-gethashcode
            return _requestedHashCode.Value;
        }
        else
            return base.GetHashCode();
    }
    public static bool operator ==(Entity left, Entity right)
    {
        if (Object.Equals(left, null))
            return (Object.Equals(right, null));
        else
            return left.Equals(right);
    }
    public static bool operator !=(Entity left, Entity right)
    {
        return !(left == right);
    }
}

```

O código anterior que usa uma lista de eventos de domínio por entidade será explicado nas próximas seções ao abordar eventos de domínio.

## Contratos de repositório (interfaces) na camada de modelo de domínio

Os contratos de repositório são simplesmente interfaces .NET que expressam os requisitos do contrato dos repositórios a serem usados para cada agregação.

Os repositórios em si, com código do EF Core ou quaisquer outras dependências de infraestrutura e código (LINQ, SQL, etc.), não devem ser implementados no modelo de domínio; os repositórios só deverão implementar as interfaces que você definir no modelo de domínio.

Um padrão relacionado a essa prática (inserir as interfaces de repositório na camada do modelo de domínio) é o padrão Interface separada. Como [explicado](#) por Martin Fowler, "Use interface separada para definir uma interface em um pacote, mas implementá-la em outro. Dessa forma, um cliente que precisa da dependência da interface pode desconhecer completamente a implementação."

Seguir o padrão Interface separada permite que a camada de aplicativo (nesse caso, o projeto da API Web para o microsserviço) tenha uma dependência nos requisitos definidos no modelo de domínio, mas não uma dependência direta com a camada de infraestrutura/persistência. Além disso, é possível usar a Injeção de dependência para isolar a implementação, o que é implementado na camada de infraestrutura/persistência que usam repositórios.

Por exemplo, o exemplo a seguir com a interface `IOrderRepository` define quais operações a classe `OrderRepository` precisará implementar na camada de infraestrutura. Na implementação atual do aplicativo, o código precisa apenas adicionar ou atualizar ordens no banco de dados, uma vez que as consultas são divididas seguindo a abordagem CQRS simplificada.

```
// Defined at IOrderRepository.cs
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);

    void Update(Order order);

    Task<Order> GetAsync(int orderId);
}

// Defined at IRepository.cs (Part of the Domain Seedwork)
public interface IRepository<T> where T : IAggregateRoot
{
    IUnitOfWork UnitOfWork { get; }
}
```

## Recursos adicionais

- **Martin Fowler. Interface separada.**  
<https://www.martinfowler.com/eaaCatalog/separatedInterface.html>

[PRÓXIMO](#)

[ANTERIOR](#)

# Implementar objetos de valor

10/09/2020 • 20 minutes to read • [Edit Online](#)

Conforme discutido nas seções anteriores sobre entidades e agregações, a identidade é fundamental para entidades. No entanto, existem muitos objetos e itens de dados em um sistema que não exigem uma identidade e um acompanhamento de identidade, como objetos de valor.

Um objeto de valor pode fazer referência a outras entidades. Por exemplo, em um aplicativo que gera uma rota que descreve como ir de um ponto para outro, essa rota deve ser um objeto de valor. Seria um instantâneo de pontos em uma rota específica, mas essa rota sugerida não teria uma identidade, embora internamente possa fazer referência a entidades, como Cidade, Estrada etc.

A Figura 7-13 mostra o objeto de valor Endereço dentro da agregação Ordem.

## Value Object within Aggregate

### Order Aggregate (Multiple entities and Value-Object)

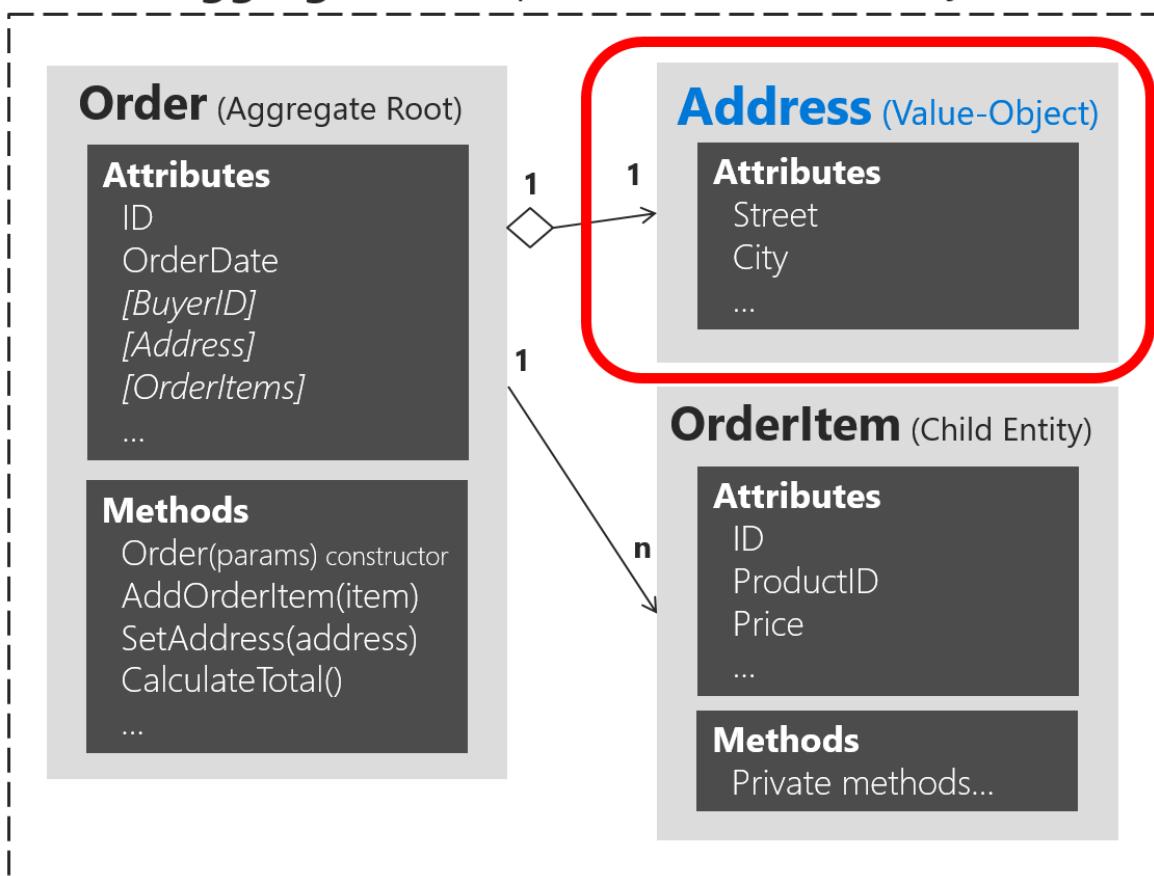


Figura 7-13. Tratar objeto de valor no agregado de Ordem

Conforme mostrado na Figura 7-13, uma entidade geralmente é composta por vários atributos. Por exemplo, a `Order` entidade pode ser modelada como uma entidade com uma identidade e composta internamente por um conjunto de atributos, como OrderID, OrderDate, OrderItems, etc. Mas o endereço, que é simplesmente um valor complexo composto de país/região, rua, cidade etc. e não tem nenhuma identidade nesse domínio, deve ser

modelado e tratado como um objeto de valor.

## Características importantes de objetos de valor

Existem duas características principais para objetos de valor:

- Eles não têm identidade.
- Eles são imutáveis.

A primeira característica já foi discutida. A imutabilidade é um requisito importante. Os valores de um objeto de valor devem ser imutáveis depois que o objeto for criado. Portanto, quando o objeto é construído, você deve fornecer os valores necessários, mas não deve permitir que eles sejam alterados durante o tempo de vida do objeto.

Objetos de valor permitem que você execute certos truques para desempenho graças à sua natureza imutável. Isso é especialmente verdadeiro em sistemas em que pode haver milhares de instâncias de objeto de valor, muitas das quais com os mesmos valores. Sua natureza imutável permite que sejam reutilizados; eles podem ser objetos intercambiáveis, desde que seus valores sejam os mesmos e não tenham identidade. Esse tipo de otimização às vezes pode ser a diferença entre o software executado lentamente e o software com bom desempenho. Obviamente, todos esses casos dependem do ambiente do aplicativo e do contexto de implantação.

## Implementação de objeto de valor em C#

Em termos de implementação, você pode ter uma classe base de objeto de valor que tenha métodos de utilitário básicos como igualdade com base na comparação entre todos os atributos (já que um objeto de valor não deve ser baseado em identidade) e outras características fundamentais. O exemplo a seguir mostra uma classe base do objeto de valor usada no microsserviço de ordenação de eShopOnContainers.

```

public abstract class ValueObject
{
    protected static bool EqualOperator(ValueObject left, ValueObject right)
    {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
        {
            return false;
        }
        return ReferenceEquals(left, null) || left.Equals(right);
    }

    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }

    protected abstract IEnumerable<object> GetEqualityComponents();

    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
        }

        var other = (ValueObject)obj;

        return this.GetEqualityComponents().SequenceEqual(other.GetEqualityComponents());
    }

    public override int GetHashCode()
    {
        return GetEqualityComponents()
            .Select(x => x != null ? x.GetHashCode() : 0)
            .Aggregate((x, y) => x ^ y);
    }
    // Other utility methods
}

```

Você pode usar essa classe durante a implementação do seu objeto de valor real, assim como acontece com o objeto de valor Endereço mostrado no exemplo a seguir:

```

public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }

    public Address() { }

    public Address(string street, string city, string state, string country, string zipcode)
    {
        Street = street;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;
    }

    protected override IEnumerable<object> GetEqualityComponents()
    {
        // Using a yield return statement to return each element one at a time
        yield return Street;
        yield return City;
        yield return State;
        yield return Country;
        yield return ZipCode;
    }
}

```

Você pode ver como essa implementação de objeto de valor do Endereço não tem nenhuma identidade e, portanto, nenhum campo de ID, nem na classe Address, nem mesmo na classe ValueObject.

Não foi possível ter nenhum campo de ID em uma classe a ser usado pelo Entity Framework (EF) até EF Core 2,0, o que ajuda muito a implementar objetos de valor melhores sem ID. Essa é justamente a explicação da próxima seção.

Pode ser argumentado que os objetos de valor, sendo imutáveis, devem ser somente leitura (ou seja, têm propriedades somente obtenção) e isso é verdade. No entanto, os objetos de valor geralmente são serializados e desserializados para passar pelas filas de mensagens e ser somente leitura interrompe o desserializador de atribuir valores, de modo que você apenas os deixa como `private set`, o que é somente leitura o suficiente para ser prático.

## Como persistir objetos de valor no banco de dados com EF Core 2,0 e posterior

Você acabou de ver como definir um objeto de valor em seu modelo de domínio. Mas como você pode realmente mantê-lo no banco de dados usando Entity Framework Core, já que ele normalmente visa entidades com identidade?

### Tela de fundo e abordagens mais antigas usando EF Core 1.1

Como segundo plano, uma limitação ao usar EF Core 1,0 e 1,1 era que você não podia usar [tipos complexos](#), conforme definido no EF 6.x na .NET Framework tradicional. Portanto, se estiver usando EF Core 1.0 ou 1.1, precisará armazenar seu objeto de valor como uma entidade EF com um campo de ID. Então, para que se pareça mais com um objeto de valor sem nenhuma identidade, você pode ocultar a ID para deixar claro que a identidade de um objeto de valor não é importante no modelo de domínio. Você pode ocultar essa ID usando a ID como um [propriedade de sombra](#). Uma vez que a configuração para ocultar a ID do modelo é configurada no nível de infraestrutura do EF, seria transparente para o seu modelo de domínio.

Na versão inicial do eShopOnContainers (.NET Core 1.1), a ID oculta necessária para a infraestrutura do EF Core foi implementada da seguinte maneira no nível de DbContext, usando a API Fluente no projeto de infraestrutura. Portanto, a ID foi oculta do ponto de vista do domínio modelo de domínio, mas ainda está na infraestrutura.

```
// Old approach with EF Core 1.1
// Fluent API within the OrderingContext:DbContext in the Infrastructure project
void ConfigureAddress(EntityTypeBuilder<Address> addressConfiguration)
{
    addressConfiguration.ToTable("address", DEFAULT_SCHEMA);

    addressConfiguration.Property<int>("Id") // Id is a shadow property
        .IsRequired();
    addressConfiguration.HasKey("Id"); // Id is a shadow property
}
```

No entanto, a persistência desse objeto de valor no banco de dados foi executada como uma entidade normal em uma tabela diferente.

Com o EF Core 2,0 e posterior, há novas e melhores maneiras de persistir objetos de valor.

## Persistir objetos de valor como tipos de entidade pertencentes no EF Core 2,0 e posterior

Mesmo com algumas lacunas entre o padrão de objeto de valor canônico em DDD e o tipo de entidade de propriedade no EF Core, atualmente, é a melhor maneira de persistir objetos de valor com o EF Core 2,0 e posterior. Você pode ver as limitações no final desta seção.

O recurso de tipo de entidade própria foi adicionado ao EF Core desde a versão 2.0.

Um tipo de entidade própria permite mapear tipos que não têm a própria identidade definida explicitamente no modelo de domínio e são usados como propriedades, como um objeto de valor, em uma de suas entidades. Um tipo de entidade de propriedade compartilha o mesmo tipo CLR com outro tipo de entidade (ou seja, é apenas uma classe regular). A entidade que contém a navegação de definição é a entidade proprietária. Ao consultar o proprietário, os tipos próprios serão incluídos por padrão.

Apenas olhando o modelo de domínio, um tipo de propriedade parece que ele não tem nenhuma identidade. No entanto, nos bastidores, os tipos de propriedade têm a identidade, mas a propriedade de navegação do proprietário faz parte dessa identidade.

A identidade de instâncias de tipos próprios não é completamente própria. Consiste em três componentes:

- A identidade do proprietário
- A propriedade de navegação apontando para elas
- No caso de coleções de tipos de propriedade, um componente independente (com suporte no EF Core 2,2 e posterior).

Por exemplo, no modelo de domínio Ordenação em eShopOnContainers, como parte da entidade de Ordem, o objeto de valor de endereço é implementado como um tipo de entidade própria dentro da entidade de proprietário, que é a entidade Ordem. `Address` é um tipo sem nenhuma propriedade de identidade definida no modelo de domínio. Ele é usado como uma propriedade do tipo Ordem para especificar o endereço para entrega para uma ordem específica.

Por convenção, uma chave primária de sombra será criada para o tipo próprio e será mapeada para a mesma tabela que a do proprietário usando a divisão de tabela. Isso permite usar tipos próprios de modo similar a como os tipos complexos são usados no EF6 no .NET Framework tradicional.

É importante observar que tipos próprios nunca são descobertos pela convenção no núcleo do EF, assim, você precisa declará-los explicitamente.

No eShopOnContainers, no arquivo OrderingContext.cs, dentro do `OnModelCreating()` método, várias configurações de infraestrutura são aplicadas. Um deles está relacionado à entidade Ordem.

```
// Part of the OrderingContext.cs class at the Ordering.Infrastructure project
//
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new ClientRequestEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new PaymentMethodEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    modelBuilder.ApplyConfiguration(new OrderItemEntityTypeConfiguration());
    //...Additional type configurations
}
```

No código a seguir, a infraestrutura de persistência é definida para a entidade Ordem:

```
// Part of the OrderEntityTypeConfiguration.cs class
//
public void Configure(EntityTypeBuilder<Order> orderConfiguration)
{
    orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
    orderConfiguration.HasKey(o => o.Id);
    orderConfiguration.Ignore(b => b.DomainEvents);
    orderConfiguration.Property(o => o.Id)
        .ForSqlServerUseSequenceHiLo("orderseq", OrderingContext.DEFAULT_SCHEMA);

    //Address value object persisted as owned entity in EF Core 2.0
    orderConfiguration.OwnsOne(o => o.Address);

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

    //...Additional validations, constraints and code...
    //...
}
```

No código anterior, o método `orderConfiguration.OwnsOne(o => o.Address)` especifica que a propriedade `Address` é uma entidade própria do tipo `Order`.

Por padrão, as colunas de nome e banco de dados de convenções do EF Core para as propriedades do tipo de entidade própria como `EntityProperty_OwnedEntityProperty`. Portanto, as propriedades internas do aparecerão `Address` na `Orders` tabela com os nomes `Address_Street` `Address_City` (e assim por diante para `State`, `Country` e `ZipCode`).

Você pode acrescentar o método fluente `Property().HasColumnName()` para renomear as colunas. No caso em que `Address` é uma propriedade pública, os mapeamentos seriam semelhantes ao seguinte:

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

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

É possível encadear o `OwnsOne` método em um mapeamento fluente. No exemplo hipotético a seguir, `OrderDetails` tem `BillingAddress` e `ShippingAddress`, que são tipos `Address`. Então `orderDetails` pertence ao tipo `Order`.

```

orderConfiguration.OwnsOne(p => p.OrderDetails, cb =>
{
    cb.OwnsOne(c => c.BillingAddress);
    cb.OwnsOne(c => c.ShippingAddress);
});
//...
//...
public class Order
{
    public int Id { get; set; }
    public OrderDetails OrderDetails { get; set; }
}

public class OrderDetails
{
    public Address BillingAddress { get; set; }
    public Address ShippingAddress { get; set; }
}

public class Address
{
    public string Street { get; set; }
    public string City { get; set; }
}

```

### Detalhes adicionais sobre tipos de entidade própria

- Os tipos próprios são definidos quando você configura uma propriedade de navegação com um tipo específico usando a API fluente OwnsOne.
- A definição de um tipo próprio em nosso modelo de metadados é uma composição de: o tipo de proprietário, a propriedade de navegação e o tipo CLR do tipo próprio.
- A identidade (chave) de uma instância de tipo próprio na nossa pilha é uma composição da identidade do tipo de proprietário e a definição do tipo próprio.

### Recursos de entidades de propriedade

- Os tipos próprios podem referenciar outras entidades, tanto próprias (tipos próprios aninhados) quanto não próprias (propriedades de navegação de referência comuns para outras entidades).
- Você pode mapear o mesmo tipo CLR como diferentes tipos próprios na mesma entidade de proprietário por meio de propriedades de navegação separadas.
- A divisão de tabela é configurada por convenção, mas você pode recusar mapeando o tipo de propriedade para uma tabela diferente usando ToTable.
- O carregamento adiantado é executado automaticamente em tipos de propriedade, ou seja, não há necessidade de chamar `.Include()` na consulta.
- Pode ser configurado com o atributo `[Owned]`, usando EF Core 2,1 e posterior.
- Pode lidar com coleções de tipos de propriedade (usando a versão 2,2 e posteriores).

### Limitações de entidades pertencentes

- Você não pode criar um `DbSet<T>` de um tipo de propriedade (por Design).
- Você não pode chamar `ModelBuilder.Entity<T>()` em tipos de propriedade (atualmente por Design).
- Não há suporte para tipos de propriedade opcionais (ou seja, anuláveis) que são mapeados com o proprietário na mesma tabela (ou seja, usando divisão de tabela). Isso ocorre porque o mapeamento é feito para cada propriedade, não há um sentinel separado para o valor complexo nulo como um todo.
- Não há suporte para mapeamento de herança para tipos de propriedade, mas você deve ser capaz de

mapear dois tipos folha das mesmas hierarquias de herança como tipos de propriedade diferentes. O EF Core não argumentará sobre o fato de que fazem parte da mesma hierarquia.

#### Principais diferenças com tipos complexos do EF6

- A divisão de tabela é opcional, ou seja, pode, opcionalmente, ser mapeada para uma tabela separada e ainda ser de tipos de propriedade.
- Eles podem referenciar outras entidades (ou seja, eles podem atuar como o lado dependente em relações com outros tipos de propriedade diferente).

## Recursos adicionais

- Martin Fowler. **Padrão valorobject**  
<https://martinfowler.com/bliki/ValueObject.html>
- Eric Evans. **Design controlado por domínio: solução de complexidade no coração do software.** (Livro; inclui uma discussão sobre objetos de valor)  
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>
- Vaughn Vernon. **Implementando o design controlado por domínio.** (Livro; inclui uma discussão sobre objetos de valor)  
<https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/>
- Tipos de entidade pertencentes  
</ef/core/modeling/owned-entities>
- Propriedades da sombra  
</ef/core/modeling/shadow-properties>
- Tipos complexos e/ou objetos de valor. Discussão no repositório GitHub do EF Core (guia Problemas)  
<https://github.com/dotnet/efcore/issues/246>
- **ValueObject.cs.** Classe de objeto de valor base no eShopOnContainers.  
<https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs>
- **Classe de endereços.** Exemplo de classe de objeto de valor em eShopOnContainers.  
<https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Address.cs>

[ANTERIOR](#)

[AVANÇAR](#)

# Usar classes de enumeração em vez de tipos enumerados

18/03/2020 • 3 minutes to read • [Edit Online](#)

[Enumerations](#) (ou *tipos enum*) são um wrapper de idioma fino em torno de um tipo integral. Talvez convenha limitar seu uso para quando você estiver armazenando um valor de um conjunto fechado de valores. A classificação com base em tamanhos (pequeno, médio ou grande) é um bom exemplo. Usar enumerações para fluxo de controle ou abstrações mais robustas pode ser um [code smell](#). Esse tipo de uso leva a um código frágil com muitas instruções de fluxo de controle que verificam os valores da enumeração.

Em vez disso, é possível criar classes Enumeration que habilitam todos os recursos avançados de uma linguagem orientada a objeto.

No entanto, isso não é um tópico crítico e, em muitos casos, para simplificar, ainda será possível usar [tipos enum](#) regulares se você preferir. De qualquer forma, o uso de classes de enumeração está mais estreitamente relacionado a conceitos relacionados a negócios.

## Implementar uma classe base de enumeração

O microsserviço de ordenação em eShopOnContainers fornece uma implementação de exemplo de classe base Enumeration, conforme mostrado no exemplo a seguir:

```

public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }

    public override string ToString() => Name;

    public static IEnumerable<T> GetAll<T>() where T : Enumeration
    {
        var fields = typeof(T).GetFields(BindingFlags.Public |
                                         BindingFlags.Static |
                                         BindingFlags.DeclaredOnly);

        return fields.Select(f => f.GetValue(null)).Cast<T>();
    }

    public override bool Equals(object obj)
    {
        var otherValue = obj as Enumeration;

        if (otherValue == null)
            return false;

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

        return typeMatches && valueMatches;
    }

    public int CompareTo(object other) => Id.CompareTo(((Enumeration)other).Id);

    // Other utility methods ...
}

```

Use essa classe como um tipo em qualquer entidade ou objeto de valor, como para a seguinte classe `CardType` :

```

Enumeration :

```

```

public class CardType : Enumeration
{
    public static readonly CardType Amex = new CardType(1, "Amex");
    public static readonly CardType Visa = new CardType(2, "Visa");
    public static readonly CardType MasterCard = new CardType(3, "MasterCard");

    public CardType(int id, string name)
        : base(id, name)
    {
    }
}

```

## Recursos adicionais

- **Jimmy Bogard. Aulas de enumeração**  
<https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/>
- **Steve Smith. Alternativas enum em C #**

<https://ardalis.com/enum-alternatives-in-c>

- **Enumeration.cs.** Classe base de enumeração em eShopOnContainers  
<https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/Enumeration.cs>
- **CardType.cs.** Exemplo de classe de enumeração em eShopOnContainers.  
<https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/AggregatesModel/BuyerAggregate/CardType.cs>
- **SmartEnum.** Ardalís – Classes para ajudar a produzir enumerações fortemente tipadas mais inteligentes no .NET.  
<https://www.nuget.org/packages/Ardalis.SmartEnum/>

[PRÓXIMO](#)

[ANTERIOR](#)

# Projetar validações na camada de modelo de domínio

10/09/2020 • 11 minutes to read • [Edit Online](#)

Em DDD, as regras de validação podem ser consideradas invariáveis. A principal responsabilidade de uma agregação é impor invariáveis entre as alterações de estado para todas as entidades dentro daquela agregação.

Entidades de domínio devem ser sempre entidades válidas. Existe um determinado número de invariáveis para um objeto que devem ser sempre verdadeiras. Por exemplo, um objeto de item do pedido sempre deve ter uma quantidade que deve ser um inteiro positivo, além de um nome de artigo e preço. Portanto, a imposição de invariáveis é de responsabilidade das entidades de domínio (especialmente da raiz de agregação) e um objeto de entidade não deve ser capaz de existir sem ser válido. Regras invariáveis simplesmente são expressas como contratos e exceções ou notificações são geradas quando elas são violadas.

O raciocínio por trás disso é que vários bugs ocorrerem porque os objetos estão em um estado em que nunca deveriam ter ficado. Esta [discussão online](#) é uma boa explicação de Greg Young.

Vamos propor que agora temos um SendUserCreationEmailService que usa um UserProfile..., como podemos racionalizar nesse serviço que o Nome não é nulo? Podemos verificar novamente? Ou, mais provável... você apenas não se preocupa em verificar e "espera pelo melhor": você espera que alguém tenha se preocupado em validá-lo antes de enviá-lo a você. É claro que, usando TDD, um dos primeiros testes que devemos escrever é que, se eu enviar um cliente com um nome nulo, isso deverá gerar um erro. Mas depois de começarmos a escrever esses tipos de testes repetidamente, percebemos... "Espere se nunca permitisse que o nome se torne nulo, não teríamos todos esses testes".

## Implementar validações na camada de modelo de domínio

As validações normalmente são implementadas em construtores de entidade de domínio ou em métodos que podem atualizar a entidade. Existem várias maneiras de implementar validações, como verificar dados e aumentar exceções se a validação falhar. Também há padrões mais avançados, como usar o padrão de Especificação para validações e o padrão de Notificação para retornar uma coleção de erros, em vez de retornar uma exceção para cada validação conforme ela ocorre.

### Validar condições e gerar exceções

O exemplo de código a seguir mostra a abordagem mais simples de validação em uma entidade de domínio gerando uma exceção. Na tabela de referências no final desta seção, você encontrará links para implementações mais avançadas com base nos padrões que discutimos anteriormente.

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
```

Um exemplo melhor seria demonstrar a necessidade de garantir que o estado interno não tenha mudado ou que todas as mutações para um método ocorreram. Por exemplo, a implementação a seguir deixará o objeto em um estado inválido:

```
public void SetAddress(string line1, string line2,
    string city, string state, int zip)
{
    _shippingAddress.line1 = line1 ?? throw new ...
    _shippingAddress.line2 = line2;
    _shippingAddress.city = city ?? throw new ...
    _shippingAddress.state = (IsValid(state) ? state : throw new ...);
}
```

Se o valor do estado for inválido, a primeira linha de endereço e a cidade já terão sido alteradas. Isso pode tornar o endereço inválido.

Uma abordagem semelhante pode ser usada no construtor da entidade, gerando uma exceção para garantir que a entidade seja válida depois de ser criada.

### Usar atributos de validação no modelo com base em anotações de dados

Anotações de dados, como os atributos Required ou MaxLength necessários, pode ser usado para configurar propriedades de campo de banco de dados do EF Core, conforme explicado em detalhes na seção [Mapeamento de tabela](#), mas [elas não funcionam mais para validação de entidade no EF Core](#) (o método `IValidatableObject.Validate` também não funciona mais para isso) como ocorria desde o EF 4.x no .NET Framework.

As anotações de dados e a `IValidatableObject` interface ainda podem ser usadas para validação de modelo durante a associação de modelo, antes da invocação de ações do controlador como de costume, mas esse modelo é destinado a ser um ViewModel ou dto e isso é um problema de API ou do MVC que não é uma preocupação com o modelo de domínio.

Tendo esclarecido a diferença conceitual, você ainda poderá usar anotações de dados e `IValidatableObject` na classe de entidade para a validação se as ações receberem um parâmetro de objeto de classe de entidade, o que não é recomendado. Nesse caso, a validação ocorrerá na associação de modelo, logo antes de invocar a ação, e você poderá verificar a Propriedade ModelState.IsValid do controlador para verificar o resultado, mas, novamente, isso ocorrerá no controlador, não antes de persistir o objeto de entidade no DbContext, como foi feito desde o EF 4.x.

Você ainda pode implementar a validação personalizada na classe de entidade usando as anotações de dados e o `IValidatableObject.Validate` método, substituindo o método `SaveChanges` do `DbContext`.

Você pode ver um exemplo de implementação para validar entidades `IValidatableObject` [neste comentário no GitHub](#). Esse exemplo não faz validações baseadas em atributo, mas elas devem ser fáceis de implementar usando reflexão na mesma substituição.

No entanto, de um ponto de vista do DDD, o modelo de domínio é mais bem mantido com o uso de exceções nos métodos de comportamento de sua entidade ou pela implementação dos padrões de especificação e de notificação para impor regras de validação.

Pode fazer sentido usar anotações de dados na camada de aplicativo em classes ViewModel (em vez de entidades de domínio) que aceitem a entrada para permitir a validação do modelo na camada da interface do usuário. No entanto, isso não deve ser feito na exclusão de validação dentro do modelo de domínio.

### Validar entidades implementando o padrão de Especificação e o padrão de Notificação

Por fim, uma abordagem mais elaborada para implementar a validação no modelo de domínio é implementando o padrão de Especificação em conjunto com o padrão de Notificação, conforme explicado em alguns dos recursos adicionais listados posteriormente.

Vale a pena mencionar que você também pode usar apenas um desses padrões — por exemplo, validação manual com instruções de controle, mas usando o padrão de Notificação para empilhar e retornar uma lista de erros de validação.

## Usar a validação adiada no domínio

Existem várias abordagens para lidar com validações adiadas no domínio. Em seu livro [Implementing Domain-Driven Design](#) (Implementando design controlado por domínio), Vaughn Vernon discute isso na seção sobre a validação.

## Validação de duas etapas

Considere também a validação de duas etapas. Use a validação em nível de campo em seu comando de DTOs (Objetos de Transferência de Dados) e a validação em nível de domínio dentro de suas entidades. Você pode fazer isso retornando um objeto de resultado, em vez de exceções para tornar mais fácil lidar com os erros de validação.

Usando a validação de campo com anotações de dados, por exemplo, você não duplica a definição de validação. A execução, no entanto, pode estar do lado do servidor e do lado do cliente no caso de DTOs (comandos e ViewModels, por exemplo).

## Recursos adicionais

- **Rachel Appel.** [Introdução à validação de modelo no ASP.NET Core MVC](#)  
[/aspnet/core/mvc/models/validation](#)
- **Rick Anderson.** [Adicionando validação](#)  
[/aspnet/core/tutorials/first-mvc-app/validation](#)
- **Martin Fowler.** [Substituindo exceções de lançamento com notificação em validações](#)  
<https://martinfowler.com/articles/replaceThrowWithNotification.html>
- **Padrões de especificação e de notificação**  
<https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns>
- **Nível Gorodinski.** [Validação no design controlado por domínio \(DDD\)](#)  
<http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/>
- **Tomada Colin.** [Validação do modelo de domínio](#)  
<https://colinjack.blogspot.com/2008/03/domain-model-validation.html>
- **Jimmy Bogard.** [Validação em um mundo DDD](#)  
<https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/>

[ANTERIOR](#)

[AVANÇAR](#)

# Validação do lado do cliente (validação nas camadas de apresentação)

09/04/2020 • 5 minutes to read • [Edit Online](#)

Mesmo quando a fonte de verdade for o modelo de domínio e, em último caso, você precisar ter validação no nível de modelo de domínio, a validação ainda poderá ser manipulada tanto no nível de modelo de domínio (lado do servidor) quanto da interface do usuário (lado do cliente).

A validação do lado do cliente é uma ótima conveniência para usuários. Ela economiza tempo que de outra forma seria gasto aguardando uma viagem de ida e volta que talvez retorne erros de validação. Em termos de negócios, até mesmo algumas frações de segundos multiplicadas por centenas de vezes por dia chega a ser muito tempo, despesa e frustração. A validação imediata e simples permite que os usuários trabalhem com mais eficiência e façam contribuições e produzam entradas e saídas de melhor qualidade.

Como o modelo de exibição e o modelo de domínio são diferentes, a validação do modelo de exibição e do modelo de domínio podem ser semelhantes, mas têm um propósito diferente. Se você está preocupado com DRY (o princípio Não Repita a si mesmo), considere que neste caso a reutilização do código também pode significar acoplamento, e em aplicativos corporativos é mais importante não acoplar o lado do servidor ao lado do cliente do que seguir o princípio DRY.

Mesmo ao usar a validação do lado do cliente, você sempre deve validar seus comandos ou DTOs de entrada no código do servidor, porque as APIs do servidor são um possível vetor de ataque. Geralmente, fazer as duas é a melhor opção, porque se você tiver um aplicativo cliente, de uma perspectiva do UX, será melhor ser proativo e não permitir que o usuário insira informações inválidas.

Portanto, no código do lado do cliente você normalmente valida os ViewModels. Você também pode validar os DTOs ou comandos de saída do cliente antes de enviá-los aos serviços.

A implementação de validação do lado do cliente depende de qual tipo de aplicativo cliente você está criando. Será diferente se você estiver validando dados em um aplicativo Web MVC da Web com a maior parte do código em .NET, um aplicativo Web SPA com a validação sendo codificada em JavaScript ou TypeScript ou um aplicativo móvel codificado com Xamarin e C#.

## Recursos adicionais

### Validação de aplicativos móveis Xamarin

- Validar a entrada de texto e mostrar erros

[https://developer.xamarin.com/recipes/ios/standard\\_controls/text\\_field/validate\\_input/](https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/)

- Chamada de validação

<https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/>

### Validação em aplicativos ASP.NET Core

- Rick Anderson. Adicionando validação

[/aspnet/core/tutorials/first-mvc-app/validation](https://aspnet/core/tutorials/first-mvc-app/validation)

### Validação em aplicativos Web SPA (Angular 2, TypeScript, JavaScript)

- Ado Kukic. Validação de formulário angular 2

<https://scotch.io/tutorials/angular-2-form-validation>

- Validação de formulários

<https://angular.io/guide/form-validation>

- **Validação.** Documentação da Breeze.

<https://breeze.github.io/doc-js/validation.html>

Em resumo, estes são os conceitos mais importantes no que diz respeito à validação:

- Entidades e agregados devem impor sua própria consistência e ser "sempre válidos". Raízes agregadas são responsáveis pela consistência de várias entidades dentro da mesma agregação.
- Se você acha que uma entidade precisa entrar em um estado inválido, considere usar um modelo de objeto diferente – por exemplo, usar um DTO temporário até criar a entidade de domínio definitiva.
- Se precisar criar vários objetos relacionados, como uma agregação, e eles apenas forem válidos depois que todos tiverem sido criados, considere usar o padrão de fábrica.
- Na maioria dos casos, ter validação redundante no lado do cliente é bom, porque o aplicativo pode ser proativo.

[PRÓXIMO](#)

[ANTERIOR](#)

# Eventos de domínio: design e implementação

10/09/2020 • 41 minutes to read • [Edit Online](#)

Use eventos de domínio para implementar explicitamente os efeitos colaterais de alterações em seu domínio. Em outras palavras, e usando terminologia DDD, use eventos de domínio para implementar explicitamente efeitos colaterais entre várias agregações. Opcionalmente, para melhor escalabilidade e menor impacto em bloqueios de banco de dados, use consistência eventual entre agregações dentro do mesmo domínio.

## O que é um evento de domínio?

Um evento é algo que ocorreu no passado. Um evento de domínio é algo que ocorreu no domínio que você deseja que outras partes do mesmo domínio (em processo) tenham conhecimento. As partes notificadas geralmente reagem de alguma forma aos eventos.

Um benefício importante dos eventos de domínio é que os efeitos colaterais podem ser expressos explicitamente.

Por exemplo, se você estivesse usando apenas o Entity Framework e precisasse haver uma reação a um evento, provavelmente você codificaria tudo o que precisa perto do que dispara o evento. Portanto, a regra fica acoplada, implicitamente, ao código e você precisa examinar o código para, com sorte, perceber que a regra está implementada lá.

Por outro lado, usar eventos de domínio torna o conceito explícito, porque há um `DomainEvent` e pelo menos um `DomainEventHandler` envolvidos.

Por exemplo, no aplicativo eShopOnContainers, quando um pedido é criado, o usuário se torna um comprador, portanto, um `OrderStartedDomainEvent` será disparado e tratado no `ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler`, de forma que o conceito subjacente é evidente.

Em resumo, eventos de domínio ajudam você a expressar, explicitamente, as regras de domínio, com base na linguagem ubíqua fornecida pelos especialistas do domínio. Os eventos de domínio também permitem uma melhor separação de interesses entre classes dentro do mesmo domínio.

É importante garantir que, assim como uma transação de banco de dados, todas as operações relacionadas a um evento de domínio sejam concluídas com êxito ou nenhuma delas seja.

Os eventos de domínio são parecidos com eventos do estilo de mensagens, com uma diferença importante. Com mensagens reais, enfileiramento de mensagens, agentes de mensagens ou um barramento de serviço que usa o AMQP, uma mensagem é sempre enviada de forma assíncrona e comunicada entre processos e computadores. Isso é útil para a integração de vários contextos delimitados, microsserviços ou até mesmo aplicativos diferentes. No entanto, com os eventos de domínio, ao acionar um evento na operação de domínio em execução no momento, você deseja que os efeitos colaterais ocorram dentro do mesmo domínio.

Os eventos de domínio e seus efeitos colaterais (as ações disparadas depois que são gerenciadas por manipuladores de eventos) devem ocorrer quase imediatamente, geralmente em processo, e dentro do mesmo domínio. Assim, os eventos de domínio podem ser síncronos ou assíncronos. Os eventos de integração, no entanto, devem sempre ser assíncronos.

## Eventos de domínio versus eventos de integração

Semanticamente, os eventos de integração e de domínio são a mesma coisa: notificações sobre algo que acabou de ocorrer. No entanto, a implementação deles deve ser diferente. Os eventos de domínio são apenas mensagens enviadas por push para um dispatcher de evento de domínio, que pode ser implementado como um mediador na

memória, com base em um contêiner de IoC ou qualquer outro método.

Por outro lado, a finalidade dos eventos de integração é a propagação de transações e atualizações confirmadas para outros subsistemas, independentemente de serem outros microsserviços, contextos delimitados ou, até mesmo, aplicativos externos. Assim, eles deverão ocorrer somente se a entidade for persistida com êxito, caso contrário, será como se toda a operação nunca tivesse acontecido.

Conforme o que foi mencionado antes, os eventos de integração devem ser baseados em comunicação assíncrona entre vários microsserviços (outros contextos delimitados) ou mesmo aplicativos/sistemas externos.

Assim, a interface do barramento de eventos precisa de alguma infraestrutura que permita a comunicação entre processos e distribuída entre serviços potencialmente remotos. Ela pode ser baseada em um barramento de serviço comercial, em filas, em um banco de dados compartilhado usado como uma caixa de correio ou em qualquer outro sistema de mensagens distribuídas e, idealmente, baseado em push.

## Eventos de domínio como uma maneira preferencial para disparar efeitos colaterais entre várias agregações dentro do mesmo domínio

Se a execução de um comando relacionado a uma instância de agregação exigir regras de domínio adicionais para ser executado em uma ou mais agregações, você deverá projetar e implementar esses efeitos colaterais para que sejam disparados por eventos de domínio. Conforme mostrado na Figura 7-14 e como um dos mais importantes casos de uso, um evento de domínio deve ser usado para propagar alterações de estado entre várias agregações dentro do mesmo modelo de domínio.

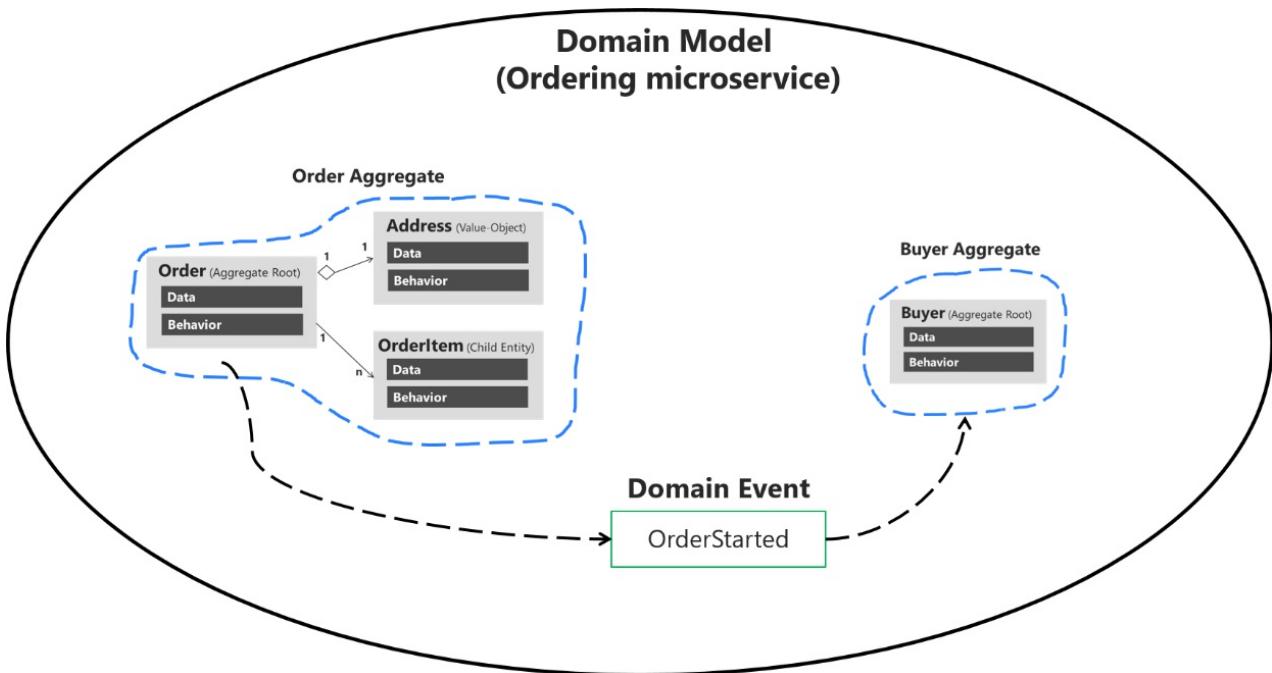


Figura 7-14. Eventos de domínio para impor consistência entre várias agregações dentro do mesmo domínio

A Figura 7-14 mostra como a consistência entre agregações é obtida por eventos de domínio. Quando o usuário inicia uma ordem, a agregação de ordem envia um `OrderStarted` evento de domínio. O evento de domínio `OrderStarted` é tratado pelo comprador agregado para criar um objeto de comprador no microserviço de ordenação, com base nas informações do usuário original do microserviço de identidade (com as informações fornecidas no comando `CreateOrder`).

Como alternativa, você pode fazer com que a raiz da agregação assine eventos acionado pelos membros de suas respectivas agregações (entidades filho). Por exemplo, cada entidade filho `OrderItem` poderá acionar um evento quando o preço do item for maior que um valor específico, ou quando a quantidade de itens do produto for muito alta. Assim, a raiz de agregação poderá receber esses eventos e executar um cálculo global ou uma agregação.

É importante entender que essa comunicação baseada em eventos não é implementada diretamente nas agregações; você precisa implementar manipuladores de eventos de domínio.

A manipulação de eventos de domínio é um interesse do aplicativo. A camada do modelo de domínio deve se concentrar apenas na lógica do domínio, algo que um especialista em domínio entende, e não na infraestrutura do aplicativo, como manipuladores e ações de persistência de efeito colateral com o uso de repositórios. Portanto, o nível de camada de aplicativo é o local em que você deve ter manipuladores de eventos de domínio disparando ações quando um evento de domínio é acionado.

Os eventos de domínio também podem ser usados para disparar um grande número de ações de aplicativo e, o mais importante, devem estar abertos para aumentar esse número no futuro de maneira separada. Por exemplo, quando o pedido é iniciado, você publica um evento de domínio para propagar essas informações para outras agregações ou, até mesmo, para gerar ações de aplicativo, como notificações.

O ponto-chave é o número indefinido de ações a serem executadas quando ocorre um evento de domínio. As ações e regras do domínio e do aplicativo vão, eventualmente, aumentar. A complexidade ou o número de ações de efeito colateral quando algo acontecer aumentará, mas se o seu código estivesse acoplado a "cola" (ou seja, criando objetos específicos com `new`), sempre que for necessário adicionar uma nova ação, você também precisaria alterar o código de trabalho e testado.

Essa alteração pode resultar em novos bugs e essa abordagem também vai contra o [Princípio Aberto/Fechado de SOLID](#). E não se trata apenas disso, pois a classe original que estava orquestrando as operações cresceria sem parar, o que vai contra o [SRP \(princípio de responsabilidade única\)](#).

Por outro lado, se você usa eventos de domínio, você pode criar uma implementação refinada e desacoplada por meio da segregação de responsabilidades, usando essa abordagem:

1. Enviar um comando (por exemplo, `CreateOrder`).
2. Receber o comando em um manipulador de comandos.
  - Executar uma única transação de agregação.
  - (Opcional) Acionar eventos de domínio para efeitos colaterais (por exemplo, `OrderStartedDomainEvent`).
3. Manipular eventos de domínio (dentro do processo atual) que executarão um número indefinido de efeitos colaterais em várias agregações ou ações de aplicativo. Por exemplo:
  - Verificar ou criar o comprador e a forma de pagamento.
  - Criar e enviar um evento de integração relacionado ao barramento de eventos a fim de propagar estados entre microsserviços ou disparar ações externas, como o envio de um email para o comprador.
  - Manipular outros efeitos colaterais.

Conforme mostrado na Figura 7-15, começando pelo mesmo evento de domínio, você pode manipular várias ações relacionadas a outras agregações do domínio ou ações de aplicativos adicionais que você precisa realizar entre microsserviços que se conectam com eventos de integração e o barramento de eventos.

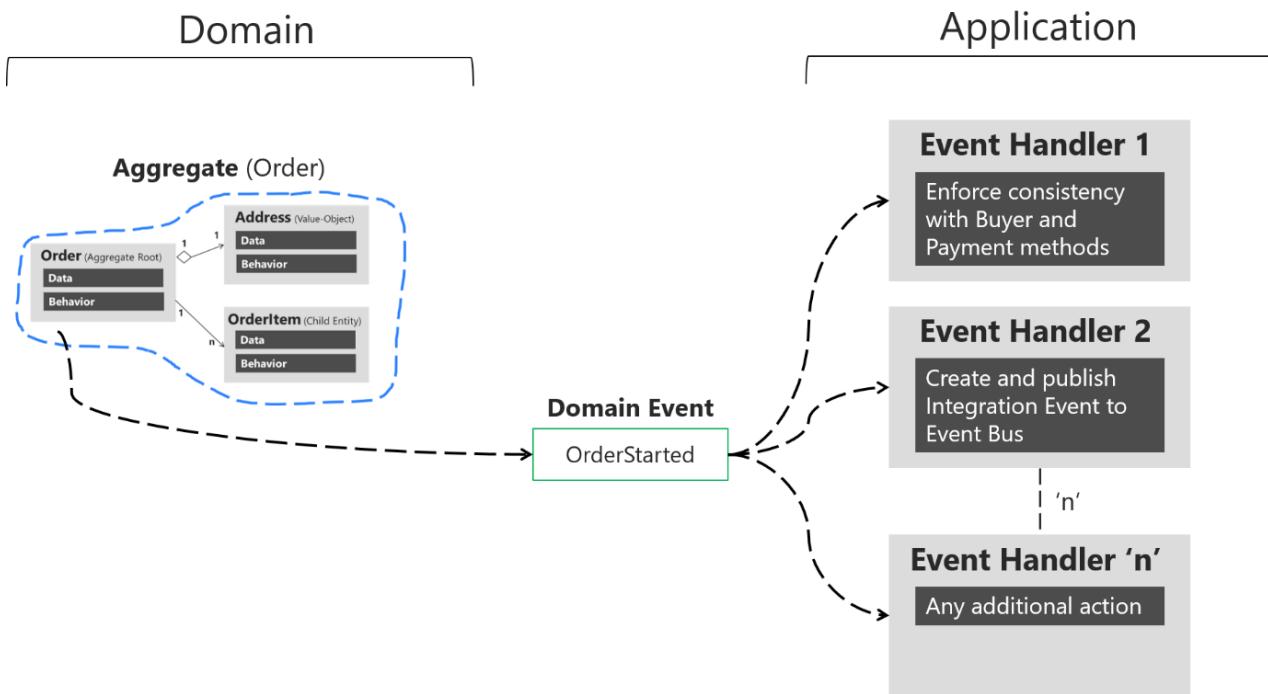


Figura 7-15. Manipulando várias ações por domínio

Pode haver vários manipuladores para o mesmo evento de domínio na camada de aplicativo, um manipulador pode resolver a consistência entre agregações e outro manipulador pode publicar um evento de integração, para que outros microsserviços possam fazer algo com ele. Os manipuladores de eventos normalmente estão na camada de aplicativo, pois você usará objetos de infraestrutura como repositórios ou uma API de aplicativo para o comportamento do microserviço. Nesse sentido, os manipuladores de eventos são semelhantes aos manipuladores de comandos, portanto, ambos fazem parte da camada de aplicativo. A diferença importante é que um comando deve ser processado apenas uma vez. Um evento de domínio pode ser processado zero ou *n* vezes, porque ele pode ser recebido por vários destinatários ou manipuladores de eventos, com uma finalidade diferente para cada manipulador.

Ter um número aberto de manipuladores por evento de domínio permite que você adicione quantas regras de domínio forem necessárias, sem afetar o código atual. Por exemplo, a implementação da seguinte regra de negócios poderá ser tão fácil quanto adicionar alguns manipuladores de eventos (ou apenas um):

Quando o valor total comprado por um cliente na loja, em qualquer número de pedidos, excede US\$ 6.000, aplicar 10% de desconto para cada novo pedido e notificar o cliente com um email, informando sobre esse desconto para pedidos futuros.

## Implementar eventos de domínio

No C#, um evento de domínio é simplesmente uma classe ou estrutura de retenção de dados, como um DTO, com todas as informações relacionadas ao que acabou de ocorrer no domínio, conforme mostrado no exemplo a seguir:

```

public class OrderStartedDomainEvent : INotification
{
    public string UserId { get; }
    public int CardTypeId { get; }
    public string CardNumber { get; }
    public string CardSecurityNumber { get; }
    public string CardHolderName { get; }
    public DateTime CardExpiration { get; }
    public Order Order { get; }

    public OrderStartedDomainEvent(Order order,
                                   int cardTypeId, string cardNumber,
                                   string cardSecurityNumber, string cardHolderName,
                                   DateTime cardExpiration)
    {
        Order = order;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}

```

Essa é essencialmente uma classe que retém todos os dados relacionados ao evento OrderStarted.

Nos termos da linguagem ubíqua do domínio, como um evento é algo que ocorreu no passado, o nome de classe do evento deverá ser representado como um verbo no passado, como OrderStartedDomainEvent ou OrderShippedDomainEvent. É assim que o evento de domínio é implementado no microsserviço de pedidos no eShopOnContainers.

Conforme observado anteriormente, uma característica importante de eventos é que, como um evento é algo que ocorreu no passado, ele não deve ser alterado. Portanto, ele deve ser uma classe imutável. Observe no código anterior que as propriedades são somente leitura. Não é possível atualizar o objeto, você pode definir os valores apenas quando ele é criado.

É importante destacar aqui que, se os eventos de domínio fossem tratados de forma assíncrona, usando uma fila que exigia a serialização e desserialização dos objetos de evento, as propriedades teriam que ser "conjunto particular" em vez de somente leitura, de modo que o desserializador seria capaz de atribuir os valores na retirada do enfileiramento. Isso não é um problema no microsserviço de pedidos, pois o evento de domínio pub/sub é implementado de forma síncrona usando o MediatR.

### Acionar eventos de domínio

A próxima pergunta é: como acionar um evento de domínio para que ele alcance os respectivos manipuladores de eventos? Você pode usar várias abordagens.

Udi Dahan originalmente propôs (em várias postagens relacionadas, como, [Domain Events – Take 2 \(Eventos de domínio – tomada 2\)](#)) o uso de uma classe estática para gerenciar e acionar eventos. Isso incluiria uma classe estática chamada DomainEvents, que geraria eventos de domínio assim que fosse chamada, usando uma sintaxe como: `DomainEvents.Raise(Event myEvent)`. Jimmy Bogard escreveu uma postagem no blog ([Strengthening your domain: Domain Events \(Fortalecendo seu domínio: eventos de domínio\)](#)) que recomenda uma abordagem semelhante.

No entanto, quando a classe dos eventos de domínio é estática, ela também faz a expedição imediata aos manipuladores. Isso torna o teste e a depuração mais difíceis, pois os manipuladores de eventos com a lógica de efeitos colaterais são executados imediatamente após o evento ser acionado. Ao testar e depurar, você quer se concentrar somente no que está acontecendo nas classes de agregação atuais; você não deseja ser redirecionado repentinamente para outros manipuladores de eventos de efeitos colaterais relacionados a outras agregações ou lógica de aplicativo. É por isso as outras abordagens evoluíram, conforme explicado na próxima seção.

### A abordagem adiada para acionar e despacho de eventos

Em vez de expedir imediatamente para um manipulador de eventos de domínio, uma abordagem melhor é adicionar os eventos de domínio a uma coleção e, em seguida, expedir esses eventos de domínio *logo antes* ou *logo depois* da confirmação da transação (como acontece com SaveChanges no EF). (Essa abordagem foi descrita por Jimmy Bogard nesta postagem [A better domain events pattern \(Um padrão de eventos de domínio melhor\)](#)).

A decisão de enviar os eventos de domínio logo antes ou logo após a confirmação da transação é importante, pois determinará se você vai incluir os efeitos colaterais como parte da mesma transação ou em transações diferentes. No último caso, você precisará lidar com a consistência eventual entre várias agregações. Este tópico será abordado na próxima seção.

A abordagem adiada é que o eShopOnContainers usa. Primeiro, você adiciona os eventos que estão ocorrendo nas suas entidades a uma coleção ou uma lista de eventos por entidade. Essa lista deve fazer parte do objeto de entidade, ou, melhor ainda, parte de sua classe de entidade base, conforme mostrado no seguinte exemplo da classe base Entity:

```
public abstract class Entity
{
    //...
    private List<INotification> _domainEvents;
    public List<INotification> DomainEvents => _domainEvents;

    public void AddDomainEvent(INotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<INotification>();
        _domainEvents.Add(eventItem);
    }

    public void RemoveDomainEvent(INotification eventItem)
    {
        _domainEvents?.Remove(eventItem);
    }
    //... Additional code
}
```

Quando você deseja acionar um evento, basta adicioná-lo à coleção de eventos através do código em qualquer método de entidade raiz de agregação.

O seguinte código, parte da [Raiz de agregação de ordem no eShopOnContainers](#), mostra um exemplo:

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

this.AddDomainEvent(orderStartedDomainEvent);
```

Observe que a única coisa que o método AddDomainEvent está fazendo é a adição de um evento à lista. Nenhum evento foi expedido, e nenhum manipulador de eventos foi invocado ainda.

Na verdade, você deseja expedir os eventos mais tarde, ao confirmar a transação no banco de dados. Se você estiver usando o Entity Framework Core, isso será realizado no método SaveChanges do DbContext do EF, como no código a seguir:

```

// EF Core DbContext
public class OrderingContext : DbContext, IUnitOfWork
{
    // ...
    public async Task<bool> SaveEntitiesAsync(CancellationToken cancellationToken =
default(CancellationToken))
    {
        // Dispatch Domain Events collection.
        // Choices:
        // A) Right BEFORE committing data (EF SaveChanges) into the DB. This makes
        // a single transaction including side effects from the domain event
        // handlers that are using the same DbContext with Scope lifetime
        // B) Right AFTER committing data (EF SaveChanges) into the DB. This makes
        // multiple transactions. You will need to handle eventual consistency and
        // compensatory actions in case of failures.
        await _mediator.DispatchDomainEventsAsync(this);

        // After this line runs, all the changes (from the Command Handler and Domain
        // event handlers) performed through the DbContext will be committed
        var result = await base.SaveChangesAsync();
    }
}

```

Com esse código, você expede os eventos de entidade aos respectivos manipuladores de eventos.

O resultado geral é que você desacoplou o acionamento de um evento de domínio (uma simples adição a uma lista na memória) da expedição dele para um manipulador de eventos. Além disso, dependendo do tipo de dispatcher que você está usando, é possível expedir os eventos de forma síncrona ou assíncrona.

Lembre-se que os limites transacionais desempenham funções significativas aqui. Se for possível sua unidade de trabalho e transação alcançar mais de uma agregação (como ao usar o EF Core e um banco de dados relacional), isso poderá funcionar bem. Mas se a transação não puder alcançar agregações, como ao usar um banco de dados NoSQL, como o Azure CosmosDB, você precisará implementar etapas adicionais para obter consistência. Essa é outra razão por que a ignorância de persistência não é universal; ela depende do sistema de armazenamento que é usado.

### **Transação única entre agregações versus consistência eventual entre agregações**

Executar uma única transação entre agregações em vez de depender de consistência eventual entre essas agregações é uma questão controversa. Muitos autores de DDD, como Eric Evans e Vaughn Vernon, defendem a regra de que uma transação = uma agregação e, portanto, defendem a consistência eventual entre agregações. Por exemplo, em seu livro *Domain-Driven Design*, Eric Evans diz:

Não é esperado que toda regra que abrange Agregações esteja atualizada em todos os momentos. Por meio de processamento de eventos, processamento em lote ou de outros mecanismos de atualização, outras dependências podem ser resolvidas dentro de um período específico. (página 128)

Vaughn Vernon diz o seguinte em [design agregado efetivo. Parte II: fazer com que as agregações funcionem juntas](#):

Portanto, se a execução de um comando em uma instância de agregação exigir que regras de negócios adicionais sejam executadas em uma ou mais agregações, use a consistência eventual [...] Há uma maneira prática de dar suporte à consistência eventual em um modelo DDD. Um método de agregação publica um evento de domínio que é entregue no momento exato a um ou mais assinantes assíncronos.

Essa lógica é baseada na adoção de transações refinadas em vez de transações que abrangem muitas agregações ou entidades. A ideia é que, no segundo caso, o número de bloqueios de banco de dados será significativo em aplicativos de larga escala com necessidades de alta escalabilidade. Aceitar o fato de que aplicativos altamente

escalonáveis não precisam de consistência transacional instantânea entre várias agregações ajuda a aceitar o conceito de consistência eventual. Geralmente, as mudanças atômicas não são necessárias aos negócios e, em todo caso, é da responsabilidade dos especialistas de domínio dizer se operações específicas precisam ou não de transações atômicas. Se uma operação sempre precisar de uma transação atômica entre várias agregações, você poderá questionar se a agregação deveria ser maior ou se não foi corretamente projetada.

No entanto, outros desenvolvedores e arquitetos, como Jimmy Bogard, estão de acordo com a abrangência de uma única transação entre várias agregações, mas somente quando essas agregações adicionais forem relacionadas a efeitos colaterais do mesmo comando original. Por exemplo, em [A better domain events pattern \(Um padrão melhor de eventos de domínio\)](#), Bogard diz:

Normalmente, quero que os efeitos colaterais de um evento de domínio ocorram dentro da mesma transação lógica, mas não necessariamente no mesmo escopo de gerar o evento de domínio [...] Logo antes de confirmarmos nossa transação, expedimos nossos eventos para seus respectivos manipuladores.

Se você expedir os eventos de domínio imediatamente *antes* da confirmação da transação original, será porque você deseja que os efeitos colaterais desses eventos sejam incluídos na mesma transação. Por exemplo, se o método SaveChanges do DbContext do EF falhar, a transação reverterá todas as alterações, incluindo o resultado de qualquer operação de efeito colateral implementada pelos manipuladores de eventos de domínio relacionados. Isso ocorre porque o escopo de vida do DbContext é, por padrão, definido como "com escopo". Portanto, o objeto DbContext é compartilhado entre vários objetos de repositório que estão sendo instanciados dentro do mesmo escopo ou objeto graph. Isso coincide com o escopo de HttpRequest ao desenvolver aplicativos da API Web ou do MVC.

Na realidade, as duas abordagens (transação atômica única e consistência eventual) podem ser corretas. Isso realmente dependerá dos seus requisitos de negócios ou do domínio e o que os especialistas de domínio lhe indicarem. Também dependerá da escalabilidade necessária para o serviço (transações mais granulares tem menos impacto em relação aos bloqueios de banco de dados). E dependerá de quanto você está disposto a investir em seu código, pois a consistência eventual exige um código mais complexo, para detectar possíveis inconsistências entre agregações, e a necessidade de implementar ações compensatórias. Considere que, se você confirmar as alterações à agregação original e posteriormente, quando os eventos estiverem sendo expedidos, houver um problema e os manipuladores de eventos não puderem confirmar os efeitos colaterais, você terá inconsistências entre agregações.

Uma forma de permitir ações compensatórias seria armazenar os eventos de domínio em tabelas de banco de dados adicionais para que eles possam fazer parte da transação original. Posteriormente, você poderia ter um processo em lote para detectar inconsistências e executar ações compensatórias por meio da comparação da lista de eventos com o estado atual das agregações. As ações compensatórias fazem parte de um tópico complexo que exigirá uma análise detalhada, incluindo discuti-las com o usuário empresarial e com os especialistas de domínio.

De qualquer maneira, você pode optar pela abordagem que seja necessária. Mas a abordagem adiada inicial — disparar os eventos antes da confirmação, de forma a usar uma única transação — é a abordagem mais simples ao usar o EF Core e um banco de dados relacional. Ela é mais fácil de implementar e é válida em muitos casos de negócio. Ela também é a abordagem usada no microsserviço de pedidos no eShopOnContainers.

Mas, de que maneira você realmente envia esses eventos aos respectivos manipuladores de eventos? O que é o objeto `_mediator` visto no exemplo anterior? Ele tem a ver com as técnicas e artefatos que você usa para mapear entre eventos e os respectivos manipuladores de eventos.

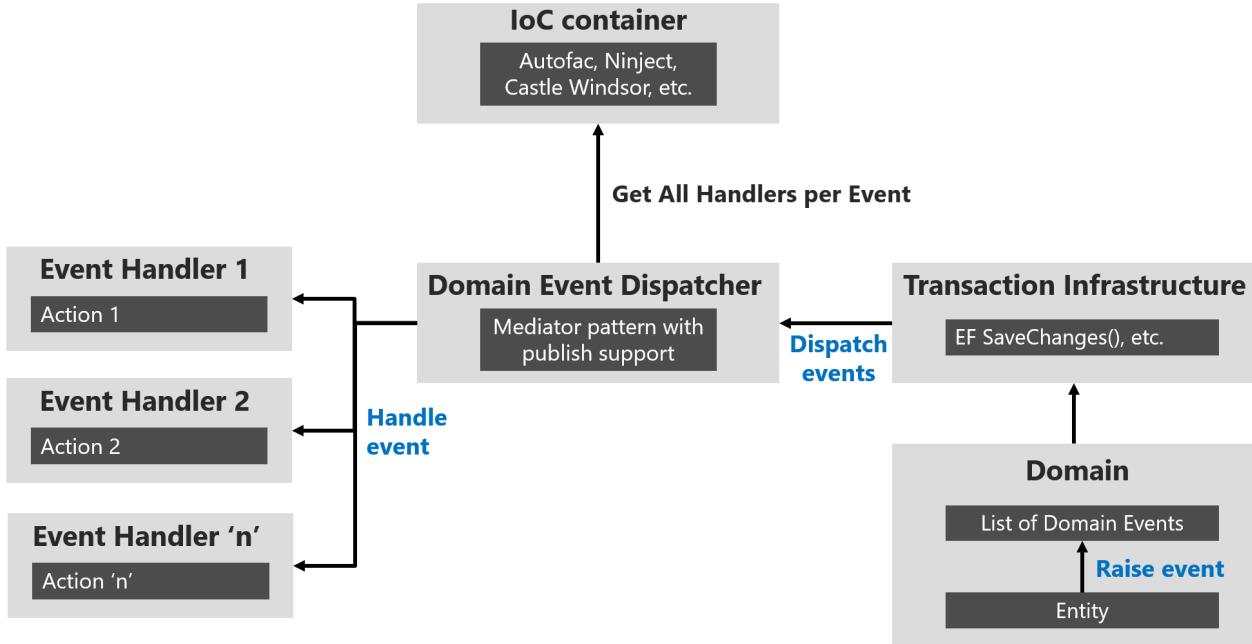
### O dispatcher de evento de domínio: mapeamento de eventos a manipuladores de eventos

Assim que estiver pronto para expedir ou publicar os eventos, você precisará de algum tipo de artefato que publicará o evento, para que cada manipulador relacionado possa obtê-lo e processar os efeitos colaterais com base nesse evento.

Uma abordagem é um sistema de mensagens real ou até mesmo um barramento de eventos, possivelmente

baseado em um barramento de serviço, em vez de eventos na memória. No entanto, no primeiro caso, os sistemas de mensagens real seria um exagero para processar eventos de domínio, pois você só precisa processar esses eventos dentro do mesmo processo (ou seja, no mesmo domínio e na mesma camada de aplicativo).

Outra maneira de mapear eventos para vários manipuladores de eventos é o uso de registro de tipos em um contêiner de IoC para que você possa inferir dinamicamente o local para expedir os eventos. Em outras palavras, você precisa saber quais manipuladores de eventos precisam obter um evento específico. A figura 7-16 mostra uma abordagem simplificada para esta abordagem.



**Figura 7-16.** Dispatcher de evento de domínio usando IoC

Você pode criar todos os detalhes técnicos e artefatos para implementar essa abordagem por si só. No entanto, você também pode usar as bibliotecas disponíveis, como a [MediatR](#), que usa seu contêiner de IoC nos bastidores. Portanto, você pode usar diretamente as interfaces predefinidas e os métodos de publicação/expedição do objeto mediador.

No código, primeiro você precisa registrar os tipos de manipulador de eventos no contêiner de IoC, conforme mostrado no seguinte exemplo no [microsserviço de pedidos eShopOnContainers](#):

```

public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        // Other registrations ...
        // Register the DomainEventHandler classes (they implement IAsyncNotificationHandler<>)
        // in assembly holding the Domain Events
        builder.RegisterAssemblyTypes(typeof(ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler)
            .GetTypeInfo().Assembly
            .AsClosedTypesOf(typeof(IAsyncNotificationHandler<>)));
        // Other registrations ...
    }
}
  
```

Primeiro, o código identifica o assembly que contém os manipuladores de eventos do domínio ao localizar o assembly que contém qualquer um dos manipuladores (usando `typeof(ValidateOrAddBuyerAggregateWhenXxxx)`), mas você poderia escolher qualquer outro manipulador de eventos para localizar o assembly). Como todos os manipuladores de eventos implementam a interface `IAsyncNotificationHandler`, então o código pesquisa apenas esses tipos e registra todos os manipuladores de eventos.

## Como assinar eventos de domínio

Quando você usa o MediatR, cada manipulador de eventos deve usar um tipo de evento que é fornecido no parâmetro genérico da interface `INotificationHandler`, como pode ser visto no código a seguir:

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : IAsyncNotificationHandler<OrderStartedDomainEvent>
```

Com base na relação entre o evento e o manipulador de eventos, que pode ser considerada a assinatura, o artefato do MediatR consegue descobrir todos os manipuladores de eventos de cada evento e disparar cada um desses manipuladores de eventos.

### Como manipular eventos de domínio

Por fim, o manipulador de eventos geralmente implementa o código da camada de aplicativo, que usa os repositórios de infraestrutura para obter as agregações adicionais necessárias e executar a lógica de domínio do efeito colateral. O seguinte [código de manipulador de eventos de domínio em eShopOnContainers](#), mostra um exemplo de implementação.

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : INotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILoggerFactory _logger;
    private readonly IBuyerRepository<Buyer> _buyerRepository;
    private readonly IIdentityService _identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
        ILoggerFactory logger,
        IBuyerRepository<Buyer> buyerRepository,
        IIdentityService identityService)
    {
        // ...Parameter validations...
    }

    public async Task Handle(OrderStartedDomainEvent orderStartedEvent)
    {
        var cardTypeId = (orderStartedEvent.CardTypeId != 0) ? orderStartedEvent.CardTypeId : 1;
        var userGuid = _identityService.GetUserIdentity();
        var buyer = await _buyerRepository.FindAsync(userGuid);
        bool buyerOriginallyExisted = (buyer == null) ? false : true;

        if (!buyerOriginallyExisted)
        {
            buyer = new Buyer(userGuid);
        }

        buyer.VerifyOrAddPaymentMethod(cardTypeId,
            $"Payment Method on {DateTime.UtcNow}",
            orderStartedEvent.CardNumber,
            orderStartedEvent.CardSecurityNumber,
            orderStartedEvent.CardHolderName,
            orderStartedEvent.CardExpiration,
            orderStartedEvent.Order.Id);

        var buyerUpdated = buyerOriginallyExisted ? _buyerRepository.Update(buyer)
            : _buyerRepository.Add(buyer);

        await _buyerRepository.UnitOfWork
            .SaveEntitiesAsync();

        // Logging code using buyerUpdated info, etc.
    }
}
```

O código de manipulador de eventos de domínio anterior é considerado um código da camada de aplicativo

porque ele usa repositórios de infraestrutura, conforme explicado na próxima seção sobre a camada de persistência de infraestrutura. Os manipuladores de eventos também podem usar outros componentes de infraestrutura.

#### **Eventos de domínio podem gerar eventos de integração para serem publicados fora dos limites do microsserviço**

Por fim, é importante mencionar que, às vezes, convém propagar eventos entre vários microsserviços. Essa propagação é considerada um evento de integração e ele pode ser publicado por meio de um barramento de eventos proveniente de qualquer manipulador de eventos de domínio específico.

## Conclusões sobre eventos de domínio

Conforme mencionado, use eventos de domínio para implementar explicitamente os efeitos colaterais de alterações em seu domínio. Para usar a terminologia DDD, use eventos de domínio para implementar explicitamente efeitos colaterais entre uma ou várias agregações. Além disso e para melhor escalabilidade e menor impacto em bloqueios de banco de dados, use consistência eventual entre agregações dentro do mesmo domínio.

O aplicativo de referência usa [mediador](#) para propagar eventos de domínio de forma síncrona entre agregações em uma única transação. No entanto, você também pode usar alguma implementação de AMQP como [RabbitMQ](#) ou [barramento de serviço do Azure](#) para propagar eventos de domínio de forma assíncrona, usando a consistência eventual, mas, como mencionado acima, você precisa considerar a necessidade de ações de compensação em caso de falhas.

## Recursos adicionais

- Greg Young. O que é um evento de domínio?  
<https://cqrssamples.com/documents.pdf#page=25>
- Stenberg de Jan. Eventos de domínio e consistência eventual  
<https://www.infoq.com/news/2015/09/domain-events-consistency>
- Jimmy Bogard. Um padrão de eventos de domínio melhor  
<https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/>
- Vaughn Vernon. Design de agregação efetivo parte II: fazer com que as agregações funcionem juntas  
[https://dddcommunity.org/wp-content/uploads/files/pdf\\_articles/Vernon\\_2011\\_2.pdf](https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf)
- Jimmy Bogard. Fortalecendo seu domínio: eventos de domínio  
<https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/>
- Tony Truong. Exemplo de padrão de eventos de domínio  
<https://www.tonytruong.net/domain-events-pattern-example/>
- Udi Dahan. Como criar modelos de domínio totalmente encapsulados  
<https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>
- Udi Dahan. Eventos de domínio – Take 2  
<https://udidahan.com/2008/08/25/domain-events-take-2/>
- Udi Dahan. Eventos de domínio – Salvation  
<https://udidahan.com/2009/06/14/domain-events-salvation/>
- Kronquist de Jan. Não publique eventos de domínio, retorne-os!  
<https://blogjayway.com/2013/06/20/dont-publish-domain-events-return-them/>
- Cesar de la Torre. Eventos de domínio vs. eventos de integração em arquiteturas DDD e de microservices

<https://devblogs.microsoft.com/cesardelatorre/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/>

ANTERIOR

AVANÇAR

# Projetar a camada de persistência da infraestrutura

10/09/2020 • 17 minutes to read • [Edit Online](#)

Os componentes de persistência de dados fornecem acesso aos dados hospedados nos limites de um microserviço (ou seja, um banco de dados de microserviço). Eles contêm a implementação real dos componentes, como repositórios e classes [Unidade de Trabalho](#), como objetos [DbContext](#) personalizados do EF (Entity Framework). O DbContext do EF implementa os padrões de Repositório e de Unidade de Trabalho.

## O padrão de repositório

Repositórios são classes ou componentes que encapsulam a lógica necessária para acessar fontes de dados. Eles centralizam a funcionalidade comum de acesso a dados, melhorando a sustentabilidade e desacoplando a infraestrutura ou a tecnologia usada para acessar os bancos de dados da camada do modelo de domínio. Se você usar um ORM (Mapeador Objeto-Relacional) como o Entity Framework, o código que precisa ser implementado será simplificado, graças à LINQ e à tipagem forte. Isso permite que você se concentre na lógica de persistência de dados e não nos detalhes técnicos do acesso a dados.

O padrão de repositório é uma maneira bem documentada de trabalhar com uma fonte de dados. No livro [Padrões de Arquitetura de Aplicações Corporativas](#), Martin Fowler descreve um repositório da seguinte maneira:

Um repositório executa as tarefas de um intermediário entre as camadas de modelo de domínio e o mapeamento de dados, funcionando de maneira semelhante a um conjunto de objetos de domínio na memória. Os objetos de clientes criam consultas de forma declarativa e enviam-nas para os repositórios buscando respostas. Conceitualmente, um repositório encapsula um conjunto de objetos armazenados no banco de dados e as operações que podem ser executadas neles, fornecendo uma maneira que é mais próxima da camada de persistência. Os repositórios também oferecem a capacidade de separação, de forma clara e em uma única direção, a dependência entre o domínio de trabalho e a alocação de dados ou o mapeamento.

### Definir um repositório por agregação

Para cada agregação ou raiz de agregação, você deve criar uma classe de repositório. Em um microsserviço baseado nos padrões de DDD (Design Orientado por Domínio), o único canal que você deve usar para atualizar o banco de dados são os repositórios. Isso ocorre porque eles têm uma relação um-para-um com a raiz agregada, que controla as invariáveis da agregação e a consistência transacional. É possível consultar o banco de dados por outros canais (como ao seguir uma abordagem de CQRS), porque as consultas não alteram o estado do banco de dados. No entanto, a área transacional (ou seja, as atualizações) sempre precisa ser controlada pelos repositórios e pelas raízes de agregação.

Basicamente, um repositório permite popular na memória dados que são provenientes do banco de dados, em forma de entidades de domínio. Depois que as entidades estão na memória, elas podem ser alteradas e persistidas novamente no banco de dados por meio de transações.

Conforme observado anteriormente, se você estiver usando o padrão de arquitetura CQS/CQRS, as consultas iniciais serão executadas por consultas à parte, fora do modelo de domínio, executadas por instruções SQL simples usando o Dapper. Essa abordagem é muito mais flexível do que os repositórios porque você pode consultar e unir as tabelas necessárias, e essas consultas não são restrinvidas pelas regras das agregações. Esses dados vão para o aplicativo cliente ou de camada de apresentação.

Se o usuário fizer alterações, os dados a serem atualizados virão da camada de apresentação ou do aplicativo cliente para a camada do aplicativo (como um serviço de API Web). Ao receber um comando em um manipulador

de comandos, use repositórios para obter os dados que deseja atualizar do banco de dados. Você os atualizará na memória com os dados transmitidos com os comandos e, em seguida, adicionará ou atualizará esses dados (entidades de domínio) no banco de dados por meio de uma transação.

É importante enfatizar novamente que você deve definir apenas um repositório para cada raiz de agregação, conforme mostrado na Figura 7-17. Para atingir a meta da raiz de agregação de manter a consistência transacional entre todos os objetos na agregação, você nunca deve criar um repositório para cada tabela no banco de dados.

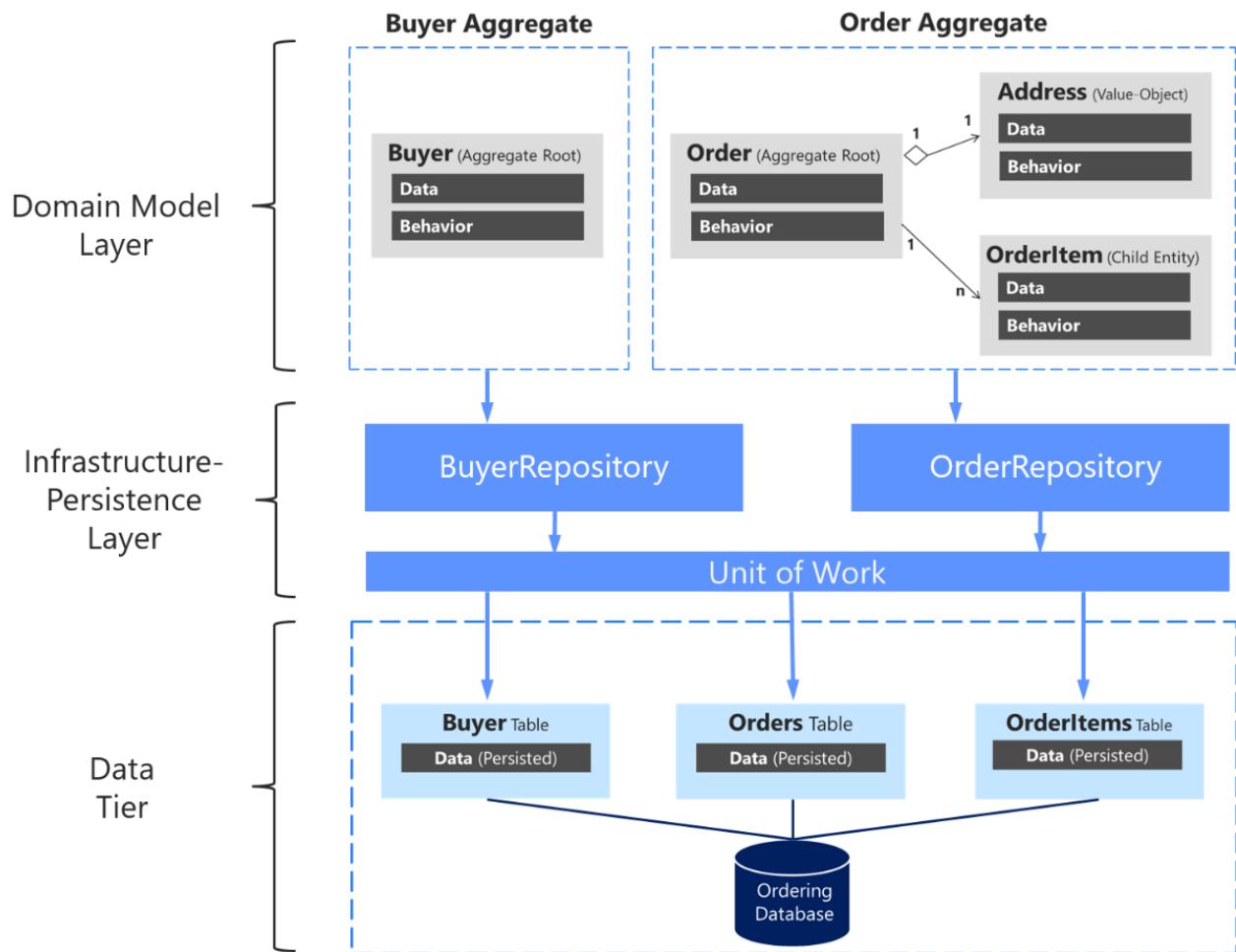


Figura 7-17. A relação entre repositórios, agregações e tabelas de banco de dados

O diagrama acima mostra as relações entre as camadas de infraestrutura e de domínio: a agregação de compradores depende do IBuyerRepository e a agregação de ordem depende das interfaces IOrderRepository, essas interfaces são implementadas na camada de infraestrutura pelos repositórios correspondentes que dependem de UnitOfWork, também implementadas ali, que acessam as tabelas na camada de dados.

#### Impor uma raiz de agregação por repositório

É importante implementar o design do repositório de uma forma que ele imponha a regra de que apenas as raízes de agregação devem ter repositórios. Você pode criar um tipo de repositório genérico ou de base que restrinja o tipo de entidades com as quais trabalha para garantir que elas tenham a interface de marcador `IAggregateRoot`.

Assim, cada classe de repositório implementada na camada de infraestrutura implementa seu próprio contrato ou interface, conforme é mostrado no código a seguir:

```
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
        // ...
    }
}
```

Cada interface de repositório específica implementa a interface IRepository genérica:

```
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);
    // ...
}
```

No entanto, uma maneira melhor de fazer com que o código imponha a convenção de que cada repositório seja relacionado a uma única agregação é implementar um tipo de repositório genérico. Dessa forma, é obrigatório que você esteja usando um repositório para direcionar a uma agregação específica. Isso pode ser feito facilmente com a implementação de uma interface base IRepository genérica, como no código a seguir:

```
public interface IRepository<T> where T : IAggregateRoot
{
    //....
}
```

### O padrão de repositório facilita os testes da lógica do aplicativo

O padrão de repositório permite testar facilmente o aplicativo com testes de unidade. Lembre-se de que os testes de unidade testam apenas o código, não a infraestrutura, assim, as abstrações de repositório facilitam alcançar essa meta.

Como observado em uma seção anterior, é recomendado que você defina e coloque as interfaces de repositório na camada de modelo de domínio para que a camada de aplicativo, como o microsserviço API Web, não dependa diretamente da camada de infraestrutura em que as classes de repositório reais foram implementadas. Fazendo isso e usando a injeção de dependência nos controladores da API Web, você pode implementar repositórios fictícios que retornam dados falsos em vez de dados do banco de dados. Essa abordagem desacoplada permite que você crie e execute testes de unidade que se focam na lógica do aplicativo sem precisar de conectividade com o banco de dados.

As conexões com bancos de dados podem falhar e, principalmente, executar centenas de testes em relação a um banco de dados é prejudicial por dois motivos. Primeiro, pode demorar muito devido ao grande número de testes. Segundo, os registros do banco de dados podem ser alterados e afetar os resultados dos testes, deixando-os inconsistentes. Testar em relação ao banco de dados não é um teste de unidade, mas sim um teste de integração. É interessante ter muitos testes de unidade em execução rápida, mas menos testes de integração em relação aos bancos de dados.

Em termos de separação de interesses para os testes de unidade, a lógica opera em entidades de domínio na memória. Ela considera que a classe de repositório as entregou. Depois que a lógica modifica as entidades de domínio, ela considera que a classe de repositório as armazenará corretamente. O ponto importante aqui é criar testes de unidade em relação ao seu modelo de domínio e à sua lógica de domínio. As raízes de agregação são os limites de consistência principais em DDD.

Os repositórios implementados no eShopOnContainers contam com a implementação DbContext de EF Core do repositório e os padrões de unidade de trabalho usando seu rastreador de alterações, portanto, eles não duplicam essa funcionalidade.

## A diferença entre o padrão de repositório e o padrão da classe DAL (classe de acesso a dados) herdada

Um objeto de acesso a dados executa operações de acesso e persistência de dados diretamente no armazenamento. Um repositório marca os dados com as operações que você deseja executar na memória de um objeto de unidade de trabalho (como no EF ao usar a classe [DbContext](#)), mas essas atualizações não são realizadas imediatamente no banco de dados.

Uma unidade de trabalho é conhecida como uma única transação que envolve várias operações de inserção, atualização ou exclusão. Simplificando, isso significa que, para uma ação de usuário específica, como o registro em um site, as operações de inserção, atualização e exclusão são tratadas em uma única transação. Isso é mais eficiente do que tratar várias transações de banco de dados de uma maneira mais intensa.

Essas várias operações de persistência serão executadas mais tarde em uma única ação quando o código da camada de aplicativo executar um comando para isso. A decisão de como aplicar as alterações realizadas na memória ao armazenamento de banco de dados geralmente se baseia no [padrão de unidade de trabalho](#). No EF, o padrão de Unidade de Trabalho é implementado como o [DbContext](#).

Em muitos casos, esse padrão ou uma maneira de aplicar operações em relação ao armazenamento pode aumentar o desempenho do aplicativo e reduzir a possibilidade de inconsistências. Além disso, ele reduz o bloqueio de transações nas tabelas do banco de dados, porque todas as operações pretendidas são confirmadas em uma única transação. Isso é mais eficiente em comparação com a execução de muitas operações isoladas no banco de dados. Portanto, o ORM selecionado é capaz de otimizar a execução no banco de dados agrupando várias ações de atualização na mesma transação, em vez de executar várias transações pequenas e separadas.

## Os repositórios não são obrigatórios

Os repositórios personalizados são úteis pelos motivos já citados e essa é a abordagem para o microsserviço de pedidos no eShopOnContainers. No entanto, esse não é um padrão essencial a ser implementado em um design DDD ou até mesmo no desenvolvimento para .NET, em geral.

Por exemplo, Jimmy Bogard, ao fornecer comentários diretos para este guia, diz o seguinte:

Isso provavelmente será meu maior comentário. Eu realmente não é fã dos repositórios, principalmente porque eles ocultam os detalhes importantes do mecanismo de persistência subjacente. É por isso que eu também sou usado para obter os comandos do mediador. Posso usar toda a capacidade da camada de persistência e enviar por push todo esse comportamento de domínio para minhas raízes de agregação. Eu geralmente não quero simular meus repositórios – ainda preciso fazer esse teste de integração com o verdadeiro. Usar o CQRS significava que não realmente precisamos de mais repositórios.

Os repositórios podem ser úteis, mas eles não são críticos para o design DDD como são o padrão de Agregação e o modelo de domínio avançado. Portanto, use o padrão de repositório ou não, conforme achar mais adequado. De qualquer forma, você usará o padrão de repositório sempre que usar EF Core embora, nesse caso, o repositório cubra todo o Microservice ou o contexto limitado.

## Recursos adicionais

### Padrão de Re却itório

- Edward Hieatt e Rob me. Padrão de repositório.  
<https://martinfowler.com/eaaCatalog/repository.html>
- O padrão de repositório  
[/previous-versions/msp-n-p/ff649690\(v=pandp.10\)](/previous-versions/msp-n-p/ff649690(v=pandp.10))
- Eric Evans. Design controlado por domínio: solução de complexidade no coração do software.  
(Livro; inclui uma discussão sobre o padrão de Re却itório)  
<https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/>

## **Padrão de unidade de trabalho**

- Martin Fowler. Padrão de unidade de trabalho.  
<https://martinfowler.com/eaaCatalog/unitOfWork.html>
- Implementando o repositório e os padrões de unidade de trabalho em um aplicativo MVC ASP.NET  
</aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>

[ANTERIOR](#)

[AVANÇAR](#)

# Implementar a camada de persistência de infraestrutura com o Entity Framework Core

10/09/2020 • 30 minutes to read • [Edit Online](#)

Ao usar bancos de dados relacionais, como o SQL Server, o Oracle ou o PostgreSQL, uma abordagem recomendada é implementar a camada de persistência com base no EF (Entity Framework). O EF é compatível com LINQ e fornece objetos fortemente tipados para o modelo, bem como uma persistência simplificada no banco de dados.

O Entity Framework tem uma longa história de participação no .NET Framework. Ao usar o .NET Core, você também deve usar o Entity Framework Core, que é executado no Windows ou no Linux da mesma maneira que o .NET Core. EF Core é uma reescrita completa de Entity Framework implementada com uma superfície muito menor e melhorias importantes no desempenho.

## Introdução ao Entity Framework Core

O Entity Framework (EF) Core é uma versão de multiplataforma leve, extensível e de plataforma cruzada da popular tecnologia de acesso a dados do Entity Framework. Ele foi introduzido com o .NET Core em meados de 2016.

Como uma introdução ao EF Core já está disponível na documentação da Microsoft, aqui nós vamos fornecer apenas os links para as informações.

### Recursos adicionais

- **Entity Framework Core**  
<https://docs.microsoft.com/ef/core/>
- **Introdução ao ASP.NET Core e Entity Framework Core usando o Visual Studio**  
<https://docs.microsoft.com/aspnet/core/data/ef-mvc/>
- **Classe DbContext**  
<https://docs.microsoft.com/dotnet/api/microsoft.entityframeworkcore.dbcontext>
- **Comparar EF Core & EF6.x**  
<https://docs.microsoft.com/ef/efcore-and-ef6/index>

## Infraestrutura no Entity Framework Core da perspectiva do DDD

Do ponto de vista do DDD, um recurso importante do EF é a capacidade de usar as entidades de domínio POCO (objeto CRL básico), também conhecidas na terminologia do EF como *entidades code-first* POCO. Se você usar as entidades de domínio POCO, as classes de modelo de domínio ignorarão a persistência, seguindo os princípios de [Ignorância de Persistência](#) e de [Ignorância de Infraestrutura](#).

De acordo com os padrões do DDD você deve encapsular o comportamento e as regras do domínio dentro da própria classe de entidade, assim ela poderá controlar as invariáveis, as validações e as regras ao acessar qualquer coleção. Portanto, não é uma prática recomendada no DDD permitir o acesso público a coleções de entidades filhas ou a objetos de valor. Em vez disso, é possível expor métodos que controlam como e quando as coleções de propriedade e os campos podem ser atualizados e qual comportamento e medidas deverão ser tomadas quando isso acontecer.

Desde o EF Core 1.1, para atender a esses requisitos de DDD, é possível ter campos simples nas entidades em vez de propriedades públicas. Se você não quiser que um campo de entidade fique acessível externamente, bastará

criar o atributo ou o campo em vez de uma propriedade. Também é possível usar setters de propriedade privada.

Da mesma forma, agora é possível ter acesso somente leitura a coleções usando uma propriedade pública digitada como `IReadOnlyCollection<T>`, com o apoio de um membro de campo privado para a coleção (como uma `List<T>`) na entidade, que se baseia no EF para persistência. As versões anteriores do Entity Framework exigiam que as propriedades da coleção fossem compatíveis com `ICollection<T>`, o que significava que qualquer desenvolvedor que usasse uma classe da entidade pai poderia adicionar ou remover itens por meio de suas coleções de propriedade. Essa possibilidade seria em relação aos padrões recomendados no DDD.

É possível usar uma coleção privada ao expor um objeto `IReadOnlyCollection<T>` somente leitura, como é mostrado no exemplo de código a seguir:

```
public class Order : Entity
{
    // Using private fields, allowed since EF Core 1.1
    private DateTime _orderDate;
    // Other fields ...

    private readonly List<OrderItem> _orderItems;
    public IReadOnlyCollection<OrderItem> OrderItems => _orderItems;

    protected Order() { }

    public Order(int buyerId, int paymentMethodId, Address address)
    {
        // Initializations ...
    }

    public void AddOrderItem(int productId, string productName,
                           decimal unitPrice, decimal discount,
                           string pictureUrl, int units = 1)
    {
        // Validation logic...

        var orderItem = new OrderItem(productId, productName,
                                      unitPrice, discount,
                                      pictureUrl, units);
        _orderItems.Add(orderItem);
    }
}
```

A `OrderItems` propriedade só pode ser acessada como somente leitura usando `IReadOnlyCollection<OrderItem>`. Esse tipo é somente leitura, portanto, ele está protegido contra as atualizações regulares.

O EF Core fornece uma maneira de mapear o modelo de domínio para o banco de dados físico sem "contaminar" o modelo de domínio. Trata-se de puro código POCO do .NET, pois a ação de mapeamento é implementada na camada de persistência. Nessa ação de mapeamento, você precisa configurar o mapeamento dos campos para o banco de dados. No exemplo a seguir o método `OnModelCreating` de `OrderingContext` e da classe `OrderEntityTypeConfiguration`, a chamada para `SetPropertyAccessMode` informa ao EF Core para acessar a propriedade `OrderItems` por meio de seu campo.

```

// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // ...
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    // Other entities' configuration ...
}

// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
        // Other configuration

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

        // EF access the OrderItem collection property through its backing field
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);

        // Other configuration
    }
}

```

Quando você usa campos em vez de propriedades, a `OrderItem` entidade é persistida como se tivesse uma `List<OrderItem>` propriedade. No entanto, ela expõe um único acessador, o método `AddOrderItem`, para adicionar novos itens ao pedido. Como resultado, o comportamento e os dados ficarão vinculados e serão consistentes em todos os códigos de aplicativo que usarem o modelo de domínio.

## Implementar repositórios personalizados com o Entity Framework Core

No nível da implementação, um repositório é simplesmente uma classe com o código de persistência de dados, coordenada por uma unidade de trabalho (DbContext no EF Core) ao executar atualizações, como mostra a seguinte classe:

```

// using directives...
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository
    {
        private readonly OrderingContext _context;
        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }

        public BuyerRepository(OrderingContext context)
        {
            _context = context ?? throw new ArgumentNullException(nameof(context));
        }

        public Buyer Add(Buyer buyer)
        {
            return _context.Buyers.Add(buyer).Entity;
        }

        public async Task<Buyer> FindAsync(string buyerIdentityGuid)
        {
            var buyer = await _context.Buyers
                .Include(b => b.Payments)
                .Where(b => b.FullName == buyerIdentityGuid)
                .SingleOrDefaultAsync();

            return buyer;
        }
    }
}

```

A `IBuyerRepository` interface vem da camada de modelo de domínio como um contrato. No entanto, a implementação do repositório é feita na camada de persistência e de infraestrutura.

O `DbContext` do EF é fornecido pelo construtor por meio de injeção de dependência. Ele é compartilhado entre vários repositórios no mesmo escopo de solicitação HTTP, graças ao seu tempo de vida padrão (`ServiceLifetime.Scoped`) no contêiner de IoC (inversão de controle) (que também pode ser definido explicitamente com `services.AddDbContext<>`).

### **Métodos a serem implementados em um repositório (atualizações ou transações em comparação com consultas)**

Em cada classe de repositório, você deve colocar os métodos de persistência que atualizam o estado das entidades contidas na agregação relacionada. Lembre-se de que há uma relação um-para-um entre uma agregação e seu repositório relacionado. Leve em consideração que um objeto de entidade de raiz de agregação pode ter entidades filhas inseridas no grafo do EF. Por exemplo, um comprador pode ter vários métodos de pagamento como entidades filhas relacionadas.

Como a abordagem para o microsserviço de pedidos no eShopOnContainers também se baseia em CQS/CQRS, a maioria das consultas não são implementadas em repositórios personalizados. Os desenvolvedores têm a liberdade de criar as consultas e junções que precisam para a camada de apresentação sem as restrições impostas pelas agregações, pelos repositórios personalizados por agregação e pelo DDD em geral. A maioria dos repositórios personalizados sugeridos por este guia tem vários métodos de atualização ou transacionais, mas apenas os métodos de consulta necessários para fazer com que os dados sejam atualizados. Por exemplo, o repositório `BuyerRepository` implementa um método `FindAsync`, porque o aplicativo precisa saber se um comprador específico existe antes de criar um novo comprador relacionado ao pedido.

No entanto, os métodos de consulta reais para obter os dados a serem enviados à camada de apresentação ou aos aplicativos clientes são implementados, conforme mencionado, nas consultas de CQRS baseadas em consultas flexíveis usando Dapper.

### Usando um repositório personalizado em vez de usar o DbContext EF diretamente

A classe DbContext Entity Framework é baseada na unidade de padrões de trabalho e de repositório e pode ser usada diretamente do seu código, como de um controlador MVC ASP.NET Core. A unidade de padrões de trabalho e de repositório resulta no código mais simples, como no microserviço de catálogo CRUD em eShopOnContainers. Nos casos em que você deseja o código mais simples possível, é possível usar diretamente a classe DbContext, como muitos desenvolvedores fazem.

No entanto, a implementação de repositórios personalizados oferece vários benefícios ao implementar microsserviços ou aplicativos mais complexos. A unidade de padrões de trabalho e de repositório destina-se a encapsular a camada de persistência de infraestrutura para que ela seja dissociada das camadas do aplicativo e do modelo de domínio. A implementação desses padrões pode facilitar o uso de repositórios fictícios para simulação de acesso ao banco de dados.

Na Figura 7-18, você pode ver as diferenças entre não usar repositórios (diretamente usando o EF DbContext) versus usar repositórios, o que torna mais fácil simular esses repositórios.

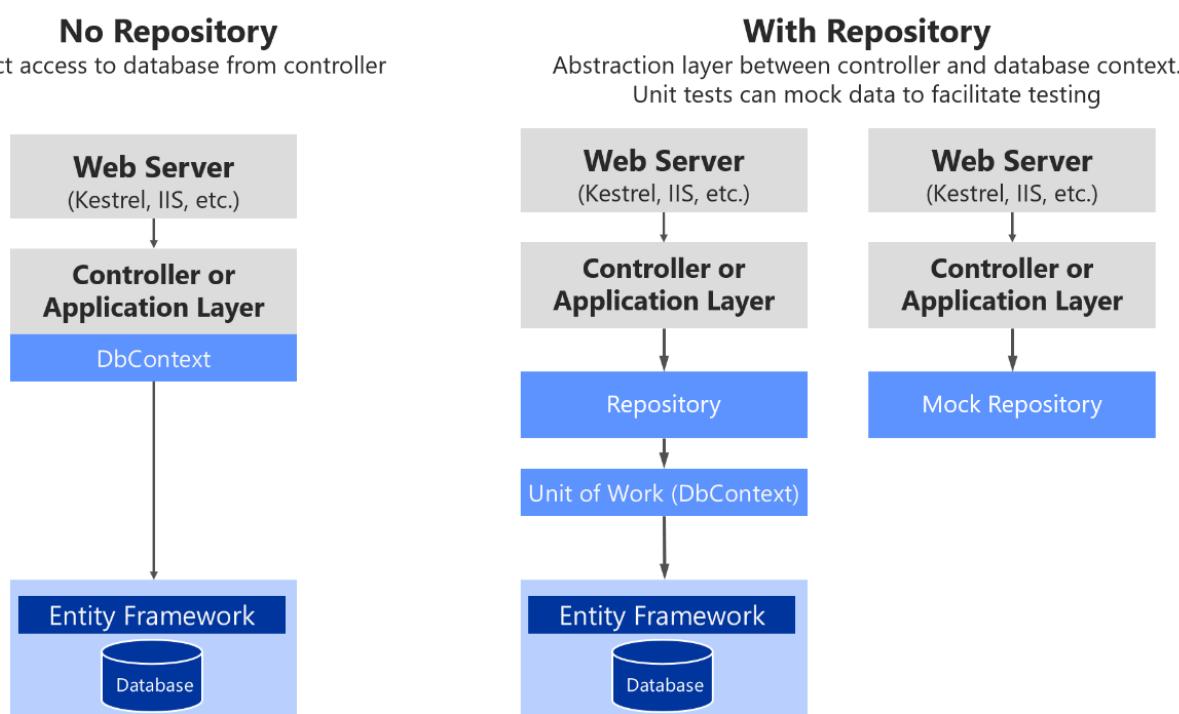


Figura 7-18. Usando repositórios personalizados em vez de um DbContext simples

A Figura 7-18 mostra que o uso de um repositório personalizado adiciona uma camada de abstração que pode ser usada para facilitar o teste, simulando o repositório. Existem várias alternativas para simulação. Você pode simular apenas repositórios ou simular toda a unidade de trabalho. Geralmente, simular apenas os repositórios já é suficiente e a complexidade de abstrair e simular toda a unidade de trabalho, normalmente, não é necessária.

Mais adiante, quando nos concentramos na camada de aplicativo, você verá como funciona a injeção de dependência no ASP.NET Core e como ela é implementada ao usar repositórios.

Em resumo, os repositórios personalizados permitem que você teste o código mais facilmente com testes de unidade que não são afetados pelo estado da camada de dados. Se você executar testes que também acessem o banco de dados real por meio do Entity Framework, eles não serão testes de unidade, mas sim testes de integração, que são muito mais lentos.

Se estivesse usando o DbContext diretamente, você precisaria simulá-lo ou executar testes de unidade usando um SQL Server na memória, com os dados previsíveis para testes de unidade. Mas simular o DbContext ou controlar

dados falsos requer mais trabalho do que a simulação no nível do repositório. Obviamente, sempre é possível testar os controladores MVC.

## Tempo de vida da instância de DbContext e de IUnitOfWork do EF no contêiner de IoC

O objeto `DbContext` (exposto como um objeto `IunitOfWork`) deve ser compartilhado entre vários repositórios dentro do mesmo escopo de solicitação HTTP. Por exemplo, isso é verdadeiro quando a operação que está sendo executada precisa lidar com várias agregações ou simplesmente porque você está usando várias instâncias do repositório. Também é importante mencionar que a interface `IunitOfWork` faz parte da camada de domínio, ela não é um tipo do EF Core.

Para isso, o tempo de vida de serviço da instância do objeto `DbContext` precisa ser definido como `ServiceLifetime.Scoped`. Este é o tempo de vida padrão ao registrar um `DbContext` com `services.AddDbContext` no contêiner de IoC (inversão de controle) do método `ConfigureServices` do arquivo `Startup.cs`, no projeto de API Web ASP.NET Core. O código a seguir ilustra isso.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc(options =>
    {
        options.Filters.Add(typeof(HttpGlobalExceptionFilter));
    }).AddControllersAsServices();

    services.AddEntityFrameworkSqlServer()
        .AddDbContext<OrderingContext>(options =>
    {
        options.UseSqlServer(Configuration["ConnectionString"],
            sqlOptions => sqlOptions.MigrationsAssembly(typeof(Startup).GetTypeInfo().
                Assembly.GetName().Name));
    },
        ServiceLifetime.Scoped // Note that Scoped is the default choice
            // in AddDbContext. It is shown here only for
            // pedagogic purposes.
    );
}
```

O modo de criação de instância do `DbContext` não deve ser configurado como `ServiceLifetime.Transient` ou `ServiceLifetime.Singleton`.

## O tempo de vida da instância de repositório no contêiner de IoC

De forma semelhante, o tempo de vida do repositório normalmente deve ser definido como com escopo (`InstancePerLifetimeScope` em Autofac). Ele também pode ser transitório (`InstancePerDependency` no Autofac), mas o serviço será mais eficiente em relação à memória ao usar o tempo de vida no escopo.

```
// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
    .As<IOderRepository>()
    .InstancePerLifetimeScope();
```

Usar o tempo de vida singleton para o repositório pode causar sérios problemas de simultaneidade quando o `DbContext` é definido como tempo de vida do `InstancePerLifetimeScope` (escopo) (os tempos de vida padrão de um `DbContext`).

### Recursos adicionais

- **Implementando o repositório e os padrões de unidade de trabalho em um aplicativo MVC ASP.NET**  
<https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application>
- **Jonathan Allen. Estratégias de implementação para o padrão de repositório com Entity Framework, Dapper e Chain**  
<https://www.infoq.com/articles/repository-implementation-strategies>
- **Cesar de la Torre. Comparando ASP.NET Core tempos de vida do serviço de contêiner IoC com escopos de instância de contêiner Autofac IoC**  
<https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/>

## Mapeamento de tabela

O mapeamento de tabela identifica os dados de tabela a serem consultados e salvos no banco de dados. Você já viu como as entidades de domínio (por exemplo, um domínio de produto ou de pedido) podem ser usadas para gerar um esquema de banco de dados relacionado. O EF foi projetado rigidamente de acordo com o conceito de *convenções*. As convenções abordam perguntas como "qual será o nome de uma tabela?" ou "qual propriedade é a chave primária?" As convenções normalmente são baseadas em nomes convencionais. Por exemplo, é comum que a chave primária seja uma propriedade que termina com `Id`.

Por convenção, cada entidade será configurada para ser mapeada para uma tabela com o mesmo nome que a propriedade `DbSet< TEntity >` que expõe a entidade no contexto derivado. Se nenhum valor de `DbSet< TEntity >` for fornecido para a entidade especificada, o nome da classe será usado.

### Anotações de dados em comparação com a API fluente

Existem muitas convenções do EF Core adicionais e a maioria delas pode ser alterada usando anotações de dados ou a API fluente, implementada no método `OnModelCreating`.

As anotações de dados devem ser usadas nas próprias classes de modelo de entidade, o que é uma maneira mais invasiva do ponto de vista de DDD. Isso ocorre porque você está contaminando o modelo com anotações de dados relacionadas ao banco de dados de infraestrutura. Por outro lado, a API fluente é uma maneira conveniente de alterar a maioria das convenções e dos mapeamentos dentro da camada de infraestrutura de persistência de dados, para que o modelo de entidade fique limpo e desacoplado da infraestrutura de persistência.

### API fluente e o método `OnModelCreating`

Conforme mencionado, para alterar as convenções e os mapeamentos, você pode usar o método `OnModelCreating` na classe `DbContext`.

O microsserviço de pedidos no `eShopOnContainers` implementa o mapeamento e configuração explícitos, quando necessário, conforme é mostrado no código a seguir.

```
// At OrderingContext.cs from eShopOnContainers
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // ...
    modelBuilder.ApplyConfiguration(new OrderEntityTypeConfiguration());
    // Other entities' configuration ...
}

// At OrderEntityTypeConfiguration.cs from eShopOnContainers
class OrderEntityTypeConfiguration : IEntityTypeConfiguration<Order>
{
    public void Configure(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", OrderingContext.DEFAULT_SCHEMA);
```

```

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

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

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

//Address value object persisted as owned entity type supported since EF Core 2.0
orderConfiguration
    .OwnsOne(o => o.Address, a =>
{
    a.WithOwner();
});

orderConfiguration
    .Property<int?>("_buyerId")
    .UsePropertyAccessMode(PropertyAccessMode.Field)
    .HasColumnName("BuyerId")
    .IsRequired(false);

orderConfiguration
    .Property<DateTime>("_orderDate")
    .UsePropertyAccessMode(PropertyAccessMode.Field)
    .HasColumnName("OrderDate")
    .IsRequired();

orderConfiguration
    .Property<int>("_orderStatusId")
    .UsePropertyAccessMode(PropertyAccessMode.Field)
    .HasColumnName("OrderStatusId")
    .IsRequired();

orderConfiguration
    .Property<int?>("_paymentMethodId")
    .UsePropertyAccessMode(PropertyAccessMode.Field)
    .HasColumnName("PaymentMethodId")
    .IsRequired(false);

orderConfiguration.Property<string>("Description").IsRequired(false);

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

// DDD Patterns comment:
//Set as field (New since EF 1.1) to access the OrderItem collection property through its field
navigation SetPropertyAccessMode(PropertyAccessMode.Field);

orderConfiguration.HasOne<PaymentMethod>()
    .WithMany()
    .HasForeignKey("_paymentMethodId")
    .IsRequired(false)
    .OnDelete(DeleteBehavior.Restrict);

orderConfiguration.HasOne<Buyer>()
    .WithMany()
    .IsRequired(false)
    .HasForeignKey("_buyerId");

orderConfiguration.HasOne(o => o.OrderStatus)
    .WithMany()
    .HasForeignKey("_orderStatusId");
}

}

```

Você pode definir todos os mapeamentos de API fluente dentro do mesmo `OnModelCreating` método, mas é aconselhável partitionar esse código e ter várias classes de configuração, uma por entidade, conforme mostrado no exemplo. Especialmente para modelos grandes, é aconselhável ter classes de configuração separadas para

configurar tipos de entidade diferentes.

O código no exemplo mostra algumas declarações e mapeamentos explícitos. No entanto, as convenções do EF Core fazem muitos desses mapeamentos automaticamente, portanto, o código real necessário para o seu caso poderá ser menor.

### O algoritmo Hi/Lo no EF Core

Um aspecto interessante de código no exemplo anterior é que ele usa o [algoritmo Hi/Lo](#) como a estratégia de geração de chave.

O algoritmo Hi/Lo é útil quando você precisa de chaves exclusivas antes de confirmar as alterações. Em resumo, o algoritmo Hi-Lo atribui identificadores exclusivos às linhas da tabela, embora ele não dependa de armazenar a linha no banco de dados imediatamente. Isso permite começar a usar os identificadores imediatamente, como acontece com as IDs de banco de dados sequenciais regulares.

O algoritmo Hi/Lo descreve um mecanismo para obter um lote de IDs exclusivas de uma sequência de banco de dados relacionado. Essas IDs são seguras usar porque o banco de dados garante a exclusividade, portanto, não haverá nenhuma colisão entre usuários. Esse algoritmo é interessante por estes motivos:

- Ele não interrompe o padrão de unidade de trabalho.
- Ele obtém as IDs da sequência em lotes, para minimizar as viagens de ida e para o banco de dados.
- Ele gera um identificador legível por pessoas, ao contrário das técnicas que usam GUIDs.

EF Core dá suporte a [Hilo](#) com o `UseHilo` método, conforme mostrado no exemplo anterior.

### Mapear campos em vez de propriedades

Com esse recurso, disponível desde o EF Core 1.1, você pode mapear colunas para campos diretamente. É possível não usar as propriedades na classe de entidade e apenas mapear as colunas de uma tabela para os campos. Um caso de uso comum para isso seria os campos privados de algum estado interno que não precisam ser acessados de fora da entidade.

Você pode fazer isso com campos únicos ou também com coleções, como um campo `List<>`. Esse ponto já foi mencionado quando discutimos a modelagem das classes de modelo de domínio, mas aqui você pode ver como esse mapeamento é realizado com a configuração `PropertyAccessMode.Field` realçada no código anterior.

### Usar propriedades de sombra no EF Core, ocultas no nível da infraestrutura

As propriedades de sombra no EF Core são propriedades que não existem no modelo de classe de entidade. Os valores e os estados dessas propriedades são mantidos unicamente na classe [ChangeTracker](#) no nível da infraestrutura.

## Implementar o padrão de especificação de consulta

Conforme já foi apresentado na seção sobre design, o padrão de especificação de consulta é um padrão de design orientado por domínio projetado para ser o local em que você pode colocar a definição de uma consulta com uma lógica opcional de classificação e paginação.

O padrão de especificação de consulta define uma consulta em um objeto. Por exemplo, para encapsular uma consulta paginada que procura por alguns produtos, você poderia criar uma especificação `PagedProduct` que usasse os parâmetros de entrada necessários (`pageNumber`, `pageSize`, filtro, etc.). Então, qualquer método de repositório [geralmente uma sobrecarga `List()`] aceitaria uma `IQuerySpecification` e executaria a consulta esperada com base nessa especificação.

Um exemplo de uma interface de especificação genérica é o código a seguir de [eShopOnWeb](#).

```
// GENERIC SPECIFICATION INTERFACE
// https://github.com/dotnet-architecture/eShopOnWeb

public interface ISpecification<T>
{
    Expression<Func<T, bool>> Criteria { get; }
    List<Expression<Func<T, object>>> Includes { get; }
    List<string> IncludeStrings { get; }
}
```

Assim, a implementação de uma classe base de especificação genérica seria a seguinte.

```
// GENERIC SPECIFICATION IMPLEMENTATION (BASE CLASS)
// https://github.com/dotnet-architecture/eShopOnWeb

public abstract class BaseSpecification<T> : ISpecification<T>
{
    public BaseSpecification(Expression<Func<T, bool>> criteria)
    {
        Criteria = criteria;
    }
    public Expression<Func<T, bool>> Criteria { get; }

    public List<Expression<Func<T, object>>> Includes { get; } =
        new List<Expression<Func<T, object>>>();

    public List<string> IncludeStrings { get; } = new List<string>();

    protected virtual void AddInclude(Expression<Func<T, object>> includeExpression)
    {
        Includes.Add(includeExpression);
    }

    // string-based includes allow for including children of children
    // e.g. Basket.Items.Product
    protected virtual void AddInclude(string includeString)
    {
        IncludeStrings.Add(includeString);
    }
}
```

A especificação a seguir carrega uma única entidade de cesta dada a ID da cesta ou a ID do comprador ao qual a cesta pertence. Ele carregará a coleção da cesta de forma adiantada `Items`.

```
// SAMPLE QUERY SPECIFICATION IMPLEMENTATION

public class BasketWithItemsSpecification : BaseSpecification<Basket>
{
    public BasketWithItemsSpecification(int basketId)
        : base(b => b.Id == basketId)
    {
        AddInclude(b => b.Items);
    }

    public BasketWithItemsSpecification(string buyerId)
        : base(b => b.BuyerId == buyerId)
    {
        AddInclude(b => b.Items);
    }
}
```

E, finalmente, veja abaixo como um repositório do EF genérico pode usar uma especificação desse tipo para filtrar

e fazer o carregamento adiantado dos dados relacionados a um determinado tipo de entidade T.

```
// GENERIC EF REPOSITORY WITH SPECIFICATION
// https://github.com/dotnet-architecture/eShopOnWeb

public IEnumerable<T> List(ISpecification<T> spec)
{
    // fetch a Queryable that includes all expression-based includes
    var queryableResultWithIncludes = spec.Includes
        .Aggregate(_dbContext.Set<T>().AsQueryable(),
            (current, include) => current.Include(include));

    // modify the IQueryable to include any string-based include statements
    var secondaryResult = spec.IncludeStrings
        .Aggregate(queryableResultWithIncludes,
            (current, include) => current.Include(include));

    // return the result of the query using the specification's criteria expression
    return secondaryResult
        .Where(spec.Criteria)
        .AsEnumerable();
}
```

Além de encapsular a lógica de filtragem, a especificação pode especificar a forma dos dados a serem retornados, incluindo quais propriedades devem ser populadas.

Embora não seja recomendável retornar `IQueryable` de um repositório, é perfeitamente preciso usá-lo dentro do repositório para criar um conjunto de resultados. Você pode ver essa abordagem usada no método `List` acima, que usa expressões intermediárias `IQueryable` para criar a lista de inclusões da consulta antes de executar a consulta com os critérios da especificação na última linha.

## Recursos adicionais

- **Mapeamento de tabela**  
<https://docs.microsoft.com/ef/core/modeling/relational/tables>
- **Use HiLo para gerar chaves com Entity Framework Core**  
<https://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/>
- **Campos de backup**  
<https://docs.microsoft.com/ef/core/modeling/backing-field>
- **Steve Smith. Coleções encapsuladas no Entity Framework Core**  
<https://ardalis.com/encapsulated-collections-in-entity-framework-core>
- **Propriedades da sombra**  
<https://docs.microsoft.com/ef/core/modeling/shadow-properties>
- **O padrão de especificação**  
<https://deviq.com/specification-pattern/>

# Usar bancos de dados NoSQL como infraestrutura de persistência

10/09/2020 • 24 minutes to read • [Edit Online](#)

Ao usar bancos de dados NoSQL para a camada de dados da infraestrutura, normalmente, não se usa um ORM (mapeamento objeto-relacional) como o Entity Framework Core. Nesse caso, é possível usar a API fornecida pelo mecanismo NoSQL, como o Azure Cosmos DB, o MongoDB, o Cassandra, o RavenDB, o CouchDB ou as tabelas de Armazenamento do Azure.

No entanto, ao usar um banco de dados NoSQL, principalmente um banco de dados orientado a documentos como o Azure Cosmos DB, o CouchDB ou o RavenDB, a maneira de criar o modelo com agregações de DDD é parcialmente semelhante à maneira de fazer isso no EF Core, em relação à identificação de raízes agregadas, classes de entidade filha e classes de objeto de valor. Mas, por fim, a seleção do banco de dados realmente afetará o design.

Ao usar um banco de dados orientado a documentos, você implementa uma agregação como um único documento serializado em JSON ou em outro formato. No entanto, o uso do banco de dados é transparente do ponto de vista do código do modelo de domínio. Ao usar um banco de dados NoSQL, você ainda está usando classes de entidade e classes de raiz de agregação, mas com maior flexibilidade do que ao usar o EF Core porque a persistência não é relacional.

A diferença está em como persistir esse modelo. Se você implementar o modelo de domínio com base em classes de entidade POCO (objeto CRL básico), independentemente da persistência da infraestrutura, poderá parecer que é possível passar para uma infraestrutura de persistência diferente, até mesmo de relacional para NoSQL. No entanto, essa não deve ser a sua meta. Sempre há restrições e compensações nas diferentes tecnologias de bancos de dados, portanto, não é possível usar o mesmo modelo para bancos de dados relacionais ou NoSQL. Alterar os modelos de persistência não é tão fácil, pois as transações e as operações de persistência são muito diferentes.

Por exemplo, em um banco de dados orientado a documentos, é possível que uma raiz de agregação tenha diversas propriedades de coleção filhas. Em um banco de dados relacional, consultar várias propriedades de coleção filhas é algo de difícil otimização, pois o EF retorna a você uma instrução UNION ALL SQL. Ter o mesmo modelo de domínio para bancos de dados relacionais ou bancos de dados NoSQL não é simples e não é recomendado. Você realmente precisa criar seu modelo com uma compreensão de como os dados serão usados em cada banco de dados específico.

Um benefício de usar bancos de dados NoSQL é que as entidades são mais desnormalizadas, portanto, você não precisa definir um mapeamento de tabela. O modelo de domínio pode ser mais flexível do que ao usar um banco de dados relacional.

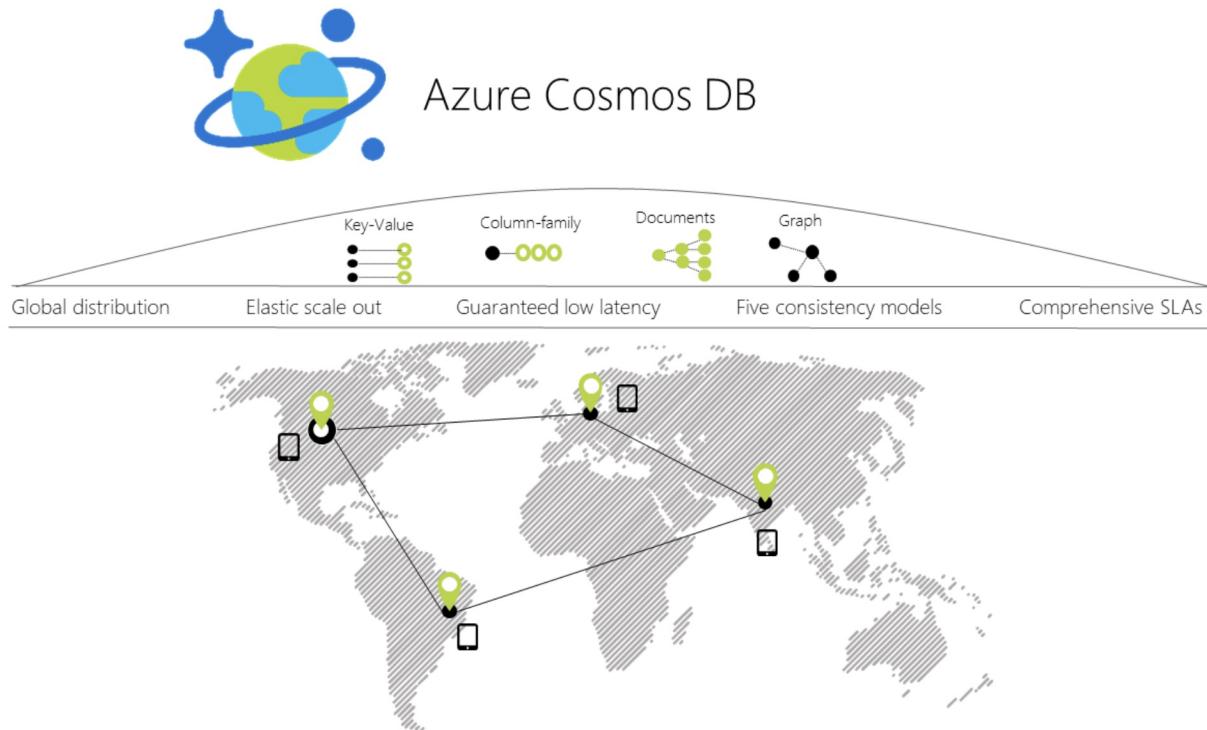
Ao criar o modelo de domínio baseado em agregações, passar para bancos de dados NoSQL e orientados a documentos pode ser ainda mais fácil do que usar um banco de dados relacional, porque as agregações criadas são semelhantes aos documentos serializados em um banco de dados orientado a documentos. Em seguida, você pode incluir nessas "bolsas" todas as informações que você pode precisar para essa agregação.

Por exemplo, o código JSON a seguir é um exemplo da implementação de uma agregação de pedido ao usar um banco de dados orientado a documentos. Ele é semelhante à agregação de pedido que foi implementada no exemplo eShopOnContainers, mas sem usar o Core EF.

```
{
  "id": "2017001",
  "orderDate": "2/25/2017",
  "buyerId": "1234567",
  "address": [
    {
      "street": "100 One Microsoft Way",
      "city": "Redmond",
      "state": "WA",
      "zip": "98052",
      "country": "U.S."
    }
  ],
  "orderItems": [
    {"id": 20170011, "productId": "123456", "productName": ".NET T-Shirt",
     "unitPrice": 25, "units": 2, "discount": 0},
    {"id": 20170012, "productId": "123457", "productName": ".NET Mug",
     "unitPrice": 15, "units": 1, "discount": 0}
  ]
}
```

## Introdução ao Azure Cosmos DB e à API nativa do Cosmos DB

O [Azure Cosmos DB](#) é o serviço de banco de dados distribuído globalmente da Microsoft para aplicativos críticos. O Azure Cosmos DB fornece [distribuição global de chave](#), [dimensionamento elástico de taxa de transferência](#) e [armazenamento](#) em todo o mundo, latências de milissegundos de dígito único no 99 ° percentil, [cinco níveis de consistência](#) bem definidos e garantia de alta disponibilidade, tudo apoiado por [SLAs líderes do setor](#). O Azure Cosmos DB [indexa dados automaticamente](#) sem a necessidade de lidar com o gerenciamento do esquema e do índice. Ele tem vários modelos e dá suporte a modelos de dados de colunas, grafos, valores-chave e documentos.



**Figura 7-19.** Distribuição global do Azure Cosmos DB

Ao usar um modelo C# para implementar a agregação a ser usada pela API do Azure Cosmos DB, a agregação pode ser semelhante às classes POCO do C# usadas com o EF Core. A diferença está na maneira de usá-las nas camadas de aplicativo e de infraestrutura, como no código a seguir:

```

// C# EXAMPLE OF AN ORDER AGGREGATE BEING PERSISTED WITH AZURE COSMOS DB API
// *** Domain Model Code ***
// Aggregate: Create an Order object with its child entities and/or value objects.
// Then, use AggregateRoot's methods to add the nested objects so invariants and
// logic is consistent across the nested properties (value objects and entities).

Order orderAggregate = new Order
{
    Id = "2017001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "P018009186470"
}

Address address = new Address
{
    Street = "100 One Microsoft Way",
    City = "Redmond",
    State = "WA",
    Zip = "98052",
    Country = "U.S."
}

orderAggregate.UpdateAddress(address);

OrderItem orderItem1 = new OrderItem
{
    Id = 20170011,
    ProductId = "123456",
    ProductName = ".NET T-Shirt",
    UnitPrice = 25,
    Units = 2,
    Discount = 0;
};

//Using methods with domain logic within the entity. No anemic-domain model
orderAggregate.AddOrderItem(orderItem1);
// *** End of Domain Model Code ***

// *** Infrastructure Code using Cosmos DB Client API ***
Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName,
    collectionName);

await client.CreateDocumentAsync(collectionUri, orderAggregate);

// As your app evolves, let's say your object has a new schema. You can insert
// OrderV2 objects without any changes to the database tier.
Order2 newOrder = GetOrderV2Sample("IdForSalesOrder2");
await client.CreateDocumentAsync(collectionUri, newOrder);

```

Você pode ver que a maneira de trabalhar com o modelo de domínio pode ser semelhante à maneira de usá-lo na camada do modelo de domínio quando a infraestrutura é o EF. Você ainda pode usar os mesmos métodos de raiz de agregação para garantir a consistência, as invariáveis e as validações na agregação.

No entanto, ao persistir o modelo para o banco de dados NoSQL, o código e a API serão radicalmente alterados em comparação com o código do EF Core ou com qualquer outro código relacionado a bancos de dados relacionais.

## Implementar o código do .NET direcionado ao MongoDB e ao Azure Cosmos DB

### Usar o Azure Cosmos DB de contêineres do .NET

Você pode acessar os bancos de dados Azure Cosmos DB do código do .NET em execução em contêineres, como

de qualquer outro aplicativo .NET. Por exemplo, os microsserviços Locations.API e Marketing.API no eShopOnContainers são implementados para que possam consumir bancos de dados Azure Cosmos DB.

No entanto, há uma limitação no Azure Cosmos DB do ponto de vista do ambiente de desenvolvimento do Docker. Embora haja um [emulador de Azure Cosmos DB](#) local que pode ser executado em um computador de desenvolvimento, ele dá suporte apenas ao Windows. Não há suporte para Linux e macOS.

Também há a possibilidade de executar esse emulador no Docker, mas apenas em contêineres do Windows, não com contêineres do Linux. Essa é uma handicap inicial para o ambiente de desenvolvimento se seu aplicativo for implantado como contêineres do Linux, já que, no momento, você não pode implantar contêineres do Linux e do Windows em Docker for Windows ao mesmo tempo. Todos os contêineres que estão sendo implantados precisam ser do Linux ou do Windows.

O modo de implantação mais simples e ideal para uma solução de Desenvolvimento/Teste é poder implantar os sistemas de banco de dados como contêineres juntamente com contêineres personalizados para que os ambientes de Desenvolvimento/Teste estejam sempre consistentes.

### **Usar a API do MongoDB para contêineres locais do Linux/Windows de Desenvolvimento/Teste além do Azure Cosmos DB**

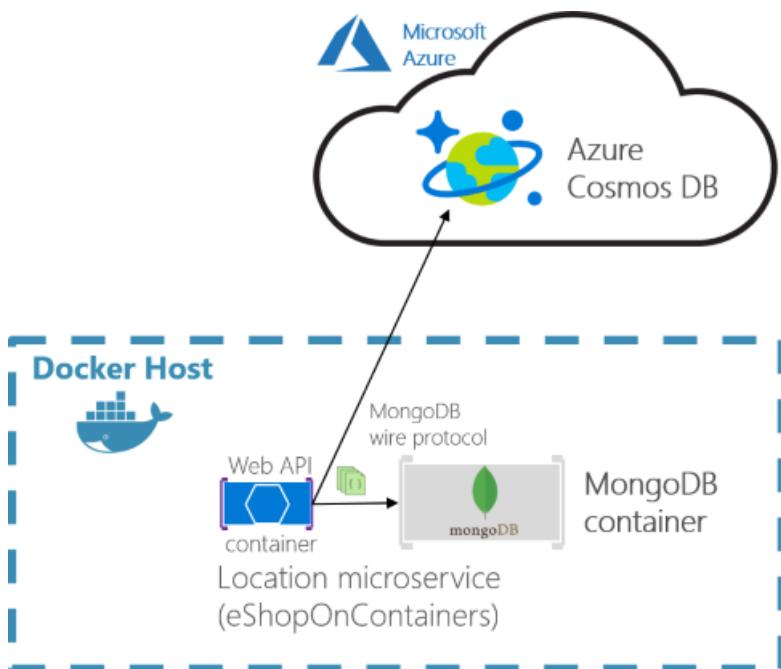
Os bancos de dados Cosmos DB são compatíveis com a API do MongoDB para .NET, e com o protocolo de transmissão nativo do MongoDB. Isso significa que, usando os drivers existentes, o aplicativo escrito para o MongoDB agora pode se comunicar com o Cosmos DB e usar os bancos de dados do Cosmos DB em vez dos bancos de dados do MongoDB, conforme é mostrado na Figura 7-20.



**Figura 7-20.** Usando a API e o protocolo do MongoDB para acessar o Azure Cosmos DB

Essa é uma abordagem muito conveniente para prova de conceitos em ambientes do Docker com contêineres do Linux, porque a [imagem do Docker do MongoDB](#) é uma imagem para várias arquiteturas, compatível com contêineres do Linux do Docker e do Windows do Docker.

Conforme é mostrado na imagem a seguir, usando a API do MongoDB, o eShopOnContainers permite contêineres do Windows e do Linux do MongoDB para o ambiente de desenvolvimento local, mas também é possível passar para uma solução de nuvem de PaaS escalonável, como o Azure Cosmos DB, simplesmente [alterando a cadeia de conexão do MongoDB para apontar para o Azure Cosmos DB](#).



**Figura 7-21.** eShopOnContainers usando contêineres do MongoDB para o ambiente de desenvolvimento ou para o Azure Cosmos DB para produção

O Azure Cosmos DB de produção seria executado na nuvem do Azure como um serviço de PaaS e escalonável.

Os contêineres do .NET Core personalizados podem ser executados em um host do Docker de desenvolvimento local (que esteja usando o Docker para Windows em um computador Windows 10) ou ser implantados em um ambiente de produção, como o Kubernetes no AKS do Azure ou no Azure Service Fabric. Nesse segundo ambiente, você implantaria apenas os contêineres personalizados do .NET Core, mas não o contêiner do MongoDB, já que usaria Azure Cosmos DB na nuvem para lidar com os dados em produção.

Um benefício claro de usar a API do MongoDB é que a solução pode ser executada em ambos os mecanismos de banco de dados, MongoDB ou Azure Cosmos DB, facilitando as migrações para diferentes ambientes. No entanto, às vezes, vale a pena usar uma API nativa (ou seja, a API nativa do Cosmos DB) para aproveitar ao máximo os recursos de um mecanismo de banco de dados específico.

Para obter mais comparações entre o simples uso do MongoDB ou do Cosmos DB na nuvem, consulte [Benefícios de usar o Azure Cosmos DB, nesta página](#).

#### **Analise sua abordagem para aplicativos de produção: API do MongoDB versus API de Cosmos DB**

No eShopOnContainers, estamos usando a API do MongoDB porque nossa prioridade era fundamental para ter um ambiente de desenvolvimento/teste consistente usando um banco de dados NoSQL que também poderia funcionar com Azure Cosmos DB.

No entanto, se você planeja usar a API do MongoDB para acessar o Azure Cosmos DB no Azure para aplicativos de produção, analise as diferenças de recursos e de desempenho ao usar a API do MongoDB para acessar bancos de dados Azure Cosmos DB em comparação com o uso da API nativa do Azure Cosmos DB. Se for semelhante, você poderá usar a API do MongoDB e obter o benefício de permitir dois mecanismos de banco de dados NoSQL simultaneamente.

Você também pode usar clusters do MongoDB como o banco de dados de produção na nuvem do Azure, também, com o [serviço MongoDB do Azure](#). Mas esse não é um serviço de PaaS fornecido pela Microsoft. Nesse caso, o Azure está apenas hospedando essa solução proveniente do MongoDB.

Basicamente, essa é apenas uma declaração informando que você não deve sempre usar a API do MongoDB em Azure Cosmos DB, como fizemos no eShopOnContainers porque ela era uma opção conveniente para contêineres do Linux. A decisão deve ser baseada as necessidades específicas e nos testes que você precisa fazer para o aplicativo de produção.

## O código: usar a API MongoDB em aplicativos .NET Core

A API do MongoDB para .NET baseia-se em pacotes NuGet que você precisa adicionar nos projetos, como no projeto Locations.API mostrado na figura a seguir.

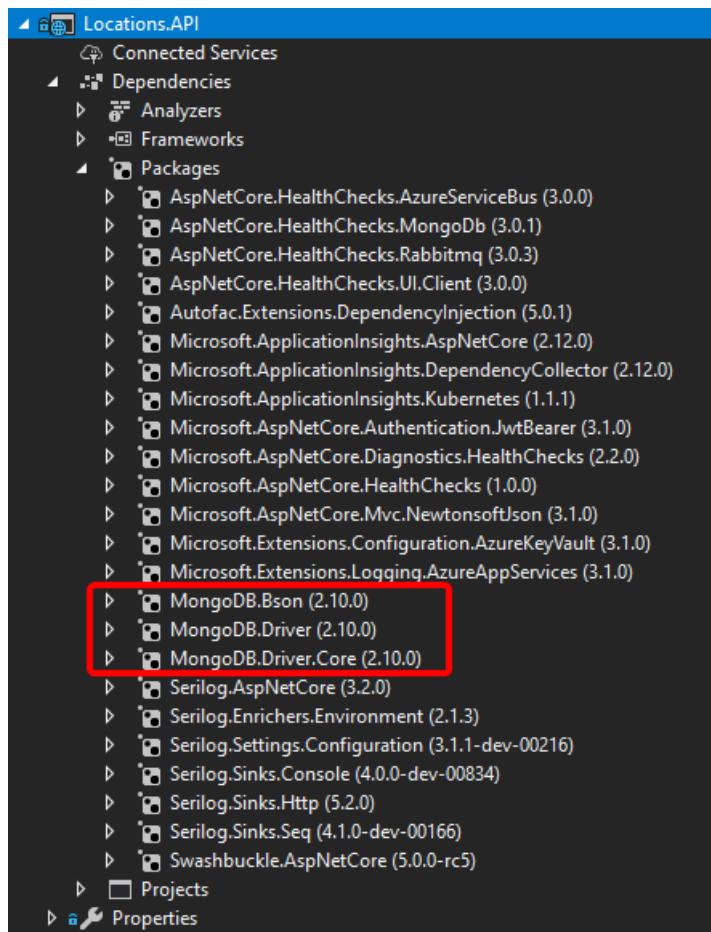


Figura 7-22. Referências de pacotes NuGet da API do MongoDB em um projeto do .NET Core

Vamos examinar o código nas seções a seguir.

### Um modelo usado pela API do MongoDB

Primeiro, você precisa definir um modelo que conterá os dados provenientes do banco de dado no espaço de memória do seu aplicativo. Aqui está um exemplo do modelo usado para locais em eShopOnContainers.

```

using MongoDB.Bson;
using MongoDB.Bson.Serialization.Attributes;
using MongoDB.Driver.GeoJsonObjectModel;
using System.Collections.Generic;

public class Locations
{
    [BsonId]
    [BsonRepresentation(BsonType.ObjectId)]
    public string Id { get; set; }
    public int LocationId { get; set; }
    public string Code { get; set; }
    [BsonRepresentation(BsonType.ObjectId)]
    public string Parent_Id { get; set; }
    public string Description { get; set; }
    public double Latitude { get; set; }
    public double Longitude { get; set; }
    public GeoJsonPoint<GeoJson2DGeographicCoordinates> Location
        { get; private set; }
    public GeoJsonPolygon<GeoJson2DGeographicCoordinates> Polygon
        { get; private set; }
    public void SetLocation(double lon, double lat) => SetPosition(lon, lat);
    public void SetArea(List<GeoJson2DGeographicCoordinates> coordinatesList)
        => SetPolygon(coordinatesList);

    private void SetPosition(double lon, double lat)
    {
        Latitude = lat;
        Longitude = lon;
        Location = new GeoJsonPoint<GeoJson2DGeographicCoordinates>(
            new GeoJson2DGeographicCoordinates(lon, lat));
    }

    private void SetPolygon(List<GeoJson2DGeographicCoordinates> coordinatesList)
    {
        Polygon = new GeoJsonPolygon<GeoJson2DGeographicCoordinates>(
            new GeoJsonPolygonCoordinates<GeoJson2DGeographicCoordinates>(
                new GeoJsonLinearRingCoordinates<GeoJson2DGeographicCoordinates>(
                    coordinatesList)));
    }
}

```

Você pode ver que há alguns atributos e tipos provenientes dos pacotes NuGet do MongoDB.

Os bancos de dados NoSQL normalmente são muito bem adequados para trabalhar com os dados hierárquicos não relacionais. Neste exemplo, estamos usando tipos do MongoDB criados especialmente para localizações geográficas, como `GeoJson2DGeographicCoordinates`.

#### **Recuperar o banco de dados e a coleção**

No eShopOnContainers, criamos um contexto de banco de dados personalizado no qual podemos implementar o código para recuperar o banco de dados e as MongoCollections, como no código a seguir.

```

public class LocationsContext
{
    private readonly IMongoDatabase _database = null;

    public LocationsContext(IOptions<LocationSettings> settings)
    {
        var client = new MongoClient(settings.Value.ConnectionString);
        if (client != null)
            _database = client.GetDatabase(settings.Value.Database);
    }

    public IMongoCollection<Locations> Locations
    {
        get
        {
            return _database.GetCollection<Locations>("Locations");
        }
    }
}

```

### Recuperar os dados

No código C#, como nos controladores de API Web ou na implementação personalizada de repositórios, você pode escrever um código semelhante ao seguinte ao consultar por meio da API do MongoDB. Observe que o objeto `_context` é uma instância da classe `LocationsContext` anterior.

```

public async Task<Locations> GetAsync(int locationId)
{
    var filter = Builders<Locations>.Filter.Eq("LocationId", locationId);
    return await _context.Locations
        .Find(filter)
        .FirstOrDefaultAsync();
}

```

### Use uma variável de ambiente no arquivo `docker-compose.override.yml` para a cadeia de conexão do MongoDB

Ao criar um objeto do MongoClient, ele precisa de um parâmetro fundamental, que é exatamente o parâmetro `ConnectionString`, apontando para o banco de dados certo. No caso do eShopOnContainers, a cadeia de conexão pode apontar para um contêiner do Docker do MongoDB local ou para um banco de dados de Azure Cosmos DB de "produção". Essa cadeia de conexão vem de variáveis de ambiente definidas em arquivos `docker-compose.override.yml` usados ao implantar com o docker-compose ou o Visual Studio, como no código yml a seguir.

```

# docker-compose.override.yml
version: '3.4'
services:
    # Other services
    locations-api:
        environment:
            # Other settings
            - ConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}

```

A variável de ambiente `ConnectionString` é resolvida desta forma: se a variável global `ESHOP_AZURE_COSMOSDB` estiver definida no arquivo `.env` com a cadeia de conexão do Azure Cosmos DB, ela o usará para acessar o banco de dados Azure Cosmos DB na nuvem. Se não estiver definido, ele utilizará o `mongodb://nosqldata` valor e usará o contêiner de desenvolvimento do MongoDB.

O código a seguir mostra o arquivo `.env` com a variável de ambiente global da cadeia de conexão do Azure Cosmos DB, conforme implementado no eShopOnContainers:

```

# .env file, in eShopOnContainers root folder
# Other Docker environment variables

ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost
ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=<YourDockerHostIP>

#ESHOP_AZURE_COSMOSDB=<YourAzureCosmosDBConnData>

#Other environment variables for additional Azure infrastructure assets
#ESHOP_AZURE_REDIS_BASKET_DB=<YourAzureRedisBasketInfo>
#ESHOP_AZURE_STORAGE_CATALOG_URL=<YourAzureStorage_Catalog_BLOB_URL>
#ESHOP_AZURE_SERVICE_BUS=<YourAzureServiceBusInfo>

```

Remova a marca de comentário da linha de ESHOP\_AZURE\_COSMOSDB e atualize-a com a cadeia de conexão Azure Cosmos DB obtida do portal do Azure, conforme explicado em [conectar um aplicativo MongoDB a Azure Cosmos DB](#).

Se a `ESHOP_AZURE_COSMOSDB` variável global estiver vazia, o que significa que ela está comentada no `.env` arquivo, o contêiner usará uma cadeia de conexão padrão do MongoDB. Essa cadeia de conexão aponta para o contêiner local do MongoDB implantado em eShopOnContainers que é nomeado `nosqldata` e definido no arquivo Docker-Compose, conforme mostrado no seguinte código. yml:

```

# docker-compose.yml
version: '3.4'
services:
  # ...Other services...
  nosqldata:
    image: mongo

```

#### Recursos adicionais

- **Modelando dados de documentos para bancos de dados NoSQL**  
[/azure/cosmos-db/modeling-data](#)
- **Vaughn Vernon. O armazenamento agregado ideal de design controlado por domínio?**  
<https://kalele.io/blog-posts/the-ideal-domain-driven-design-aggregate-store/>
- **Introdução à Azure Cosmos DB: API para MongoDB**  
[/azure/cosmos-db/mongodb-introduction](#)
- **Azure Cosmos DB: compilar um aplicativo Web da API do MongoDB com o .NET e o portal do Azure**  
[/azure/cosmos-db/create-mongodb-dotnet](#)
- **Usar o emulador de Azure Cosmos DB para desenvolvimento e teste locais**  
[/azure/cosmos-db/local-emulator](#)
- **Conectar um aplicativo MongoDB ao Azure Cosmos DB**  
[/azure/cosmos-db/connect-mongodb-account](#)
- **A imagem do Docker do emulador Cosmos DB (contêiner do Windows)**  
<https://hub.docker.com/r/microsoft/azure-cosmosdb-emulator/>
- **A imagem do Docker do MongoDB (contêiner do Linux e do Windows)**  
[https://hub.docker.com/\\_/mongo/](https://hub.docker.com/_/mongo/)
- **Usar MongoChef (Studio 3T) com uma conta Azure Cosmos DB: API para MongoDB**  
[/azure/cosmos-db/mongodb-mongochef](#)

[ANTERIOR](#)

[AVANÇAR](#)

# Projetar a camada de aplicativos de microsserviço e a API Web

09/04/2020 • 3 minutes to read • [Edit Online](#)

## Usar princípios SOLID e Injeção de Dependência

Os princípios SOLID são técnicas críticas para serem usadas em qualquer aplicativo moderno e crítico, como desenvolver um microsserviço com padrões DDD. SOLID é um acrônimo que agrupa cinco princípios fundamentais:

- Princípio da Responsabilidade única
- Princípio do aberto/fechado
- Princípio da Substituição de Liskov
- Princípio da Segregação de interface
- Princípio da Inversão de dependência

SOLID trata-se da maneira como você cria as camadas internas do microsserviço ou do aplicativo e do desacoplamento das dependências entre eles. Não está relacionado ao domínio, mas ao projeto técnico do aplicativo. O princípio final, o princípio de Inversão de dependência, permite a você desacoplar a camada de infraestrutura do restante das camadas, que permite uma melhor implementação desacoplada das camadas DDD.

DI (Injeção de Dependência) é uma maneira de implementar o princípio de Inversão de Dependência. É uma técnica para obter um acoplamento flexível entre objetos e suas dependências. Em vez de criar uma instância de colaboradores diretamente ou usar referências estáticas (ou seja, usar novos...), os objetos de que uma classe precisa para executar suas ações são fornecidos (ou "injetados") na classe. Geralmente, as classes declararão suas dependências por meio de seu construtor, possibilitando que elas sigam o princípio de Dependências Explícitas. Geralmente, a Injeção de Dependência baseia-se em contêineres IoC (Inversão de Controle) específicos. O ASP.NET Core fornece um contêiner IoC interno simples, mas também é possível usar seu contêiner IoC favorito, como Autofac ou Ninject.

Seguindo os princípios SOLID, as classes naturalmente tenderão a ser pequenas, bem fatoradas e facilmente testadas. Mas como é possível saber se muitas dependências estão sendo injetadas suas classes? Se você usar a DI por meio do construtor, será fácil detectar isso apenas observando o número de parâmetros para o construtor. Se houver mais dependências, isso geralmente será um sinal (um [code smell](#)) de que sua classe está tentando fazer muito mais e está provavelmente violando o princípio de Responsabilidade única.

Seria preciso outro guia para abordar o SOLID em detalhes. Portanto, este guia exige que você tenha apenas um conhecimento mínimo sobre estes tópicos.

### Recursos adicionais

- **SÓLIDO: Princípios Fundamentais da OOP**  
<https://deviq.com/solid/>
- **Inversão de Recipientes de Controle e o padrão de injeção de dependência**  
<https://martinfowler.com/articles/injection.html>
- **Steve Smith. Novo é Cola**  
<https://ardalis.com/new-is-glue>

PRÓXIMO

ANTERIOR

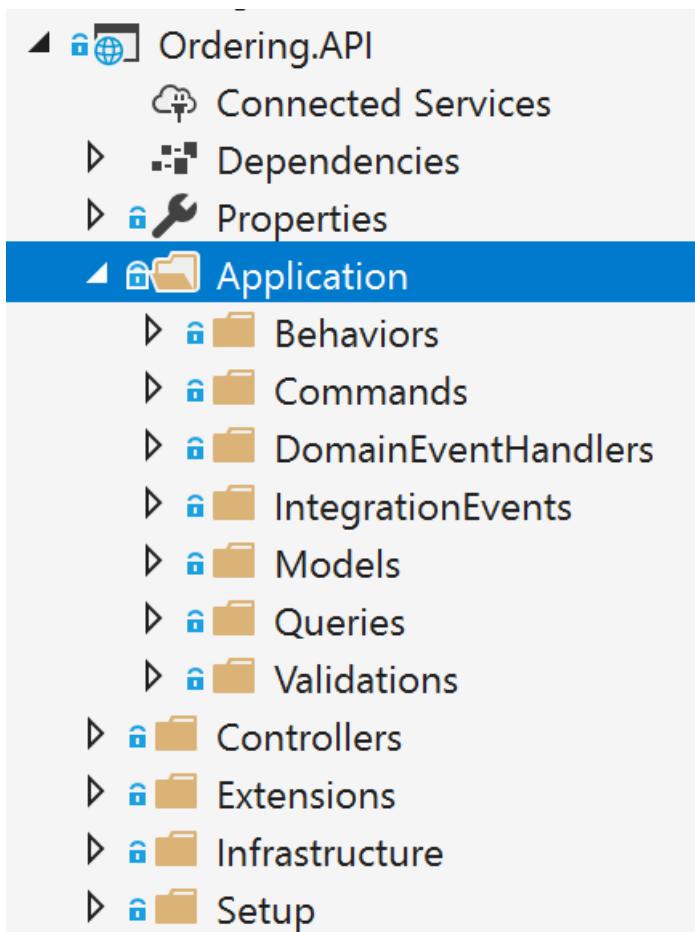
# Implementar a camada de aplicativos de microsserviço usando a API Web

10/09/2020 • 58 minutes to read • [Edit Online](#)

## Usar a injeção de dependência para injetar objetos de infraestrutura em sua camada de aplicativo

Conforme mencionado anteriormente, a camada de aplicativo pode ser implementada como parte do artefato (assembly) que você está criando, tal como em um projeto de API Web ou um projeto de aplicativo Web do MVC. No caso de um microsserviço criado com o ASP.NET Core, a camada de aplicativo geralmente será a biblioteca da API Web. Se quiser separar o que é proveniente do ASP.NET Core (a infraestrutura e os controladores) do código personalizado da camada de aplicativo, você também poderá colocar a camada de aplicativo em uma biblioteca de classes separada, mas isso é opcional.

Por exemplo, o código da camada de aplicativo do microsserviço de ordenação é implementado diretamente como parte do projeto `Ordering.API` (um projeto da API Web ASP.NET Core), como mostrado na Figura 7-23.



O Gerenciador de Soluções exibição do microsserviço de classificação. API, mostrando as subpastas na pasta do aplicativo: comportamentos, comandos, DomainEventHandlers, IntegrationEvents, modelos, consultas e validações.

Figura 7-23. A camada de aplicativo no projeto Ordering.API da API Web ASP.NET Core

O ASP.NET Core inclui um [contêiner interno de IoC](#) simples (representado pela interface `IServiceProvider`) que é compatível com a injeção de construtor por padrão, e o ASP.NET disponibiliza alguns serviços por meio da DI

(injeção de dependência). O ASP.NET Core usa o termo *serviço* para qualquer um dos tipos que você registra e que serão injetados pela DI. Você configura os serviços internos do contêiner no método ConfigureServices na classe Startup do seu aplicativo. As dependências são implementadas nos serviços que são necessários para um tipo e que você registra no contêiner de IoC.

Normalmente, você deseja injetar dependências que implementam objetos de infraestrutura. Uma dependência típica para injetar é um repositório. Mas você poderá injetar qualquer outra dependência de infraestrutura que tiver. Para implementações mais simples, você injeta diretamente o objeto de padrão da Unidade de Trabalho (o objeto DbContext do EF), porque o DbContext também é a implementação dos objetos de persistência da sua infraestrutura.

Veja no exemplo a seguir como o .NET Core está injetando os objetos de repositório necessários por meio do construtor. A classe é um manipulador de comando, que será abordado na próxima seção.

```

public class CreateOrderCommandHandler
    : IRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;
    private readonly IOrderingIntegrationEventService _orderingIntegrationEventService;
    private readonly ILogger<CreateOrderCommandHandler> _logger;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService identityService,
        ILogger<CreateOrderCommandHandler> logger)
    {
        _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
        _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new
        ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<bool> Handle(CreateOrderCommand message, CancellationToken cancellationToken)
    {
        // Add Integration event to clean the basket
        var orderStartedIntegrationEvent = new OrderStartedIntegrationEvent(message.UserId);
        await _orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegrationEvent);

        // Add/Update the Buyer AggregateRoot
        // DDD patterns comment: Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate
        var address = new Address(message.Street, message.City, message.State, message.Country,
            message.ZipCode);
        var order = new Order(message.UserId, message.UserName, address, message.CardTypeId,
            message.CardNumber, message.CardSecurityNumber, message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice, item.Discount,
                item.PictureUrl, item.Units);
        }

        _logger.LogInformation("----- Creating Order - Order: {@Order}", order);

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);
    }
}

```

A classe usa os repositórios injetados para executar a transação e persistir as alterações de estado. Não importa se a classe é um manipulador de comandos, um método de controlador da API Web ASP.NET Core ou um [Serviço de aplicativo DDD](#). Em última análise, ela uma classe simples que usa repositórios, entidades de domínio e outras coordenações de aplicativos de forma semelhante a um manipulador de comandos. A injeção de dependência funciona da mesma forma para todas as classes mencionadas, como no exemplo que usa a injeção de dependência com base no construtor.

### Registrar tipos e interfaces ou abstrações da implementação de dependência

Antes de usar os objetos injetados por meio de construtores, você precisa saber em que lugar registrar as

interfaces e classes que produzem os objetos injetados nas classes do aplicativo por meio da DI. (Como na DI baseada no construtor, conforme mostrado anteriormente).

#### Usar o contêiner interno de IoC fornecido pelo ASP.NET Core

Ao usar o contêiner interno de IoC fornecido pelo ASP.NET Core, você registra os tipos que deseja injetar no método ConfigureServices no arquivo Startup.cs, como no código a seguir:

```
// Registration of types into ASP.NET Core built-in container
public void ConfigureServices(IServiceCollection services)
{
    // Register out-of-the-box framework services.
    services.AddDbContext<CatalogContext>(c =>
        c.UseSqlServer(Configuration["ConnectionString"]),
        ServiceLifetime.Scoped);

    services.AddMvc();
    // Register custom application dependencies.
    services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();
}
```

O padrão mais comum ao registrar tipos em um contêiner de IoC é registrar um par de tipos: uma interface e a respectiva classe de implementação. Então, quando solicita um objeto do contêiner de IoC por meio de nenhum construtor, você solicita um objeto de um determinado tipo de interface. Assim, como visto no exemplo anterior, a última linha indica que, quando qualquer um dos seus construtores tiver uma dependência em IMyCustomRepository (interface ou abstração), o contêiner de IoC injetará uma instância da classe de implementação MyCustomSQLServerRepository.

#### Usar a biblioteca Scrutor para registro automático de tipos

Ao usar a DI no .NET Core, talvez seja interessante verificar um assembly e registrar automaticamente seus tipos por convenção. Este recurso não está disponível atualmente no ASP.NET Core. No entanto, você pode usar a biblioteca [Scrutor](#) para fazer isso. Essa abordagem é conveniente quando você tem muitos tipos que precisam ser registrados no contêiner de IoC.

#### Recursos adicionais

- **Matthew King. Registrando serviços com o Scrutor**  
<https://www.mking.net/blog/registering-services-with-scrutor>
- **Kristian Hellang. Scrutor.** Repositório do GitHub.  
<https://github.com/khellang/Scrutor>

#### Usar o Autofac como um contêiner de IoC

Você também pode usar mais contêineres de IoC e conectá-los no pipeline do ASP.NET Core, como no microsserviço de ordenação em eShopOnContainers, que usa o [Autofac](#). Ao usar Autofac, você normalmente registra os tipos por meio de módulos, que permitem dividir os tipos de registro entre vários arquivos, dependendo do local em que seus tipos estão, assim como os tipos de aplicativos podem ser distribuídos entre várias bibliotecas de classes.

Por exemplo, a seguir está o [módulo de aplicativo do Autofac](#) para o projeto de [API Web Ordering API](#) com os tipos que você pode injetar.

```

public class ApplicationModule : Autofac.Module
{
    public string QueriesConnectionString { get; }
    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }

    protected override void Load(ContainerBuilder builder)
    {
        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>()
            .InstancePerLifetimeScope();
        builder.RegisterType<BuyerRepository>()
            .As<IBuyerRepository>()
            .InstancePerLifetimeScope();
        builder.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();
        builder.RegisterType<RequestManager>()
            .As< IRequestManager>()
            .InstancePerLifetimeScope();
    }
}

```

O Autofac também tem um recurso para [digitalizar assemblies e registrar tipos pelas convenções de nome](#).

O processo e os conceitos de registro são muito semelhantes à forma pela qual você registra tipos com o contêiner de IoC interno do ASP.NET Core, mas a sintaxe, ao usar Autofac, é um pouco diferente.

No código de exemplo, a abstração `IOrderRepository` é registrada juntamente com a classe de implementação `OrderRepository`. Isso significa que, sempre que um construtor estiver declarando uma dependência por meio da abstração ou interface `IOrderRepository`, o contêiner de IoC injetará uma instância da classe `OrderRepository`.

O tipo de escopo da instância determina como uma instância é compartilhada entre as solicitações para o mesmo serviço ou dependência. Quando uma solicitação é feita a uma dependência, o contêiner de IoC pode retornar o seguinte:

- Uma única instância por escopo de tempo de vida (conhecida no contêiner de IoC do ASP.NET Core como *com escopo*).
- Uma nova instância por dependência (conhecida no contêiner de IoC do ASP.NET Core como *transitória*).
- Uma única instância compartilhada entre todos os objetos que usam o contêiner de IoC (conhecida no contêiner de IoC do ASP.NET Core como *singleton*).

#### Recursos adicionais

- **Introdução à injeção de dependência no ASP.NET Core**  
<https://docs.microsoft.com/aspnet/core/fundamentals/dependency-injection>
- **Autofac.** Documentação oficial.  
<https://docs.autofac.org/en/latest/>
- **Comparando os tempos de vida do serviço de contêiner de IoC do ASP.NET Core com os escopos de instância de contêiner de IoC do Autofac – Cesar de la Torre.**  
<https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/>

## Implementar os padrões de Comando e Manipulador de Comandos

No exemplo de DI por meio de construtor, mostrado na seção anterior, o contêiner de IoC injetou repositórios por

meio de um construtor em uma classe. Mas em que local eles foram exatamente injetados? Em uma API Web simples (por exemplo, o microserviço de catálogo em eShopOnContainers), você os injeta no nível dos controladores MVC, em um construtor de controlador, como parte do pipeline de solicitação de ASP.NET Core. Entretanto, no código inicial desta seção (a classe `CreateOrderCommandHandler` do serviço Ordering.API em eShopOnContainers), a injeção de dependências é feita por meio do construtor de um manipulador comandos específico. Vamos explicar o que é um manipulador de comandos e por que você o usaria.

O padrão Command é intrinsecamente relacionado ao padrão CQRS, apresentado anteriormente neste guia. O CQRS tem dois lados. A primeira área é a de consultas, usando consultas simplificadas com o micro ORM [Dapper](#), explicado anteriormente. A segunda área é a de comandos, os quais são o ponto de partida para transações e o canal de entrada do serviço.

Como mostra a Figura 7-24, o padrão é baseado em aceitar comandos do lado do cliente, processá-los com base nas regras de modelo de domínio e, finalmente, persistir os Estados com transações.

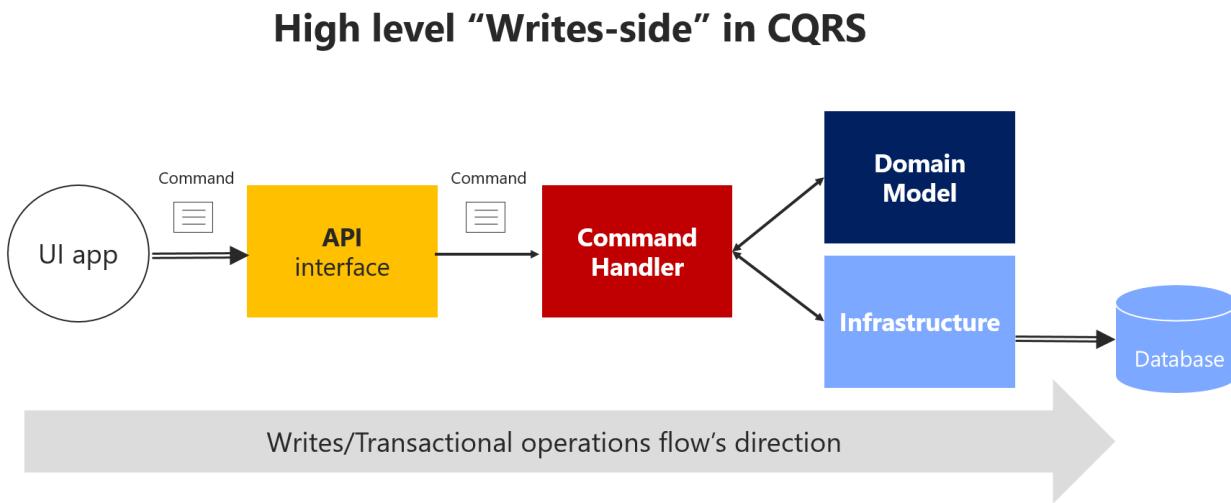


Figura 7-24. Exibição de alto nível dos comandos ou "lado transacional" em um padrão CQRS

A Figura 7-24 mostra que o aplicativo de interface do usuário envia um comando por meio da API que chega a um `CommandHandler`, que depende do modelo de domínio e da infraestrutura, para atualizar o banco de dados.

#### A classe de comando

Um comando é uma solicitação ao sistema para executar uma ação que altera o estado do sistema. Os comandos são imperativos e devem ser processados apenas uma vez.

Como são imperativos, os comandos geralmente são nomeados com um verbo no modo imperativo, por exemplo, "criar" ou "atualizar", e eles podem incluir o tipo de agregação, como `CreateOrderCommand`. Ao contrário de um evento, um comando não é um fato do passado; ele é apenas uma solicitação e, portanto, pode ser recusado.

Os comandos podem se originar na interface do usuário, como resultado do início de uma solicitação por um usuário ou de um gerenciador de processos, quando ele está instruindo uma agregação a executar uma ação.

Uma característica importante de um comando é que ele deve ser processado apenas uma vez por um único destinatário. Isso porque um comando é uma ação ou transação única que você deseja executar no aplicativo. Por exemplo, o mesmo comando de criação de pedido não deve ser processado mais de uma vez. Essa é uma importante diferença entre comandos e eventos. Os eventos podem ser processados várias vezes, porque muitos sistemas ou microsserviços podem estar interessados no evento.

Além disso, é importante que um comando seja processado apenas uma vez, caso ele não seja idempotente. Um comando será idempotente se ele puder ser executado várias vezes sem alterar o resultado, devido à natureza do comando ou por causa da forma como o sistema manipula o comando.

É uma boa prática fazer seus comandos e atualizações idempotentes quando fizer sentido nas regras de negócios

e nas invariáveis de seu domínio. Assim, para usar o mesmo exemplo, se por algum motivo (lógica de repetição, hackers, etc.) o mesmo comando CreateOrder chegar até seu sistema diversas vezes, você deverá ser capaz de identificá-lo e ter a certeza de não criar vários pedidos. Para fazer isso, você precisa anexar algum tipo de identidade nas operações e identificar se o comando ou a atualização já foi processada.

Você envia um comando a um único destinatário; você não publica um comando. A publicação serve para eventos que declaram um fato, ou seja, que algo aconteceu e pode ser interessante para destinatários de eventos. No caso de eventos, o publicador não tem preocupações sobre quais destinatários recebem o evento ou o que eles fazem com isso. No entanto, os eventos de domínio ou de integração são outra história, já apresentada nas seções anteriores.

Um comando é implementado com uma classe que contém campos de dados ou coleções com todas as informações necessárias para executar esse comando. Um comando é um tipo especial de DTO (Objeto de Transferência de Dados), usado especificamente para solicitar alterações ou transações. O próprio comando baseia-se estritamente nas informações necessárias para processar o comando e nada mais.

O exemplo a seguir mostra a `CreateOrderCommand` classe simplificada. Este é um comando imutável que é usado no microsserviço de ordenação em eShopOnContainers.

```
// DDD and CQRS patterns comment: Note that it is recommended to implement immutable Commands
// In this case, its immutability is achieved by having all the setters as private
// plus only being able to update the data just once, when creating the object through its constructor.
// References on Immutable Commands:
// http://cqrsshipped.com/2012/06/immutable-command-pattern/
// https://docs.spine3.org/motivation/immutability.html
// http://blog.gauffin.org/2012/06/griffin-container-introducing-command-support/
// https://docs.microsoft.com/dotnet/csharp/programming-guide/classes-and-structs/how-to-implement-a-lightweight-class-with-auto-implemented-properties

[DataContract]
public class CreateOrderCommand
    : IRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;

    [DataMember]
    public string UserId { get; private set; }

    [DataMember]
    public string UserName { get; private set; }

    [DataMember]
    public string City { get; private set; }

    [DataMember]
    public string Street { get; private set; }

    [DataMember]
    public string State { get; private set; }

    [DataMember]
    public string Country { get; private set; }

    [DataMember]
    public string ZipCode { get; private set; }

    [DataMember]
    public string CardNumber { get; private set; }

    [DataMember]
    public string CardHolderName { get; private set; }

    [DataMember]
```

```

public DateTime CardExpiration { get; private set; }

[DataMember]
public string CardSecurityNumber { get; private set; }

[DataMember]
public int CardTypeId { get; private set; }

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

public CreateOrderCommand()
{
    _orderItems = new List<OrderItemDTO>();
}

public CreateOrderCommand(List<BasketItem> basketItems, string userId, string userName, string city,
string street, string state, string country, string zipcode,
string cardNumber, string cardHolderName, DateTime cardExpiration,
string cardSecurityNumber, int cardTypeId) : this()
{
    _orderItems = basketItems.ToOrderItemsDTO().ToList();
    UserId = userId;
    UserName = userName;
    City = city;
    Street = street;
    State = state;
    Country = country;
    ZipCode = zipcode;
    CardNumber = cardNumber;
    CardHolderName = cardHolderName;
    CardExpiration = cardExpiration;
    CardSecurityNumber = cardSecurityNumber;
    CardTypeId = cardTypeId;
    CardExpiration = cardExpiration;
}
}

public class OrderItemDTO
{
    public int ProductId { get; set; }

    public string ProductName { get; set; }

    public decimal UnitPrice { get; set; }

    public decimal Discount { get; set; }

    public int Units { get; set; }

    public string PictureUrl { get; set; }
}
}

```

Basicamente, a classe de comando contém todos os dados necessários para realizar uma transação comercial, usando os objetos do modelo de domínio. Assim, os comandos são simplesmente estruturas de dados que contêm dados somente leitura e nenhum comportamento. O nome do comando indica sua finalidade. Em várias linguagens, como C#, os comandos são representados como classes, mas eles não são verdadeiramente classes, no real sentido de serem orientados a objetos.

Como uma característica adicional, os comandos são imutáveis, porque o uso esperado é que eles sejam processados diretamente pelo modelo de domínio. Eles não precisam ser alterados durante o tempo de vida projetado. Em uma classe C#, a imutabilidade pode ser obtida não tendo nenhum setter ou outro método que altere o estado interno.

Tenha em mente que, se você pretender ou esperar que os comandos passem por um processo de serialização/desserialização, as propriedades deverão ter um setter particular e o `[DataMember]` atributo (ou `[JsonProperty]`). Caso contrário, o desserializador não poderá reconstruir o objeto no destino com os valores necessários. Você também pode usar Propriedades verdadeiramente somente leitura se a classe tiver um construtor com parâmetros para todas as propriedades, com a Convenção de nomenclatura camelCase comum e anotar o construtor como `[JsonConstructor]`. No entanto, essa opção requer mais código.

Por exemplo, a classe de comando para a criação de um pedido é, provavelmente, semelhante em relação aos dados para o pedido que você deseja criar, mas é provável que você não precise dos mesmos atributos. Por exemplo, não `CreateOrderCommand` tem uma ID de pedido, porque a ordem ainda não foi criada.

Muitas classes de comando podem ser simples, exigindo apenas alguns campos de algum estado que tenha que ser alterado. Esse seria o caso se você estiver apenas alterando o status de um pedido de "em processo" para "pago" ou "enviado" usando um comando semelhante ao seguinte:

```
[DataContract]
public class UpdateOrderStatusCommand
    : IRequest<bool>
{
    [DataMember]
    public string Status { get; private set; }

    [DataMember]
    public string OrderId { get; private set; }

    [DataMember]
    public string BuyerIdentityGuid { get; private set; }
}
```

Alguns desenvolvedores criam os objetos de solicitação da interface do usuário separados dos DTOs do comando, mas essa é apenas uma questão de preferência. É uma separação entediante, sem muito valor adicional, e os objetos são quase exatamente a mesma forma. Por exemplo, no eShopOnContainers, alguns comandos vêm diretamente do lado do cliente.

### A classe do manipulador de comandos

Você deve implementar uma classe manipuladora de comandos específica para cada comando. É assim que o padrão funciona, e é onde você usará o objeto de comando, os objetos de domínio e os objetos de repositório de infraestrutura. O manipulador de comandos é, na verdade, a essência da camada de aplicativo em termos de CQRS e DDD. No entanto, toda a lógica de domínio deve estar contida nas classes de domínio — dentro das raízes de agregação (entidades raiz), entidades filho ou [serviços de domínio](#), mas não dentro do manipulador de comandos, que é uma classe da camada de aplicativo.

A classe de manipulador de comando oferece um forte ponto de partida no caminho para alcançar o SRP (Princípio de Responsabilidade Único) mencionado em uma seção anterior.

Um manipulador de comandos recebe um comando e obtém um resultado da agregação que é usada. O resultado deverá ser a execução bem-sucedida do comando ou uma exceção. No caso de uma exceção, o estado do sistema deve permanecer inalterado.

O manipulador de comandos geralmente realiza as seguintes etapas:

- Ele recebe o objeto de comando, como um DTO (do [mediador](#) ou de outro objeto da infraestrutura).
- Ele verifica se o comando é válido (se não tiver sido validado pelo mediador).
- Ele cria a instância da raiz de agregação que é o destino do comando atual.
- Ele executa o método na instância raiz da agregação, obtendo os dados necessários do comando.

- Ele persiste o novo estado da agregação no respectivo banco de dados. Esta última operação é, verdadeiramente, a transação.

Normalmente, um manipulador de comandos lida com uma única agregação orientada por sua raiz de agregação (entidade de raiz). Se várias agregações devem ser afetadas pela recepção de um único comando, você pode usar eventos de domínio para propagar estados ou ações entre várias agregações.

O ponto importante aqui é que, quando um comando está sendo processado, toda a lógica do domínio deve estar dentro do modelo de domínio (as agregações), totalmente encapsulada e pronta para o teste de unidade. O manipulador de comandos atua apenas como uma maneira de obter o modelo de domínio do banco de dados e como a etapa final, para informar à camada de infraestrutura (aos repositórios) para persistir as alterações quando o modelo for alterado. A vantagem dessa abordagem é que você pode refatorar a lógica do domínio em um modelo de domínio isolado, totalmente encapsulado, avançado e comportamental sem alterar o código do aplicativo ou as camadas de infraestrutura, que fazem parte do nível de detalhes técnicos (manipuladores de comandos, API Web, repositórios, etc.).

Quando manipuladores de comandos se tornam complexos, com muita lógica, começam a ficar parecidos com código. Examine-os e, se você encontrar lógica de domínio, refatore o código para mover esse comportamento de domínio para os métodos dos objetos de domínio (a raiz de agregação e a entidade filho).

Como exemplo de uma classe de manipulador de comando, o código a seguir mostra a mesma `CreateOrderCommandHandler` classe que você viu no início deste capítulo. Nesse caso, ele também realça o método `Handle` e as operações com os objetos/agregados de modelo de domínio.

```

public class CreateOrderCommandHandler
    : IRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;
    private readonly IOrderingIntegrationEventService _orderingIntegrationEventService;
    private readonly ILogger<CreateOrderCommandHandler> _logger;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService identityService,
        ILogger<CreateOrderCommandHandler> logger)
    {
        _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
        _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new
        ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<bool> Handle(CreateOrderCommand message, CancellationToken cancellationToken)
    {
        // Add Integration event to clean the basket
        var orderStartedIntegrationEvent = new OrderStartedIntegrationEvent(message.UserId);
        await _orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegrationEvent);

        // Add/Update the Buyer AggregateRoot
        // DDD patterns comment: Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate
        var address = new Address(message.Street, message.City, message.State, message.Country,
            message.ZipCode);
        var order = new Order(message.UserId, message.UserName, address, message.CardTypeId,
            message.CardNumber, message.CardSecurityNumber, message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice, item.Discount,
                item.PictureUrl, item.Units);
        }

        _logger.LogInformation("----- Creating Order - Order: {@Order}", order);

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);
    }
}

```

Aqui estão etapas adicionais que um manipulador de comandos deve realizar:

- Use os dados do comando para operar com os métodos e o comportamento da raiz agregada.
- Internamente, dentro dos objetos de domínio, acionar eventos de domínio enquanto a transação é executada, mas isso é transparente do ponto de vista de um manipulador comandos.
- Se o resultado da operação da agregação for bem-sucedido e depois que a transação for concluída, aumente os eventos de integração. (Eles também podem ser acionados por classes de infraestrutura como repositórios).

## Recursos adicionais

- Marque Seemann. Nos limites, os aplicativos não são orientados a objeto  
<https://blog.ploeh.dk/2011/05/31/AttheBoundaries,ApplicationsareNotObject-Oriented/>
- Comandos e eventos  
<https://cqrs.nu/Faq/commands-and-events>
- O que faz um manipulador de comandos?  
<https://cqrs.nu/Faq/command-handlers>
- Jimmy Bogard. Padrões de comando de domínio – manipuladores  
<https://jimmybogard.com/domain-command-patterns-handlers/>
- Jimmy Bogard. Padrões de comando de domínio – validação  
<https://jimmybogard.com/domain-command-patterns-validation/>

## O pipeline de processo Comando: como disparar um manipulador de comandos

A próxima pergunta é como invocar um manipulador de comandos. Você poderia chamá-lo manualmente em cada controlador do ASP.NET Core relacionado. No entanto, essa abordagem seria muito acoplada e não seria o ideal.

As outras duas opções principais, que são as opções recomendadas, são:

- Por meio de um artefato de padrão Mediador na memória.
- Com uma fila de mensagens assíncronas, entre os controladores e manipuladores.

### Usar o padrão Mediador (na memória) no pipeline de comando

Conforme mostrado na Figura 7-25, em uma abordagem CQRS você usa um mediador, semelhante a um barramento na memória, que é inteligente o suficiente para redirecionar para o manipulador de comandos correto com base no tipo de comando ou DTO que está sendo recebido. As setas pretas simples entre componentes representam as dependências entre objetos (em muitos casos, injetadas por meio da DI) com as respectivas interações.

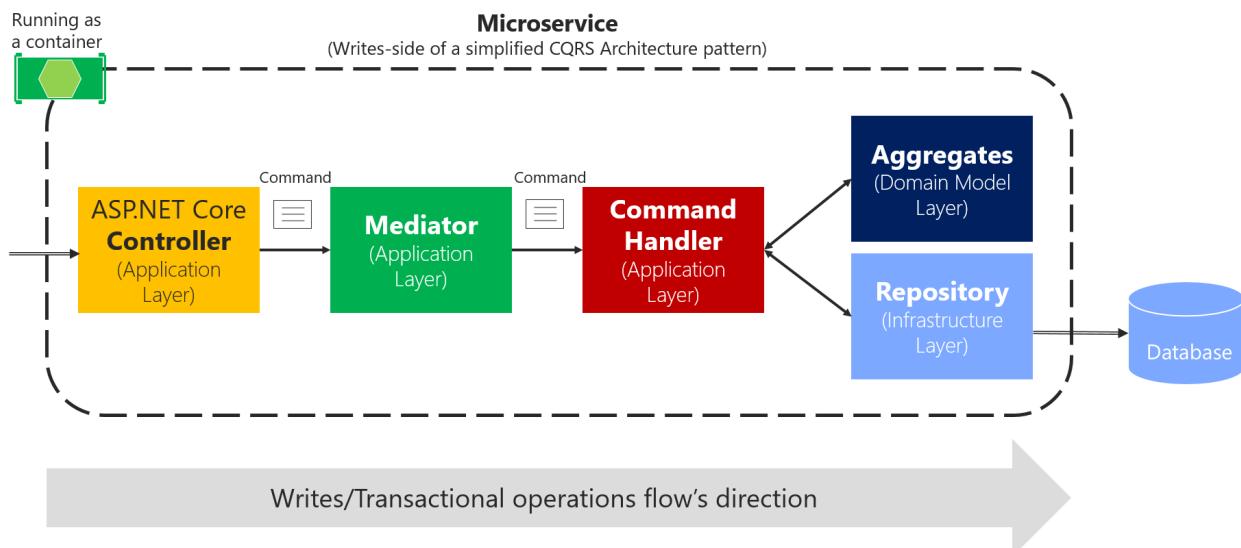


Figura 7-25. Usando o padrão Mediador no processo em um único microserviço CQRS

O diagrama acima mostra um zoom da imagem 7-24: o controlador de ASP.NET Core envia o comando para o pipeline de comando do mediador, para que eles obtenham o manipulador apropriado.

O motivo pelo qual o uso do padrão Mediador faz sentido é porque, em aplicativos empresariais, as solicitações de processamento podem ficar complicadas. Você almeja adicionar um número indefinido de interesses transversais como registro em log, validações, auditoria e segurança. Nesses casos, você pode confiar em um pipeline mediador (veja [Padrão mediador](#)) para oferecer um meio para esses comportamentos adicionais ou interesses transversais.

Um mediador é um objeto que encapsula o "como" desse processo: ele coordena a execução com base no estado, a maneira como um manipulador de comando é invocado ou a carga que você fornece ao manipulador. Com um componente do mediador, você pode aplicar preocupações abrangentes de forma centralizada e transparente aplicando decoradores (ou [comportamentos de pipeline](#) desde o [mediador 3](#)). Para obter mais informações, consulte o [Padrão decorador](#).

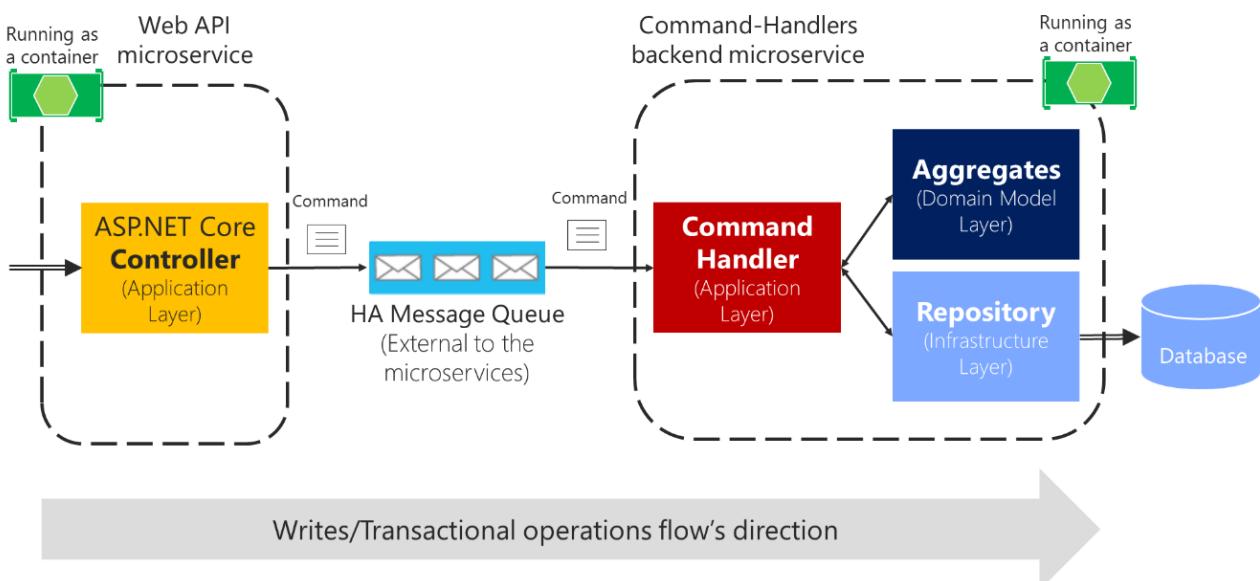
Os decoradores e comportamentos são semelhantes à [AOP \(Programação orientada a aspectos\)](#), aplicada somente a um pipeline de processo específico gerenciado pelo componente mediador. Os aspectos na AOP, que implementam interesses transversais, são aplicados com base em *construtores de aspecto* injetados em tempo de compilação ou com base na interceptação da chamada de objeto. Às vezes, essas duas abordagens de AOP típicas parecem funcionar "como mágica", porque não é fácil entender como a AOP faz seu trabalho. Ao lidar com problemas sérios ou bugs, pode ser difícil depurar a AOP. Por outro lado, esses decoradores/comportamentos são explícitos e aplicados apenas no contexto do mediador, assim, a depuração fica muito mais fácil e previsível.

Por exemplo, no microserviço de ordenação eShopOnContainers, o tem uma implementação de dois comportamentos de exemplo, uma classe [LogBehavior](#) e uma classe [ValidatorBehavior](#). A implementação dos comportamentos é explicada na próxima seção mostrando como o eShopOnContainers usa os [comportamentos do mediador](#).

#### **Usar filas de mensagens (fora do processo) no pipeline do comando**

Outra opção é usar mensagens assíncronas com base em agentes ou filas de mensagens, conforme mostrado na Figura 7-26. Essa opção também pode ser combinada com o componente mediador imediatamente antes do manipulador de comandos.

#### **Writes-side of a CQRS Architecture pattern using messaging**



**Figura 7-26.** Usando filas de mensagens (fora do processo e comunicação entre processos) com comandos CQRS

O pipeline do comando também pode ser tratado por uma fila de mensagens de alta disponibilidade para entregar os comandos ao manipulador adequado. O uso de filas de mensagens para aceitar os comandos pode complicar ainda mais o pipeline do comando, pois você provavelmente precisará dividir o pipeline em dois processos conectados por meio da fila de mensagens externa. Ainda assim, isso deve ser usado se você precisa ter

melhor escalabilidade e desempenho com base no sistema de mensagens assíncrono. Considere que, no caso da Figura 7-26, o controlador apenas posta a mensagem de comando na fila e retorna. Em seguida, o manipulador de comandos processa as mensagens em seu próprio ritmo. Esse é um grande benefício das filas: a fila de mensagens pode agir como um buffer nos casos em que a hiperescalabilidade é necessária, como para estoques ou qualquer outro cenário com um alto volume de dados de entrada.

No entanto, devido à natureza assíncrona das filas de mensagens, você precisa descobrir como se comunicar com o aplicativo cliente sobre o êxito ou a falha do processo do comando. Como regra, você nunca deve usar comandos "Fire and esqueça". Todos os aplicativos de negócios precisam saber se um comando foi processado com êxito ou, pelo menos, validado e aceito.

Portanto, ser capaz de responder ao cliente depois de validar uma mensagem de comando que foi enviada para uma fila assíncrona adiciona complexidade ao seu sistema, em comparação com um processo de comando em processo que retorna o resultado da operação após a execução da transação. Ao usar filas, talvez seja necessário retornar o resultado do processo do comando por meio de outras mensagens de resultado da operação, o que exigirá comunicação personalizada e componentes adicionais em seu sistema.

Além disso, os comandos assíncronos são unidirecionais, o que, em muitos casos, pode não ser necessário, conforme explicado na seguinte discussão interessante entre Burtsev Alexey e Greg Young em uma [conversa online](#):

[Burtsev Alexey] eu encontrei muitos códigos em que as pessoas usam o manuseio de comandos assíncronos ou mensagens de comando unidirecionais sem qualquer motivo para fazer isso (eles não estão fazendo uma operação longa, eles não estão executando código assíncrono externo, eles nem mesmo limites entre aplicativos para usar o barramento de mensagem). Por que eles introduzem essa complexidade desnecessária? E, na verdade, eu não vi nenhum exemplo de código CQRS com manipuladores de comando de bloqueio até o momento, embora isso funcionaria muito bem na maioria dos casos.

[Greg Young] [...] um comando assíncrono não existe; na verdade, ele é outro evento. Se eu precisar aceitar o que você me envia e gerar um evento se eu discordar, não será mais que você me informasse a fazer algo [, não é um comando]. É você me informando que algo foi feito. Parece que essa é apenas uma pequena diferença inicialmente, mas isso tem várias implicações.

Os comandos assíncronos aumentam significativamente a complexidade de um sistema, porque não há nenhuma maneira simples de indicar falhas. Portanto, os comandos assíncronos não são recomendados a não ser quando há requisitos de dimensionamento ou em casos especiais, ao comunicar os microsserviços internos por meio do sistema de mensagens. Nesses casos, você deve projetar um sistema de relatórios e de recuperação separado para falhas.

Na versão inicial do eShopOnContainers, foi decidido usar o processamento de comando síncrono, iniciado a partir de solicitações HTTP e controladas pelo padrão mediador. Isso permite retornar o êxito ou a falha do processo com facilidade, como na implementação de [CreateOrderCommandHandler](#).

Em qualquer caso, isso deve ser uma decisão com base nos requisitos de negócios do seu aplicativo ou do microsserviço.

## Implementar o pipeline de processo de comando com um padrão mediador (MediatR)

Como uma implementação de exemplo, este guia propõe o uso do pipeline em processo baseado no padrão Mediador para orientar a ingestão de comando e rotear comandos, na memória, para os manipuladores de comando corretos. O guia também propõe a aplicação de [comportamentos](#) para separar interesses transversais.

Para a implementação no .NET Core, há várias bibliotecas de software livre disponíveis que implementam o padrão Mediador. A biblioteca usada neste guia é a [MediatR](#), biblioteca de software livre criada por Jimmy Bogard,

mas você pode usar outra abordagem. A MediatR é uma biblioteca pequena e simples que permite que você processe mensagens na memória como um comando, aplicando, ao mesmo tempo, decoradores ou comportamentos.

O uso do padrão Mediador ajuda a reduzir o acoplamento e isolar as preocupações com o trabalho solicitado, ao conectar-se automaticamente com manipulador que executa esse trabalho — nesse caso, os manipuladores de comando.

Outra boa razão para usar o padrão Mediador foi explicada por Jimmy Bogard durante a revisão desse guia:

Acho que vale a pena mencionar testes aqui – eles oferecem uma janela consistente e adequada sobre o comportamento do seu sistema. Solicitação-entrada, resposta. Descobrimos que o aspecto é bastante valioso na criação de testes com consistência.

Primeiro, vamos dar uma olhada em um controlador WebAPI de exemplo onde você realmente usaria o objeto mediador. Se você não estiver usando o objeto mediador, precisará injetar todas as dependências para esse controlador, coisas como um objeto logger e outros. Portanto, o Construtor seria complicado. Por outro lado, se você usasse o objeto mediador, o construtor do controlador poderia ser muito mais simples, com apenas algumas dependências em vez de muitas dependências, se você tivesse um por operação transversal, como no exemplo a seguir:

```
public class MyMicroserviceController : Controller
{
    public MyMicroserviceController(IMediator mediator,
                                    IMyMicroserviceQueries microserviceQueries)
    {
        // ...
    }
}
```

Veja que o mediador fornece um construtor de controlador da API Web simples e eficiente. Além disso, dentro dos métodos do controlador, o código para enviar um comando para o objeto mediador tem quase uma linha:

```
[Route("new")]
[HttpPost]
public async Task<IActionResult> ExecuteBusinessOperation([FromBody]RunOpCommand
                                                       runOperationCommand)
{
    var commandResult = await _mediator.SendAsync(runOperationCommand);

    return commandResult ? (IActionResult)Ok() : (IActionResult)BadRequest();
}
```

### Implementar comandos idempotentes

Em [eShopOnContainers](#), um exemplo mais avançado que o que foi visto está enviando um objeto `CreateOrderCommand` do microserviço de pedidos. Mas como o processo comercial de ordenação é um pouco mais complexo e, em nosso caso, ele realmente começa no microserviço basket, essa ação de enviar o objeto `CreateOrderCommand` é executada de um manipulador de eventos de integração chamado [`UserCheckoutAcceptedIntegrationEventHandler`](#) em vez de um controlador WebAPI simples chamado do aplicativo cliente como no exemplo mais simples anterior.

Mesmo assim, a ação de enviar o Comando para o MediatR é bem semelhante, conforme mostrado no código a seguir.

```

var createOrderCommand = new CreateOrderCommand(eventMsg.Basket.Items,
                                                eventMsg.UserId, eventMsg.City,
                                                eventMsg.Street, eventMsg.State,
                                                eventMsg.Country, eventMsg.ZipCode,
                                                eventMsg.CardNumber,
                                                eventMsg.CardHolderName,
                                                eventMsg.CardExpiration,
                                                eventMsg.CardSecurityNumber,
                                                eventMsg.CardTypeId);

var requestCreateOrder = new IdentifiedCommand<CreateOrderCommand, bool>(createOrderCommand,
                                                                           eventMsg.RequestId);

result = await _mediator.Send(requestCreateOrder);

```

No entanto, esse caso também é um pouco mais avançado, pois também estamos implementando comandos idempotentes. O processo CreateOrderCommand deve ser idempotente, portanto, se a mesma mensagem vier duplicada pela rede, independentemente do motivo, como repetições, a mesma ordem de negócios será processada apenas uma vez.

Isso é implementado encapsulando o comando de negócios (neste caso, CreateOrderCommand) e inserindo-o em um IdentifiedCommand genérico, que é acompanhado por uma ID de cada mensagem proveniente da rede que precisa ser idempotente.

Veja no código abaixo que o IdentifiedCommand não passa de um DTO com uma ID, além do objeto do comando de negócios encapsulado.

```

public class IdentifiedCommand<T, R> : IRequest<R>
{
    where T : IRequest<R>

    public T Command { get; }
    public Guid Id { get; }
    public IdentifiedCommand(T command, Guid id)
    {
        Command = command;
        Id = id;
    }
}

```

Então, o CommandHandler do IdentifiedCommand chamado [IdentifiedCommandHandler.cs](#) vai basicamente verificar se a ID que está chegando como parte da mensagem já existe em uma tabela. Se ele já existir, esse comando não será processado novamente, portanto, ele se comporta como um comando idempotente. Esse código de infraestrutura é executado pela chamada de método `_requestManager.ExistAsync` abaixo.

```

// IdentifiedCommandHandler.cs
public class IdentifiedCommandHandler<T, R> : IRequestHandler<IdentifiedCommand<T, R>, R>
{
    where T : IRequest<R>

    private readonly IMediator _mediator;
    private readonly IRequestManager _requestManager;
    private readonly ILogger<IdentifiedCommandHandler<T, R>> _logger;

    public IdentifiedCommandHandler(
        IMediator mediator,
        IRequestManager requestManager,
        ILogger<IdentifiedCommandHandler<T, R>> logger)
    {
        _mediator = mediator;
        _requestManager = requestManager;
        _logger = logger ?? throw new System.ArgumentNullException(nameof(logger));
    }
}

```

```

/// <summary>
/// Creates the result value to return if a previous request was found
/// </summary>
/// <returns></returns>
protected virtual R CreateResultForDuplicateRequest()
{
    return default(R);
}

/// <summary>
/// This method handles the command. It just ensures that no other request exists with the same ID, and if
this is the case
/// just enqueues the original inner command.
/// </summary>
/// <param name="message">IdentifiedCommand which contains both original command & request ID</param>
/// <returns>Return value of inner command or default value if request same ID was found</returns>
public async Task<R> Handle(IdentifiedCommand<T, R> message, CancellationToken cancellationToken)
{
    var alreadyExists = await _requestManager.ExistAsync(message.Id);
    if (alreadyExists)
    {
        return CreateResultForDuplicateRequest();
    }
    else
    {
        await _requestManager.CreateRequestForCommandAsync<T>(message.Id);
        try
        {
            var command = message.Command;
            var commandName = command.GetGenericTypeName();
            var idProperty = string.Empty;
            var commandId = string.Empty;

            switch (command)
            {
                case CreateOrderCommand createOrderCommand:
                    idProperty = nameof(createOrderCommand.UserId);
                    commandId = createOrderCommand.UserId;
                    break;

                case CancelOrderCommand cancelOrderCommand:
                    idProperty = nameof(cancelOrderCommand.OrderNumber);
                    commandId = $"{cancelOrderCommand.OrderNumber}";
                    break;

                case ShipOrderCommand shipOrderCommand:
                    idProperty = nameof(shipOrderCommand.OrderNumber);
                    commandId = $"{shipOrderCommand.OrderNumber}";
                    break;

                default:
                    idProperty = "Id?";
                    commandId = "n/a";
                    break;
            }

            _logger.LogInformation(
                "----- Sending command: {CommandName} - {IdProperty}: {CommandId} ({@Command})",
                commandName,
                idProperty,
                commandId,
                command);
        }
        // Send the embeded business command to mediator so it runs its related CommandHandler
        var result = await _mediator.Send(command, cancellationToken);

        _logger.LogInformation(
            "----- Command result: {@Result} - {CommandName} - {IdProperty}: {CommandId}"
            {@Command}),
    }
}

```

```
        result,
        commandName,
        idProperty,
        commandId,
        command);

    return result;
}
catch
{
    return default(R);
}
}
}
}
```

Como o IdentifiedCommand atua como um envelope de comando de negócios, quando o comando comercial precisa ser processado porque não é uma ID repetida, ele pega o comando comercial interno e o reenvia para mediador, como na última parte do código mostrado acima durante `_mediator.Send(message.Command)` a execução, do [IdentifiedCommandHandler.cs](#).

Ao fazer isso, ele vinculará e executará o manipulador de comandos de negócios, nesse caso, o [CreateOrderCommandHandler](#), que está executando transações em relação ao banco de dados de ordenação, conforme mostrado no código a seguir.

```

// CreateOrderCommandHandler.cs
public class CreateOrderCommandHandler
    : IRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;
    private readonly IIdentityService _identityService;
    private readonly IMediator _mediator;
    private readonly IOrderingIntegrationEventService _orderingIntegrationEventService;
    private readonly ILogger<CreateOrderCommandHandler> _logger;

    // Using DI to inject infrastructure persistence Repositories
    public CreateOrderCommandHandler(IMediator mediator,
        IOrderingIntegrationEventService orderingIntegrationEventService,
        IOrderRepository orderRepository,
        IIdentityService identityService,
        ILogger<CreateOrderCommandHandler> logger)
    {
        _orderRepository = orderRepository ?? throw new ArgumentNullException(nameof(orderRepository));
        _identityService = identityService ?? throw new ArgumentNullException(nameof(identityService));
        _mediator = mediator ?? throw new ArgumentNullException(nameof(mediator));
        _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new
        ArgumentNullException(nameof(orderingIntegrationEventService));
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task<bool> Handle(CreateOrderCommand message, CancellationToken cancellationToken)
    {
        // Add Integration event to clean the basket
        var orderStartedIntegrationEvent = new OrderStartedIntegrationEvent(message.UserId);
        await _orderingIntegrationEventService.AddAndSaveEventAsync(orderStartedIntegrationEvent);

        // Add/Update the Buyer AggregateRoot
        // DDD patterns comment: Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate
        var address = new Address(message.Street, message.City, message.State, message.Country,
            message.ZipCode);
        var order = new Order(message.UserId, message.UserName, address, message.CardTypeId,
            message.CardNumber, message.CardSecurityNumber, message.CardHolderName, message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice, item.Discount,
                item.PictureUrl, item.Units);
        }

        _logger.LogInformation("----- Creating Order - Order: {@Order}", order);

        _orderRepository.Add(order);

        return await _orderRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);
    }
}

```

## Registrar os tipos usados pelo MediatR

Para que o MediatR tome ciência de suas classes de manipulador de comando, é necessário registrar as classes de mediador e as classes de manipulador de comandos em seu contêiner de IoC. Por padrão, o MediatR usa o Autofac como contêiner de IoC, mas você também pode usar o contêiner de IoC interno do ASP.NET Core ou qualquer outro contêiner compatível com o MediatR.

O código a seguir mostra como registrar tipos e comandos do mediador ao usar módulos Autofac.

```

public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(typeof(CreateOrderCommand).GetTypeInfo().Assembly)
            .AsClosedTypesOf(typeof(IRequestHandler<,>));
        // Other types registration
        //...
    }
}

```

É aí que "a mágica acontece" com o mediador.

Como cada manipulador de comandos implementa a `IRequestHandler<T>` interface genérica, quando você registra os assemblies usando o `RegisteredAssemblyTypes` método, todos os tipos marcados como `IRequestHandler` também são registrados com seus `Commands`. Por exemplo:

```

public class CreateOrderCommandHandler
    : IRequestHandler<CreateOrderCommand, bool>
{

```

Esse é o código que correlaciona comandos com manipuladores de comandos. O manipulador é apenas uma classe simples, mas ele herda de `RequestHandler<T>`, em que T é o tipo de comando, enquanto o MediatR garante que ele seja invocado com o conteúdo correto.

## Aplicar questões abrangentes ao processar comandos com os Comportamentos no MediatR

Há mais um assunto para discutir: a capacidade de aplicar interesses transversais ao pipeline mediador. Você também pode ver, no final do código do módulo de registro do Autofac, como ele registra um tipo de comportamento, especificamente uma classe LoggingBehavior personalizada e uma classe ValidatorBehavior. Mas você também pode adicionar outros comportamentos personalizados.

```

public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        // Register all the Command classes (they implement IRequestHandler)
        // in assembly holding the Commands
        builder.RegisterAssemblyTypes(
            typeof(CreateOrderCommand).GetTypeInfo().Assembly).
            AsClosedTypesOf(typeof(IRequestHandler<,>));

        // Other types registration
        //...
        builder.RegisterGeneric(typeof(LoggingBehavior<,>)).
            As(typeof(IPipelineBehavior<,>));
        builder.RegisterGeneric(typeof(ValidatorBehavior<,>)).
            As(typeof(IPipelineBehavior<,>));
    }
}

```

Essa classe [LoggingBehavior](#), que registra informações sobre o manipulador de comando que está sendo executado e se ele foi bem-sucedido ou não, pode ser implementada como no código a seguir.

```
public class LoggingBehavior<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
{
    private readonly ILogger<LoggingBehavior<TRequest, TResponse>> _logger;
    public LoggingBehavior(ILogger<LoggingBehavior<TRequest, TResponse>> logger) =>
        _logger = logger;

    public async Task<TResponse> Handle(TRequest request,
                                         RequestHandlerDelegate<TResponse> next)
    {
        _logger.LogInformation($"Handling {typeof(TRequest).Name}");
        var response = await next();
        _logger.LogInformation($"Handled {typeof(TResponse).Name}");
        return response;
    }
}
```

Basta implementar essa classe de comportamento e registrar o pipeline com ela (no MediatorModule acima) e todos os comandos processados por meio do MediatR registrarão em log as informações sobre a execução.

O microserviço de pedidos eShopOnContainers também aplica um segundo comportamento para validações básicas, a classe [ValidatorBehavior](#), que depende da biblioteca [FluentValidation](#), conforme mostrado no código a seguir:

```
public class ValidatorBehavior<TRequest, TResponse>
    : IPipelineBehavior<TRequest, TResponse>
{
    private readonly IValidator<TRequest>[] _validators;
    public ValidatorBehavior(IValidator<TRequest>[] validators) =>
        _validators = validators;

    public async Task<TResponse> Handle(TRequest request,
                                         RequestHandlerDelegate<TResponse> next)
    {
        var failures = _validators
            .Select(v => v.Validate(request))
            .SelectMany(result => result.Errors)
            .Where(error => error != null)
            .ToList();

        if (failures.Any())
        {
            throw new OrderingDomainException(
                $"Command Validation Errors for type {typeof(TRequest).Name}",
                new ValidationException("Validation exception", failures));
        }

        var response = await next();
        return response;
    }
}
```

Aqui, o comportamento é gerar uma exceção se a validação falhar, mas você também poderá retornar um objeto de resultado, que contém o resultado do comando se ele tiver êxito ou as mensagens de validação, caso não tenha sido. Isso provavelmente tornaria mais fácil exibir os resultados da validação para o usuário.

Em seguida, com base na biblioteca [FluentValidation](#), você criaria a validação para os dados passados com CreateOrderCommand, como no código a seguir:

```

public class CreateOrderCommandValidator : AbstractValidator<CreateOrderCommand>
{
    public CreateOrderCommandValidator()
    {
        RuleFor(command => command.City).NotEmpty();
        RuleFor(command => command.Street).NotEmpty();
        RuleFor(command => command.State).NotEmpty();
        RuleFor(command => command.Country).NotEmpty();
        RuleFor(command => command.ZipCode).NotEmpty();
        RuleFor(command => command.CardNumber).NotEmpty().Length(12, 19);
        RuleFor(command => command.CardHolderName).NotEmpty();
        RuleFor(command => command.CardExpiration).NotEmpty().Must(BeValidExpirationDate).WithMessage("Please
specify a valid card expiration date");
        RuleFor(command => command.CardSecurityNumber).NotEmpty().Length(3);
        RuleFor(command => command.CardTypeId).NotEmpty();
        RuleFor(command => command.OrderItems).Must(ContainOrderItems).WithMessage("No order items found");
    }

    private bool BeValidExpirationDate(DateTime dateTime)
    {
        return dateTime >= DateTime.UtcNow;
    }

    private bool ContainOrderItems(IEnumerable<OrderItemDTO> orderItems)
    {
        return orderItems.Any();
    }
}

```

Você pode criar validações adicionais. Essa é uma maneira muito eficiente e elegante de se implementar validações de comando.

De maneira semelhante, você pode implementar outros comportamentos para aspectos adicionais ou interesses transversais que você deseja aplicar aos comandos ao manipulá-los.

#### **Recursos adicionais**

O padrão mediador

- Padrão mediador

[https://en.wikipedia.org/wiki/Mediator\\_pattern](https://en.wikipedia.org/wiki/Mediator_pattern)

O padrão decorador

- Padrão de decorador

[https://en.wikipedia.org/wiki/Decorator\\_pattern](https://en.wikipedia.org/wiki/Decorator_pattern)

MediatR (Jimmy Bogard)

- MediatR. Repositório do GitHub.

<https://github.com/jbogard/MediatR>

- CQRS com mediador e AutoMapper

<https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/>

- Coloque os controladores de dieta: POSTs e comandos.

<https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/>

- Lidando com preocupações abrangentes com um pipeline mediador

<https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/>

- CQRS e REST: a combinação perfeita

<https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/>

- Exemplos de pipeline do mediador

<https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/>

- Acessórios de teste de fatia vertical para mediador e ASP.NET Core

<https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/>

- Extensões do mediador para injeção de dependência da Microsoft liberadas

<https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/>

Validação fluente

- Jeremy Skinner. FluentValidation. Repositório do GitHub.

<https://github.com/JeremySkinner/FluentValidation>

[ANTERIOR](#)

[AVANÇAR](#)

# Implementar aplicações resilientes

18/03/2020 • 3 minutes to read • [Edit Online](#)

*Seus aplicativos de microserviço e baseados em nuvem devem abraçar as falhas parciais que certamente ocorrerão eventualmente. Você deve projetar sua aplicação para ser resiliente a essas falhas parciais.*

A resiliência é a capacidade de se recuperar de falhas e continuar funcionando. Não se trata de evitar falhas, mas de aceitar o fato de que as falhas acontecerão, e responder a elas de uma maneira que evite tempo de inatividade ou perda de dados. A meta de resiliência é retornar o aplicativo para um estado totalmente funcional após uma falha.

Já é um grande desafio criar e implantar um aplicativo baseado em microsserviços. E você ainda precisa manter o aplicativo em execução em um ambiente em que algum tipo de falha certamente ocorrerá. Portanto, seu aplicativo precisa ser resiliente. Ele deve ser projetado para lidar com falhas parciais, como interrupções da rede ou falhas de nós ou de VMs na nuvem. Até mesmo os microsserviços (contêineres) que estão sendo movidos para outro nó em um cluster podem causar falhas curtas intermitentes no aplicativo.

Os diversos componentes individuais do aplicativo também precisam incorporar recursos de monitoramento de integridade. Seguindo as diretrizes neste capítulo, você poderá criar um aplicativo que pode funcionar perfeitamente apesar do tempo de inatividade temporário ou das interrupções normais que ocorrem em implantações complexas e baseadas em nuvem.

## IMPORTANT

O eShopOnContainer estava usando a [biblioteca Polly](#) para implementar resiliência usando [clientes digitados](#) até a versão 3.0.0.

A partir da versão 3.0.0, o HTTP chama a resiliência usando uma [malha Linkerd](#), que lida com repetições de forma transparente e configurável, dentro de um cluster Kubernetes, sem ter que lidar com essas preocupações no código.

A biblioteca Polly ainda é usada para adicionar resiliência às conexões de banco de dados, especialmente durante a inicialização dos serviços.

## WARNING

Todas as amostras de código nesta seção eram válidas antes de usar o Linkerd e não são atualizadas para refletir o código real atual. Então eles fazem sentido no contexto desta seção.

[PRÓXIMO](#)

[ANTERIOR](#)

# Tratar falhas parciais

09/04/2020 • 5 minutes to read • [Edit Online](#)

Em sistemas distribuídos como aplicativos baseados em microsserviços, há um risco de falha parcial sempre presente. Por exemplo, um único microsserviço/contêiner pode falhar ou pode não estar disponível para responder por um curto período, ou um único servidor ou VM pode falhar. Como clientes e serviços são processos separados, um serviço pode não ser capaz de responder em tempo háprimo à solicitação de um cliente. O serviço pode estar sobrecarregado e respondendo a solicitações de maneira muito lenta ou pode simplesmente não estar acessível durante um curto período devido a problemas de rede.

Por exemplo, considere a página de detalhes Ordem do aplicativo de exemplo eShopOnContainers. Se o microsserviço de ordenação não estiver respondendo quando o usuário tentar enviar uma ordem, uma implementação incorreta do processo do cliente (o aplicativo Web MVC) – por exemplo, se o código do cliente usasse RPCs síncronos sem tempo limite – bloqueará threads indefinidamente que estão aguardando uma resposta. Além de criar uma experiência de usuário ruim, toda espera sem resposta consome ou bloqueia um thread, e os threads são extremamente valiosos em aplicativos altamente escalonáveis. Se houver muitos threads bloqueados, eventualmente o tempo de execução do aplicativo pode ficar sem threads. Nesse caso, o aplicativo poderá ficar globalmente sem resposta, em vez de apenas parcialmente sem resposta, como mostra a Figura 8-1.

## Partial failures

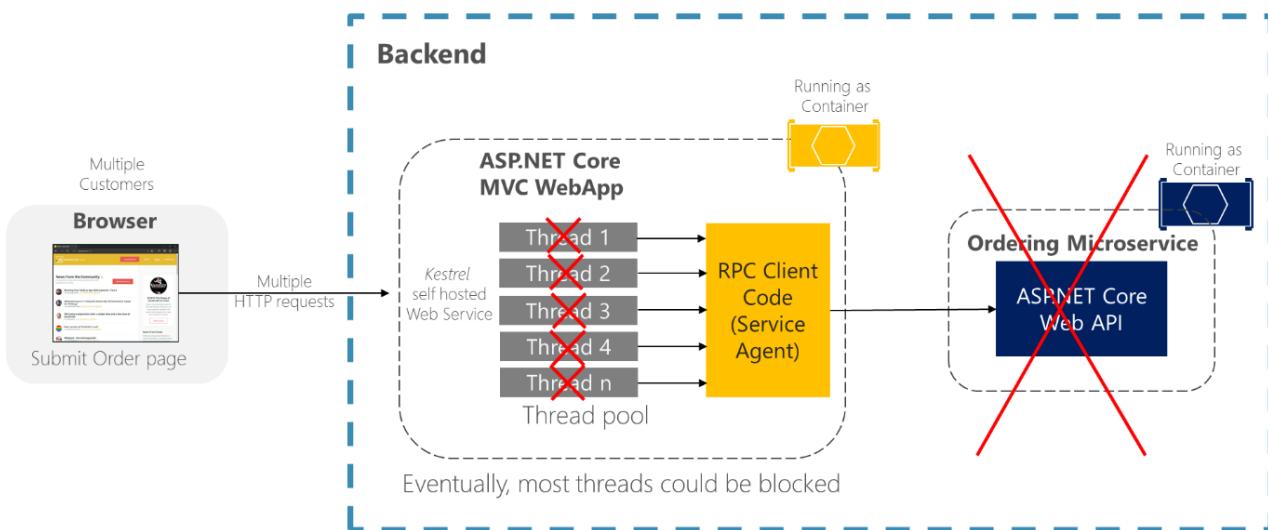


Figura 8-1. Falhas parciais devido a dependências que afetam a disponibilidade do thread de serviço

Em um aplicativo grande baseado em microsserviços, falhas parciais poderão ser amplificadas, principalmente se a maioria da interação interna dos microsserviços for baseada em chamadas HTTP síncronas (o que é considerado um antipadrão). Pense em um sistema que recebe milhões de chamadas de entrada por dia. Se o seu sistema tiver um design ruim baseado em longas cadeias de chamadas HTTP síncronas, essas chamadas recebidas podem resultar em muito mais milhões de chamadas de saída (vamos supor uma proporção de 1:4) para dezenas de microsserviços internos como dependências síncronas. Essa situação é mostrada na Figura #8-2, especialmente a dependência 3, que inicia uma cadeia, chamando dependência #4. que os chamados #5.

# Multiple distributed dependencies

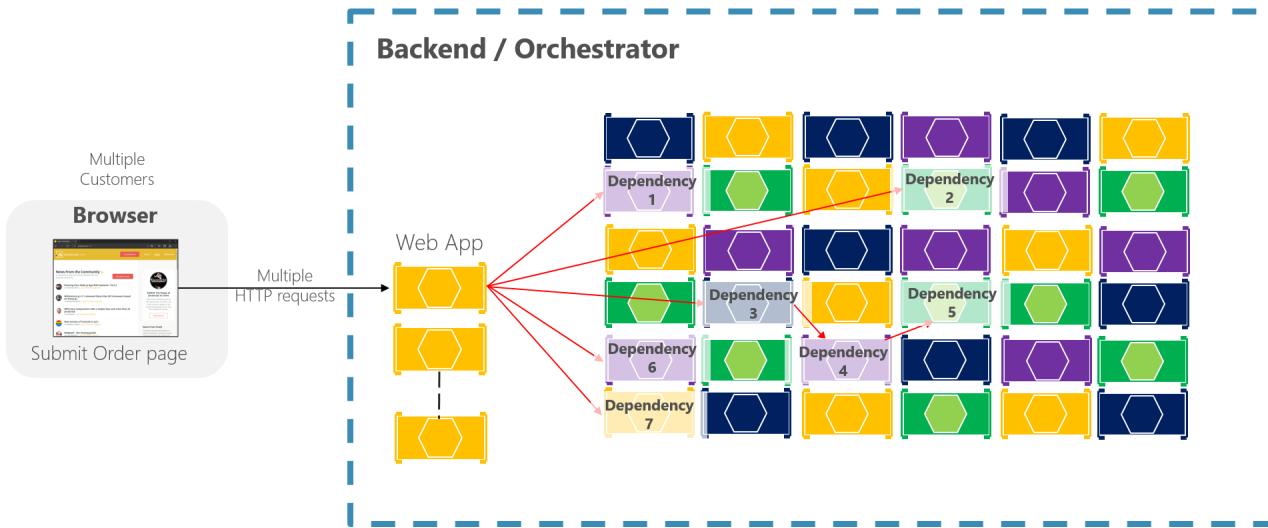


Figura 8-2. O impacto de ter um design incorreto com longas cadeias de solicitações HTTP

A falha intermitente é garantida em um sistema distribuído e baseado em nuvem, mesmo quando cada dependência têm uma disponibilidade excelente. Esse é um fato que você precisa considerar.

Se você não criar nem implementar técnicas para garantir a tolerância a falhas, até mesmo pequenos tempos de inatividade poderão ser amplificados. Por exemplo, 50 dependências, cada uma com 99,99% de disponibilidade, poderia resultar em várias horas de tempo de inatividade por mês devido a esse efeito de ondulação. Quando uma dependência de microserviço falhar ao manipular um alto volume de solicitações, essa falha poderá saturar rapidamente todos os threads de solicitação disponíveis em cada serviço e causar uma pane em todo o aplicativo.

## Partial Failure Amplified in Microservices

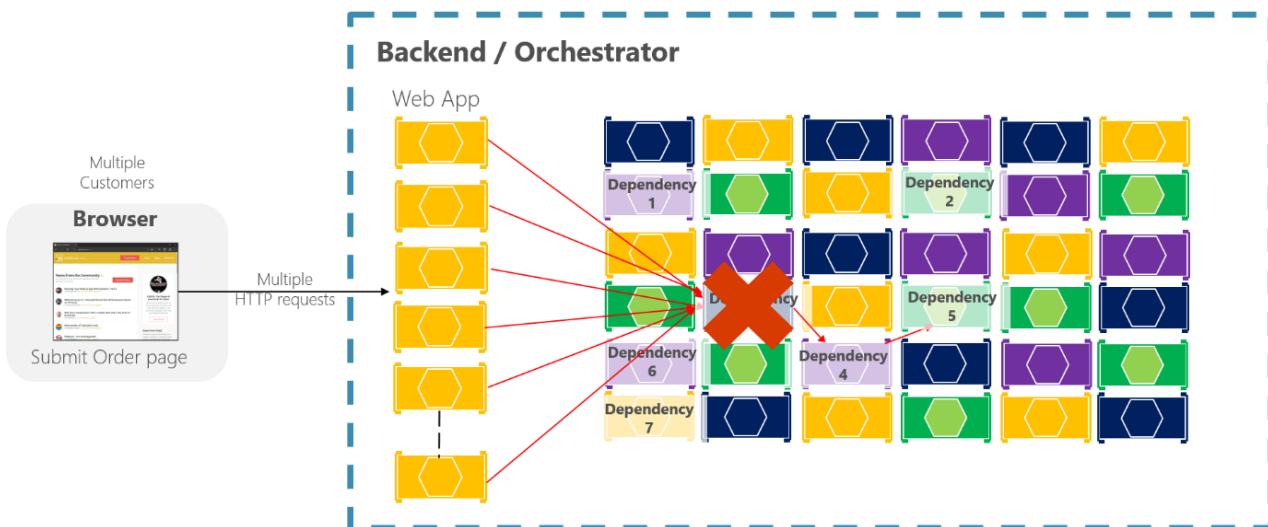


Figura 8-3. Falha parcial amplificada por microserviços com longas cadeias de chamadas HTTP síncronas

Para minimizar esse problema, na seção Integração de [microserviços Assíncronos impõe a autonomia do microserviço](#), este guia incentiva você a usar a comunicação assíncrona em todos os microserviços internos.

Além disso, é essencial que você crie seus aplicativos cliente e microserviços para lidar com falhas parciais, ou seja, crie microserviços e aplicativos cliente resilientes.



# Estratégias para tratar falhas parciais

09/04/2020 • 5 minutes to read • [Edit Online](#)

As estratégias para lidar com falhas parciais incluem o seguinte.

**Usar a comunicação assíncrona (por exemplo, comunicação baseada em mensagens) entre microsserviços internos.** É altamente recomendado não criar cadeias longas de chamadas HTTP síncronas entre os microsserviços internos, porque esse design incorreto poderá se tornar a principal causa de interrupções incorretas. Pelo contrário, exceto pelas comunicações de front-end entre os aplicativos cliente e o primeiro nível de microsserviços ou Gateways de API refinados, é recomendado usar apenas a comunicação assíncrona (com base em mensagem) uma vez após o ciclo inicial de resposta/solicitação, entre os microsserviços internos. Consistência eventual e arquiteturas orientadas a eventos ajudarão a minimizar os efeitos de ondulação. Essas abordagens impõem um nível mais alto de autonomia do microsserviço e, portanto, previnem contra o problema observado aqui.

**Usar novas tentativas com retirada exponencial.** Essa técnica ajuda a evitar falhas curtas e intermitentes executando novas tentativas de chamada um determinado número de vezes, caso o serviço não tivesse estado disponível apenas por um curto período de tempo. Isso pode ocorrer devido a problemas de rede intermitentes ou quando um microsserviço/contêiner é movido para um nó diferente em um cluster. No entanto, se essas novas tentativas não foram criadas corretamente com disjuntores, os efeitos de ondulação poderão ser agravados e, em último caso, poderá até haver uma [DoS \(Negação de Serviço\)](#).

**Solução alternativa para tempos limite de rede.** Em geral, os clientes devem ser criados para não serem bloqueados indefinidamente e para sempre usar tempos limite ao aguardar uma resposta. Usar tempos limite garante que os recursos nunca fiquem bloqueados indefinidamente.

**Usar o padrão de disjuntor.** Nessa abordagem, o processo do cliente rastreia o número de solicitações com falha. Se a taxa de erro exceder um limite configurado, um "disjuntor" será executado para que novas tentativas falhem imediatamente. (Se um grande número de solicitações estiver falhando, isso sugere que o serviço não está disponível e que o envio de solicitações é inútil.) Após um período de tempo, o cliente deve tentar novamente e, se as novas solicitações forem bem sucedidas, feche o disjuntor.

**Fornecer fallbacks.** Nessa abordagem, o processo de cliente executa a lógica de fallback quando uma solicitação falha, como retornar dados armazenados em cache ou um valor padrão. Essa é uma abordagem adequada para consultas e é mais complexa para atualizações ou comandos.

**Limitar o número de solicitações na fila.** Os clientes também devem impor um limite superior no número de solicitações pendentes que um microsserviço cliente pode enviar para um serviço específico. Se o limite for atingido, provavelmente será ineficaz fazer mais solicitações. As tentativas falharão imediatamente. Em termos de implementação, a política [Isolamento do bulkhead](#) da Polly pode ser usada para atender a esse requisito. Essa abordagem é essencialmente uma restrição de paralelização com [SemaphoreSlim](#) como a implementação. Ela também permite uma "fila" fora do bulkhead. É possível lançar proativamente uma carga excessiva mesmo antes da execução (por exemplo, devido à capacidade ser considerada cheia). Isso torna sua resposta a determinados cenários de falha mais rápida do que um disjuntor seria, uma vez que o disjuntor aguarda as falhas. O objeto BulkheadPolicy na [Polly](#) expõe se o bulkhead e a fila estão cheios e oferece eventos em estouro, portanto, ele também pode ser usado para permitir a escala horizontal automatizada.

## Recursos adicionais

- Padrões de resiliência

<https://docs.microsoft.com/azure/architecture/patterns/category/resiliency>

- Adicionando resiliência e otimizando desempenho  
[/previous-versions/msp-n-p/jj591574\(v=pandp.10\)](/previous-versions/msp-n-p/jj591574(v=pandp.10))
- Anteparo. Repositório do GitHub. Implementação com a política Polly.  
<https://github.com/App-vNext/Polly/wiki/Bulkhead>
- Projetando aplicativos resilientes para o Azure  
<https://docs.microsoft.com/azure/architecture/resiliency/>
- Manuseio de falhas transitórias  
<https://docs.microsoft.com/azure/architecture/best-practices/transient-faults>

[PRÓXIMO](#)

[ANTERIOR](#)

# Implementar repetição com retirada exponencial

18/03/2020 • 2 minutes to read • [Edit Online](#)

*Repetições com retirada exponencial* é uma técnica que repete uma operação, com um tempo de espera aumentando exponencialmente, até que uma contagem máxima de repetições seja atingida (a **retirada exponencial**). Essa técnica adota o fato de que recursos de nuvem podem estar temporariamente não disponíveis por mais de alguns segundos por qualquer motivo. Por exemplo, um orquestrador pode estar movendo um contêiner para outro nó em um cluster para balanceamento de carga. Durante esse tempo, algumas solicitações podem falhar. Outro exemplo poderia ser um banco de dados como o SQL Azure, em que um banco de dados pode ser movido para outro servidor para balanceamento de carga, fazendo o banco de dados ficar não disponível por alguns segundos.

Há muitas abordagens para implementar a lógica de repetições com retirada exponencial.

[PRÓXIMO](#)

[ANTERIOR](#)

# Implementar conexões SQL resilientes com o Entity Framework Core

18/03/2020 • 5 minutes to read • [Edit Online](#)

Para o BD SQL do Azure, o EF (Entity Framework) Core já fornece a lógica interna de resiliência e repetição de conexão de banco de dados. Mas você precisará habilitar a estratégia de execução do Entity Framework para cada conexão de `DbContext`, caso deseje [conexões resilientes do EF Core](#).

Por exemplo, o código a seguir no nível de conexão do EF Core permite conexões SQL resilientes que serão repetidas se a conexão falhar.

```
// Startup.cs from any ASP.NET Core Web API
public class Startup
{
    // Other code ...
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        // ...
        services.AddDbContext<CatalogContext>(options =>
        {
            options.UseSqlServer(Configuration["ConnectionString"],
                sqlServerOptionsAction: sqlOptions =>
            {
                sqlOptions.EnableRetryOnFailure(
                    maxRetryCount: 10,
                    maxRetryDelay: TimeSpan.FromSeconds(30),
                    errorNumbersToAdd: null);
            });
        });
        //...
    }
}
```

## Estratégias de execução e transações explícitas que usam `BeginTransaction` e várias `DbContexts`

Quando as repetições estão habilitadas nas conexões do EF Core, cada operação executada que usa o EF Core se torna sua própria operação repetível. Cada consulta e cada chamada para `SaveChanges` serão repetidas como uma unidade se ocorrer uma falha transitória.

No entanto, se seu código iniciar uma transação usando `BeginTransaction`, você estará definindo seu próprio grupo de operações que precisam ser tratadas como uma unidade. Tudo dentro da transação deverá ser revertido se ocorrer uma falha.

Ao tentar executar essa transação usando uma estratégia de execução do EF (política de repetição) e chamando `SaveChanges` de vários `DbContexts`, você receberá uma exceção como esta:

```
System.InvalidOperationException: a estratégia de execução configurada 'SqlServerRetryingExecutionStrategy' não dá suporte a transações iniciadas pelo usuário. Use a estratégia de execução retornada por 'DbContext.Database.CreateExecutionStrategy()' para executar todas as operações na transação como uma unidade repetível.
```

A solução é invocar manualmente a estratégia de execução do EF com um delegado que representa tudo que

precisa ser executado. Se ocorrer uma falha transitória, a estratégia de execução invocará o representante novamente. Por exemplo, o código a seguir mostra como ela é implementada no eShopOnContainers com dois Dbcontexts múltiplos (\_catalogContext e o IntegrationEventLogContext) ao atualizar um produto e salvar o objeto ProductPriceChangedIntegrationEvent, que precisa usar um DbContext diferente.

```
public async Task<IActionResult> UpdateProduct(
    [FromBody]CatalogItem productToUpdate)
{
    // Other code ...

    var oldPrice = catalogItem.Price;
    var raiseProductPriceChangedEvent = oldPrice != productToUpdate.Price;

    // Update current product
    catalogItem = productToUpdate;

    // Save product's data and publish integration event through the Event Bus
    // if price has changed
    if (raiseProductPriceChangedEvent)
    {
        //Create Integration Event to be published through the Event Bus
        var priceChangedEvent = new ProductPriceChangedIntegrationEvent(
            catalogItem.Id, productToUpdate.Price, oldPrice);

        // Achieving atomicity between original Catalog database operation and the
        // IntegrationEventLog thanks to a local transaction
        await _catalogIntegrationEventService.SaveEventAndCatalogContextChangesAsync(
            priceChangedEvent);

        // Publish through the Event Bus and mark the saved event as published
        await _catalogIntegrationEventService.PublishThroughEventBusAsync(
            priceChangedEvent);
    }
    // Just save the updated product because the Product's Price hasn't changed.
    else
    {
        await _catalogContext.SaveChangesAsync();
    }
}
```

O primeiro `DbContext` é `_catalogContext` e o segundo `DbContext` está dentro do objeto `_catalogIntegrationEventService`. A ação de confirmação é executada em todos os objetos `DbContext` usando uma estratégia de execução do EF.

Para atingir essa confirmação de `DbContext` múltipla, o `SaveEventAndCatalogContextChangesAsync` usa uma classe `ResilientTransaction`, como mostra o código a seguir:

```

public class CatalogIntegrationEventService : ICatalogIntegrationEventService
{
    //...
    public async Task SaveEventAndCatalogContextChangesAsync(
        IntegrationEvent evt)
    {
        // Use of an EF Core resiliency strategy when using multiple DbContexts
        // within an explicit BeginTransaction():
        // https://docs.microsoft.com/ef/core/miscellaneous/connection-resiliency
        await ResilientTransaction.New(_catalogContext).ExecuteAsync(async () =>
    {
        // Achieving atomicity between original catalog database
        // operation and the IntegrationEventLog thanks to a local transaction
        await _catalogContext.SaveChangesAsync();
        await _eventLogService.SaveEventAsync(evt,
            _catalogContext.Database.CurrentTransaction.GetDbTransaction());
    });
}
}

```

O método `ResilientTransaction.ExecuteAsync` começa basicamente uma transação usando o `DbContext` passado (`_catalogContext`) e, em seguida, faz com que o `EventLogService` use essa transação para salvar as alterações do `IntegrationEventLogContext` e, em seguida, confirmar a transação inteira.

```

public class ResilientTransaction
{
    private DbContext _context;
    private ResilientTransaction(DbContext context) =>
        _context = context ?? throw new ArgumentNullException(nameof(context));

    public static ResilientTransaction New (DbContext context) =>
        new ResilientTransaction(context);

    public async Task ExecuteAsync(Func<Task> action)
    {
        // Use of an EF Core resiliency strategy when using multiple DbContexts
        // within an explicit BeginTransaction():
        // https://docs.microsoft.com/ef/core/miscellaneous/connection-resiliency
        var strategy = _context.Database.CreateExecutionStrategy();
        await strategy.ExecuteAsync(async () =>
    {
        using (var transaction = _context.Database.BeginTransaction())
        {
            await action();
            transaction.Commit();
        }
    });
}
}

```

## Recursos adicionais

- **Resiliência de conexão e interceptação de comando com EF em uma aplicação mvc ASP.NET**  
<https://docs.microsoft.com/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/connection-resiliency-and-command-interception-with-the-entity-framework-in-an-asp-net-mvc-application>
- **Cesar de la Torre. Usando conexões e transações sql do núcleo do framework da entidade resiliente**  
<https://devblogs.microsoft.com/cesardelatorre/using-resilient-entity-framework-core-sql-connections-and-transactions-retries-with-exponential-backoff/>

[PRÓXIMO](#)

[ANTERIOR](#)

# Use o IHttpClientFactory para implementar solicitações HTTP resilientes

10/09/2020 • 15 minutes to read • [Edit Online](#)

`IHttpClientFactory` é um contrato implementado pelo `DefaultHttpClientFactory`, uma fábrica conceituada, disponível desde o .NET Core 2,1, para `HttpClient` a criação de instâncias a serem usadas em seus aplicativos.

## Problemas com a classe HttpClient original disponível no .NET Core

A classe original e conhecida `HttpClient` pode ser facilmente usada, mas, em alguns casos, ela não está sendo usada corretamente por muitos desenvolvedores.

Embora essa classe implemente `IDisposable`, a declaração e a criação de uma instância dele dentro de uma `using` instrução não é preferida porque, quando o `HttpClient` objeto é Descartado, o soquete subjacente não é lançado imediatamente, o que pode levar a um problema de *esgotamento de soquete*. Para obter mais informações sobre esse problema, consulte a postagem de blog [que você está usando HttpClient errado e está desestabilizando o software](#).

Portanto, `HttpClient` deve ser instanciado uma única vez e reutilizado durante a vida útil de um aplicativo. A criação de uma instância de uma classe `HttpClient` para cada solicitação esgotará o número de soquetes disponíveis em condições de carga pesada. Esse problema resultará em erros de `SocketException`. Abordagens possíveis para resolver o problema baseiam-se na criação do objeto `HttpClient` como singleton ou estático, conforme é explicado neste [artigo da Microsoft sobre o uso do HttpClient](#). Isso pode ser uma boa solução para aplicativos de console de curta duração ou semelhante, que são executados algumas vezes por dia.

Outro problema que os desenvolvedores executam é ao usar uma instância compartilhada do `HttpClient` em processos de execução longa. Em uma situação em que o `HttpClient` é instanciado como um singleton ou um objeto estático, ele não lida com as alterações de DNS, conforme descrito neste [problema](#) do repositório do GitHub de dotnet/tempo de execução.

No entanto, o problema não é realmente com `HttpClient` o por si, mas com o [construtor padrão para HttpClient](#), porque ele cria uma nova instância concreta do `HttpMessageHandler`, que é aquela que tem os problemas de *esgotamento de soquetes* e alterações de DNS mencionados acima.

Para resolver os problemas mencionados acima e tornar as `HttpClient` instâncias gerenciáveis, o .NET Core 2,1 introduziu a `IHttpClientFactory` interface que pode ser usada para configurar e criar `HttpClient` instâncias em um aplicativo por meio de injeção de dependência (di). Ele também fornece extensões para que o middleware baseado em Polly Aproveite a delegação de manipuladores no `HttpClient`.

O [Polly](#) é uma biblioteca de tratamento de falhas transitórias que ajuda os desenvolvedores a adicionar resiliência aos seus aplicativos, usando algumas políticas predefinidas de forma fluente e thread-safe.

## Benefícios do uso do IHttpClientFactory

A implementação atual do `IHttpClientFactory`, que também implementa `IHttpMessageHandlerFactory`, oferece os seguintes benefícios:

- Fornece um local central para nomear e configurar `HttpClient` objetos lógicos. Por exemplo, você pode configurar um cliente (agente de serviço) pré-configurado para acessar um microsserviço específico.
- Codificar o conceito de middleware de saída por meio da delegação de manipuladores no `HttpClient` e

implementação do middleware baseado em Polly para tirar proveito das políticas de Polly para resiliência.

- O `HttpClient` já tem o conceito de delegar manipuladores que podem ser vinculados uns aos outros para solicitações HTTP de saída. Você pode registrar clientes HTTP na fábrica e usar um manipulador Polly para usar políticas de Polly para repetir, CircuitBreakers e assim por diante.
- Gerencie o tempo de vida de `HttpMessageHandler` para evitar problemas/problemas mencionados que podem ocorrer ao gerenciar os `HttpClient` tempos de vida por conta própria.

#### TIP

As `HttpClient` instâncias injetadas por DI podem ser descartadas de forma segura, porque o associado `HttpMessageHandler` é gerenciado pela fábrica. Na verdade, as instâncias injetadas `HttpClient` têm o *escopo* de uma perspectiva de DI.

#### NOTE

A implementação de `IHttpClientFactory` (`DefaultHttpClientFactory`) está rigidamente ligada à implementação de DI no `Microsoft.Extensions.DependencyInjection` pacote NuGet. Para obter mais informações sobre como usar outros contêineres de DI, consulte esta [discussão do GitHub](#).

## Várias maneiras de usar o `IHttpClientFactory`

Há várias maneiras de usar o `IHttpClientFactory` no aplicativo:

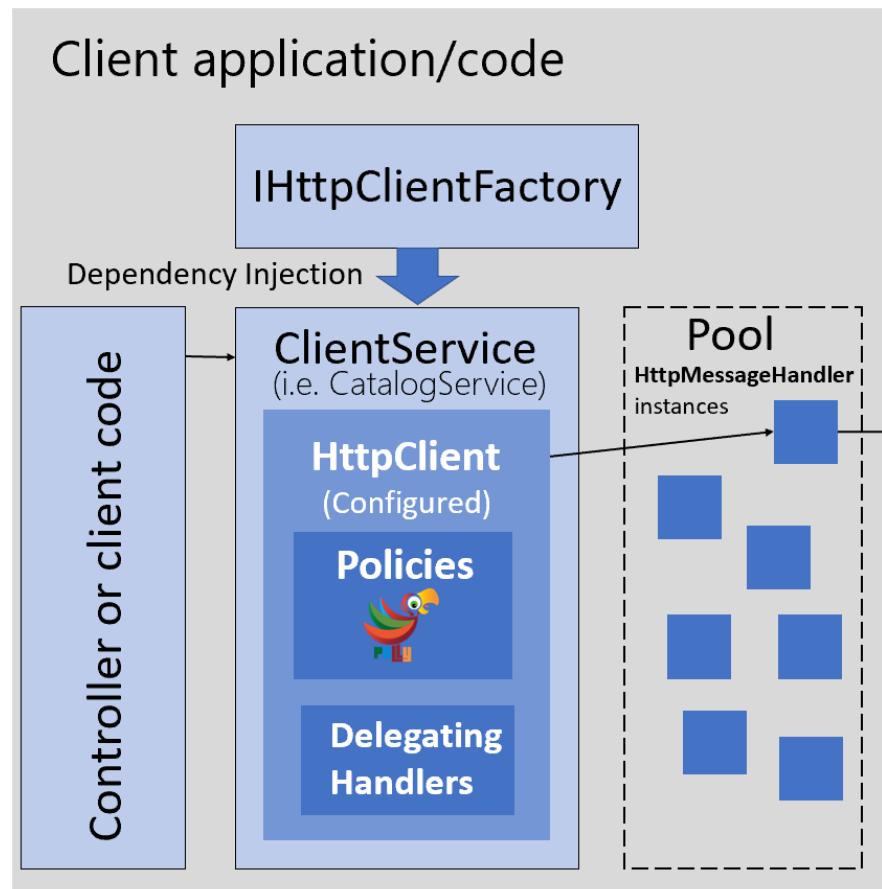
- Uso básico
- Usar clientes nomeados
- Usar clientes tipados
- Usar clientes gerados

Para fins de brevidade, essas diretrizes mostram a maneira mais estruturada de usar `IHttpClientFactory`, que é usar clientes digitados (padrão de agente de serviço). No entanto, todas as opções estão documentadas e estão listadas neste [artigo abordando o `IHttpClientFactory` USO](#).

## Como usar clientes digitados com `IHttpClientFactory`

Portanto, o que é um "cliente tipado"? É apenas um `HttpClient` que é pré-configurado para um uso específico. Essa configuração pode incluir valores específicos, como o servidor base, cabeçalhos HTTP ou tempos limite.

O diagrama a seguir mostra como os clientes tipados são usados com o `IHttpClientFactory`:



**Figura 8-4.** Usando `IHttpClientFactory` com classes de cliente tipadas.

Na imagem acima, a `ClientService` (usada por um controlador ou código de cliente) usa um `HttpClient` criado pelo registrado `IHttpClientFactory`. Essa fábrica atribui um `HttpMessageHandler` de um pool ao `HttpClient`. O `HttpClient` pode ser configurado com as políticas do Polly ao registrar o `IHttpClientFactory` no contêiner di com o método de extensão `AddHttpClient`.

Para configurar a estrutura acima, adicione `IHttpClientFactory` em seu aplicativo instalando o `Microsoft.Extensions.Http` pacote NuGet que inclui o `AddHttpClient` método de extensão para `IServiceCollection`. Esse método de extensão registra a `DefaultHttpClientFactory` classe interna a ser usada como um singleton para a interface `IHttpClientFactory`. Ele define uma configuração transitória para o `HttpMessageHandlerBuilder`. Esse manipulador de mensagens (objeto `HttpMessageHandler`), obtido de um pool, é usado pelo `HttpClient` retornado do alocador.

No próximo código, veja como `AddHttpClient()` pode ser usado para registrar clientes tipados (agentes de serviço) que precisam usar `HttpClient`.

```
// Startup.cs
//Add http client services at ConfigureServices(IServiceCollection services)
services.AddHttpClient<ICatalogService, CatalogService>();
services.AddHttpClient<IBasketService, BasketService>();
services.AddHttpClient<IOrderingService, OrderingService>();
```

O registro dos serviços do cliente, conforme mostrado no código anterior, faz com que o `DefaultClientFactory` crie um padrão `HttpClient` para cada serviço.

Você também pode adicionar a configuração específica da instância no registro para, por exemplo, configurar o endereço base e adicionar algumas políticas de resiliência, conforme mostrado no código a seguir:

```
services.AddHttpClient<ICatalogService, CatalogService>(client =>
{
    client.BaseAddress = new Uri(Configuration["BaseUrl"]);
})
    .AddPolicyHandler(GetRetryPolicy())
    .AddPolicyHandler(GetCircuitBreakerPolicy());
```

Apenas para o exemplo, você pode ver uma das políticas acima no próximo código:

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)));
}
```

Você pode encontrar mais detalhes sobre como usar o Polly no [próximo artigo](#).

### Tempos de vida de HttpClient

Sempre que você receber um objeto `HttpClient` do `IHttpClientFactory`, uma nova instância será retornada. Mas cada `HttpClient` usa um `HttpMessageHandler` que foi colocado em pool e reutilizado pelo `IHttpClientFactory` para reduzir o consumo de recursos, desde que o tempo de vida do `HttpMessageHandler` não tenha expirado.

O pooling de manipuladores é interessante porque cada manipulador normalmente gerencia suas próprias conexões de HTTP subjacentes. Criar mais manipuladores do que o necessário pode resultar em atrasos de conexão. Alguns manipuladores também mantêm as conexões abertas indefinidamente, o que pode impedir que o manipulador reaja a alterações de DNS.

Os objetos `HttpMessageHandler` no pool têm um tempo de vida que é o período de tempo em que uma instância `HttpMessageHandler` no pool pode ser reutilizada. O valor padrão é dois minutos, mas pode ser substituído por cliente tipado. Para substituí-lo, chame `SetHandlerLifetime()` no `IHttpClientBuilder` que é retornado ao criar o cliente, como mostra o código a seguir:

```
//Set 5 min as the lifetime for the HttpMessageHandler objects in the pool used for the Catalog Typed Client
services.AddHttpClient<ICatalogService, CatalogService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

Cada cliente tipado pode ter seu próprio valor de tempo de vida do manipulador configurado. Defina o tempo de vida como `InfiniteTimeSpan` para desabilitar a expiração do manipulador.

### Implementar suas classes de cliente tipado que usam o HttpClient injetado e configurado

Como uma etapa anterior, você precisa ter suas classes de cliente digitadas definidas, como as classes no código de exemplo, como 'BasketService', 'CatalogService', 'OrderingService', etc. – um cliente tipado é uma classe que aceita um `HttpClient` objeto (injetado através de seu construtor) e o usa para chamar algum serviço http remoto. Por exemplo:

```

public class CatalogService : ICatalogService
{
    private readonly HttpClient _httpClient;
    private readonly string _remoteServiceBaseUrl;

    public CatalogService(HttpClient httpClient)
    {
        _httpClient = httpClient;
    }

    public async Task<Catalog> GetCatalogItems(int page, int take,
                                                int? brand, int? type)
    {
        var uri = API.Catalog.GetAllCatalogItems(_remoteServiceBaseUrl,
                                                page, take, brand, type);

        var responseString = await _httpClient.GetStringAsync(uri);

        var catalog = JsonConvert.DeserializeObject<Catalog>(responseString);
        return catalog;
    }
}

```

O cliente digitado (`CatalogService` no exemplo) é ativado por DI (injeção de dependência), isso significa que ele pode aceitar qualquer serviço registrado em seu construtor, além de `HttpClient`.

Um cliente tipado é efetivamente um objeto transitório, isso significa que uma nova instância é criada cada vez que uma é necessária. Ele recebe uma nova `HttpClient` instância toda vez que ela é construída. No entanto, os `HttpMessageHandler` objetos no pool são os objetos que são reutilizados por várias `HttpClient` instâncias.

### Usar suas classes de cliente tipado

Por fim, depois de implementar as classes tipadas, você poderá tê-las registradas e configuradas com `AddHttpClient()`. Depois disso, você pode usá-los sempre que tiverem serviços injetados por DI. Por exemplo, em um código de página do Razor ou no controlador de um aplicativo Web MVC, como no código a seguir de eShopOnContainers:

```

namespace Microsoft.eShopOnContainers.WebMVC.Controllers
{
    public class CatalogController : Controller
    {
        private ICatalogService _catalogSvc;

        public CatalogController(ICatalogService catalogSvc) =>
            _catalogSvc = catalogSvc;

        public async Task<IActionResult> Index(int? BrandFilterApplied,
                                                int? TypesFilterApplied,
                                                int? page,
                                                [FromQuery]string errorMsg)
        {
            var itemsPage = 10;
            var catalog = await _catalogSvc.GetCatalogItems(page ?? 0,
                                                            itemsPage,
                                                            BrandFilterApplied,
                                                            TypesFilterApplied);
            //... Additional code
        }
    }
}

```

Até este ponto, o trecho de código acima só mostrou o exemplo de execução de solicitações HTTP regulares. Mas

a "mágica" vem nas seções a seguir, em que ele mostra como todas as solicitações HTTP feitas pelo `HttpClient` podem ter políticas resilientes, como repetições com retirada exponencial, separadores de circuito, recursos de segurança usando tokens de autenticação ou até mesmo qualquer outro recurso personalizado. E tudo isso pode ser feito apenas adicionando políticas e delegando manipuladores a seus clientes tipados registrados.

## Recursos adicionais

- Usando `HttpClientFactory` no .NET Core  
<https://docs.microsoft.com/aspnet/core/fundamentals/http-requests>
- `HttpClientFactory` o código-fonte no `dotnet/extensions` repositório github  
<https://github.com/dotnet/extensions/tree/master/src/HttpClientFactory>
- Polly (biblioteca de tratamento de falhas transitórias e resiliência do .NET)  
<http://www.thepollyproject.org/>
- Usando `IHttpClientFactory` sem injeção de dependência (problema do GitHub)  
<https://github.com/dotnet/extensions/issues/1345>

[ANTERIOR](#)

[AVANÇAR](#)

# Implementar tentativas de chamada HTTP com backoff exponencial com políticas IHttpClientFactory e Polly

18/03/2020 • 4 minutes to read • [Edit Online](#)

A abordagem recomendada para repetições com retirada exponencial é aproveitar as bibliotecas do .NET mais avançadas como a [biblioteca Polly](#) de software livre.

A Polly é uma biblioteca .NET que fornece resiliência e recursos de tratamento de falhas temporárias. Você pode implementar essas funcionalidades por meio da aplicação de políticas da Polly como repetição, disjuntor, isolamento do bulkhead, tempo limite e fallback. O Polly tem como alvo o .NET Framework 4.x e o .NET Standard 1.0, 1.1 e 2.0 (que suporta o .NET Core).

As etapas a seguir mostram como você `IHttpClientFactory` pode usar as repetições http com polly integrada, o que é explicado na seção anterior.

## Referenciar os pacotes ASP.NET Core 3.1

`IHttpClientFactory` está disponível desde .NET Core 2.1 no entanto recomendamos que você use os pacotes mais recentes ASP.NET Core 3.1 do NuGet em seu projeto. Você normalmente também precisa fazer `Microsoft.Extensions.Http.Polly` referência ao pacote de extensão .

## Configure um cliente com a política de Repetição da Polly, em Inicialização

Conforme mostrado nas seções anteriores, você precisa definir uma configuração cliente do HttpClient nomeada ou tipada no método padrão `Startup.ConfigureServices(...)`, mas agora, adicione o código incremental especificando a política para as repetições de HTTP com retirada exponencial, como é mostrado abaixo:

```
//ConfigureServices() - Startup.cs
services.AddHttpClient<IBasketService, BasketService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5)) //Set lifetime to five minutes
    .AddPolicyHandler(GetRetryPolicy());
```

O método `AddPolicyHandler()` é aquele que adiciona políticas aos objetos `HttpClient` que você usará. Neste caso, está adicionando uma política da Polly para Http Retries com recuo exponencial.

Para obter uma abordagem mais modular, a política de repetição de HTTP pode ser definida em um método separado no arquivo `Startup.cs`, como mostra o código a seguir:

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
            retryAttempt)));
}
```

Com a Polly, você pode definir uma política de repetição com o número de repetições, a configuração de retirada exponencial e as ações a serem executadas quando houver uma exceção de HTTP, como registrar o erro em log. Nesse caso, a política é configurada para tentar seis vezes com uma repetição exponencial, começando em dois

segundos.

## Adicionar uma estratégia de tremulação à política de repetição

Uma política de Repetição regular pode afetar o sistema em casos de alta simultaneidade e escalabilidade e sob alta contenção. Para superar os picos de novas tentativas semelhantes provenientes de muitos clientes em caso de interrupções parciais, uma boa solução alternativa é adicionar uma estratégia de variação à política/ao algoritmo de novas tentativas. Isso pode melhorar o desempenho geral do sistema de ponta a ponta, adicionando aleatoriedade à retirada exponencial. Isso espalha os picos quando surgem problemas. O princípio é ilustrado pelo seguinte exemplo:

```
Random jitterer = new Random();
var retryWithJitterPolicy = HttpPolicyExtensions
    .HandleTransientHttpError()
    .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
    .WaitAndRetryAsync(6,      // exponential back-off plus some jitter
                      retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt))
                                    + TimeSpan.FromMilliseconds(jitterer.Next(0, 100)))
    );
```

Polly fornece algoritmos de jitter prontos para a produção através do site do projeto.

## Recursos adicionais

- Padrão de repetição  
<https://docs.microsoft.com/azure/architecture/patterns/retry>
- Polly e IHttpClientFactory  
<https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory>
- Polly (biblioteca de tratamento de falhas transitórias e resiliência do .NET)  
<https://github.com/App-vNext/Polly>
- Polly: Tente com jitter  
<https://github.com/App-vNext/Polly/wiki/Retry-with-jitter>
- Marc Brooker. Jitter: Tornando as coisas melhores com a leatoriedade  
<https://brooker.co.za/blog/2015/03/21/backoff.html>

PRÓXIMO

ANTERIOR

# Implementar o padrão de disjuntor

09/04/2020 • 14 minutes to read • [Edit Online](#)

Conforme observado anteriormente, você deve tratar falhas que podem consumir uma quantidade variável de tempo de recuperação, como pode acontecer quando você tenta se conectar a um serviço ou um recurso remoto. Lidar com esse tipo de falha pode melhorar a estabilidade e a resiliência de um aplicativo.

Em um ambiente distribuído, chamadas para serviços e recursos remotos poderão falhar devido a falhas transitórias, como conexões lentas de rede e tempos limites ou se os recursos estiverem lentos ou temporariamente não disponíveis. Essas falhas geralmente são corrigidas automaticamente após um curto período, e um aplicativo em nuvem robusto deve estar preparado para lidar com elas usando uma estratégia como o “padrão de repetição”.

No entanto, também pode haver situações em que as falhas são devido a eventos inesperados que podem levar muito mais tempo para serem corrigidos. Essas falhas podem variar de gravidade de uma perda parcial de conectividade até a falha completa de um serviço. Nessas situações, talvez não tenha sentido um aplicativo repetir continuamente uma operação que provavelmente não será bem-sucedida.

Em vez disso, o aplicativo deve ser codificado para aceitar que a operação falhou e lidar com falhas adequadamente.

O mau uso das repetições de HTTP pode resultar na criação de um ataque de **DoS** (negação de serviço) dentro do próprio software. Quando um microsserviço falha ou apresenta um desempenho lento, vários clientes podem fazer novas tentativas de solicitações com falha repetidamente. Isso cria um risco perigoso de aumentar exponencialmente o tráfego direcionado ao serviço com falha.

Portanto, você precisa de algum tipo de barreira de defesa para as que solicitações excessivas sejam interrompidas quando não valer a pena continuar tentando. Essa barreira de defesa é exatamente o disjuntor.

O padrão de disjuntor tem uma finalidade diferente do “padrão de repetição”. O “padrão de repetição” permite que um aplicativo repita uma operação na expectativa de que a ela acabará sendo bem-sucedida. O padrão de disjuntor impede que um aplicativo execute uma operação que provavelmente falhará. O aplicativo pode combinar esses dois padrões. No entanto, a lógica de repetição deve reconhecer qualquer exceção retornada pelo disjuntor e deve abandonar as tentativas de repetição quando o disjuntor indica que uma falha não é transitória.

## Implementar padrão de `IHttpClientFactory` disjuntor com e Polly

Como ao implementar repetições, a abordagem recomendada para disjuntores é aproveitar bibliotecas `IHttpClientFactory` comprovadas .NET como polly e sua integração nativa com .

Adicionar uma política de `IHttpClientFactory` disjuntor em seu pipeline de middleware de saída é `IHttpClientFactory` tão simples quanto adicionar um único pedaço de código incremental ao que você já tem ao usar .

Aqui, a única adição ao código usado para repetições de chamada HTTP é o código no qual você adiciona a política de disjuntor à lista de políticas a serem usadas, conforme é mostrado no código incremental a seguir, que faz parte do método `ConfigureServices()`.

```
//ConfigureServices() - Startup.cs
services.AddHttpClient<IBasketService, BasketService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5)) //Sample. Default lifetime is 2 minutes
    .AddHttpMessageHandler<HttpClientAuthorizationDelegatingHandler>()
    .AddPolicyHandler(GetRetryPolicy())
    .AddPolicyHandler(GetCircuitBreakerPolicy());
```

O método `AddPolicyHandler()` é aquele que adiciona políticas aos objetos `HttpClient` que você usará. Nesse caso, ele está adicionando uma política da Polly a um disjuntor.

Para obter uma abordagem mais modular, a política de disjuntor é definida em um método separado chamado `GetCircuitBreakerPolicy()`, mostrado no código a seguir:

```
static IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(5, TimeSpan.FromSeconds(30));
}
```

No exemplo de código acima, a política de disjuntor é configurada para interromper ou abrir o circuito quando ocorrerem cinco falhas consecutivas ao repetir as solicitações HTTP. Quando isso acontece, o circuito será interrompido por 30 segundos: nesse período, chamadas falharão imediatamente pelo disjuntor em vez de serem colocadas. A política automaticamente interpreta [exceções relevantes e códigos de status HTTP](#) como falhas.

Os disjuntores também devem ser usados para redirecionar solicitações a uma infraestrutura de fallback quando há problemas em um recurso específico implantado em um ambiente diferente do aplicativo ou do serviço cliente que está executando a chamada HTTP. Dessa forma, se houver uma interrupção no datacenter que afete apenas os microsserviços de back-end, mas não os aplicativos cliente, os aplicativos cliente poderão redirecionar para os serviços de fallback. O Polly está planejando uma nova política para automatizar esse cenário de [política de failover](#).

Todos esses recursos são para casos em que você está gerenciando o failover de dentro do código do .NET, em vez de deixar que o Azure o gerencie automaticamente com transparência de local.

Do ponto de vista de uso, ao usar `HttpClient`, não há necessidade de adicionar `HttpClient` nada `IHttpClientFactory` de novo aqui porque o código é o mesmo do que quando usado com , como mostrado nas seções anteriores.

## Testar repetições de HTTP e disjuntores no eShopOnContainers

Sempre que você inicia a solução eShopOnContainers em um host do Docker, ela precisa iniciar vários contêineres. Alguns dos contêineres são mais lentos em iniciar e inicializar, como o contêiner do SQL Server. Isso é verdadeiro principalmente na primeira vez que você implanta o aplicativo eShopOnContainers no Docker, porque é necessário configurar as imagens e o banco de dados. O fato de que alguns contêineres iniciam mais lentamente do que outros pode fazer o restante dos serviços lançarem exceções HTTP, mesmo que você defina dependências entre contêineres no nível do Docker Compose, como explicado nas seções anteriores. Essas dependências do Docker Compose entre os contêineres são apenas no nível do processo. O processo de ponto de entrada do contêiner pode ser iniciado, mas o SQL Server pode não estar pronto para consultas. O resultado pode ser uma cascata de erros e o aplicativo pode obter uma exceção ao tentar consumir aquele contêiner específico.

Você também pode ver esse tipo de erro na inicialização quando o aplicativo está sendo implantado para a nuvem. Nesse caso, os orquestradores podem estar movendo contêineres de um nó ou VM para outro (ou seja, iniciando novas instâncias) ao equilibrar o número de contêineres nos nós do cluster.

A maneira como 'eShopOnContainers' resolve esses problemas ao iniciar todos os contêineres é usando o padrão

de repetições ilustrado anteriormente.

### Testar o disjuntor no eShopOnContainers

Há algumas maneiras de interromper/abrir o circuito e testá-lo com o eShopOnContainers.

Uma opção é reduzir o número permitido de novas tentativas a 1 na política de disjuntor e reimplantar toda a solução no Docker. Com uma única nova tentativa, há uma boa chance de que uma solicitação HTTP falhe durante a implantação, o disjuntor seja aberto e você receba um erro.

Outra opção é usar o middleware personalizado implementado no microsserviço **Cesta**. Quando esse middleware é habilitado, ele captura todas as solicitações HTTP e retorna o código de status 500. Você pode habilitar o middleware fazendo uma solicitação GET para o URI de falha, como o seguinte:

- `GET http://localhost:5103/failing`

Essa solicitação retorna o estado atual do middleware. Se o middleware estiver habilitado, a solicitação retornará o código de status 500. Se o middleware estiver desabilitado, não haverá nenhuma resposta.

- `GET http://localhost:5103/failing?enable`

Essa solicitação habilita o middleware.

- `GET http://localhost:5103/failing?disable`

Essa solicitação desabilita o middleware.

Por exemplo, quando o aplicativo estiver em execução, você poderá habilitar o middleware fazendo uma solicitação usando o seguinte URI em qualquer navegador. Observe que o microsserviço de ordenação usa a porta 5103.

`http://localhost:5103/failing?enable`

Em seguida, você pode verificar o status usando o URI `http://localhost:5103/failing`, como mostra a Figura 8-5.

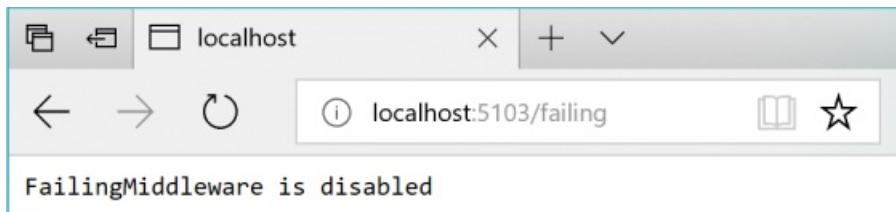


Figura 8-5. Verificando o estado do "Falhando" ASP.NET middleware – Neste caso, desativado.

Neste ponto, o de microsserviço Cesta responde com o código de status 500 sempre que você o chama ou invoca.

Depois que o middleware estiver em execução, você poderá tentar fazer um pedido do aplicativo Web MVC. Como as solicitações falham, o circuito é aberto.

No exemplo a seguir, você pode ver que o aplicativo Web MVC tem um bloco catch na lógica para fazer um pedido. Se o código capturar uma exceção de circuito aberto, ele mostrará ao usuário uma mensagem amigável informando-o para esperar.

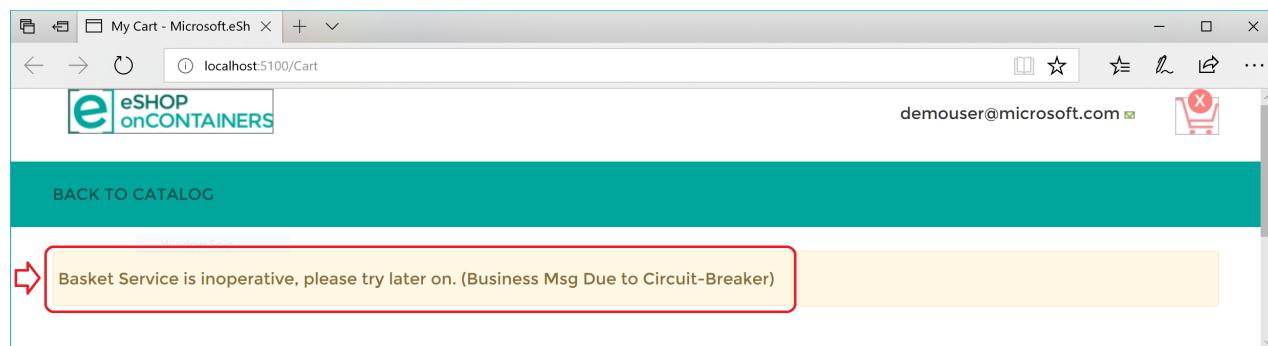
```

public class CartController : Controller
{
    ...
    public async Task<IActionResult> Index()
    {
        try
        {
            var user = _appUserParser.Parse(HttpContext.User);
            //Http requests using the Typed Client (Service Agent)
            var vm = await _basketSvc.GetBasket(user);
            return View(vm);
        }
        catch (BrokenCircuitException)
        {
            // Catches error when Basket.api is in circuit-opened mode
            HandleBrokenCircuitException();
        }
        return View();
    }

    private void HandleBrokenCircuitException()
    {
        TempData["BasketInoperativeMsg"] = "Basket Service is inoperative, please try later on. (Business message due to Circuit-Breaker)";
    }
}

```

Aqui está um resumo. A política de repetição tenta várias vezes fazer a solicitação HTTP e obtém os erros HTTP. Quando o número de repetições atinge o número máximo definido para a política de Disjuntor (nesse caso, 5), o aplicativo gera uma `BrokenCircuitException`. O resultado é uma mensagem amigável, como mostra a Figura 8-6.



**Figura 8-6.** Disjuntor retornando um erro na interface do usuário

Você pode implementar uma lógica diferente para quando abrir/interromper o circuito. Ou você poderá tentar uma solicitação HTTP para um microsserviço de back-end diferente se houver um datacenter de fallback ou um sistema de back-end redundante.

Por fim, outra possibilidade do `CircuitBreakerPolicy` é usar `Isolate` (que força o circuito a abrir e a continuar aberto) e `Reset` (que o fecha novamente). Isso pode ser usado para criar um ponto de extremidade HTTP de utilitário que invoque `Isolar` e `Reiniciar` diretamente na política. Esse ponto de extremidade HTTP também pode ser usado, adequadamente protegido, em produção para isolar temporariamente um sistema downstream, como quando você deseja atualizá-lo. Ou poderia desarmar o circuito manualmente para proteger o sistema a downstream que você suspeita ter falha.

## Recursos adicionais

- Padrão do disjuntor

<https://docs.microsoft.com/azure/architecture/patterns/circuit-breaker>

PRÓXIMO

ANTERIOR

# Monitoramento da integridade

10/09/2020 • 18 minutes to read • [Edit Online](#)

O monitoramento de integridade pode permitir informações quase em tempo real sobre o estado de seus contêineres e microsserviços. O monitoramento de integridade é fundamental para vários aspectos da operação de microsserviços e é especialmente importante quando orquestradores executam upgrades parciais de aplicativo em fases, conforme explicado posteriormente.

Aplicativos baseados em microsserviços geralmente usam pulsações ou verificações de integridade para habilitar seus monitores de desempenho, agendadores e orquestradores a controlar a variedade de serviços. Se os serviços não puderem enviar algum tipo de sinal "estou ativo", sob demanda ou em um agendamento, seu aplicativo poderá enfrentar riscos quando você implantar atualizações ou pode simplesmente detectar falhas muito tarde e não conseguir interromper as falhas em cascata que podem acabar em grandes interrupções.

No modelo comum, serviços enviam relatórios sobre o status, e essas informações são agregadas para fornecer uma exibição geral do estado de integridade do seu aplicativo. Se você estiver usando um orquestrador, poderá fornecer informações de integridade para o cluster do Orchestrator, para que o cluster possa agir de acordo. Se você investir em relatórios de integridade de alta qualidade personalizados para seu aplicativo, poderá detectar e corrigir problemas do aplicativo em execução com muito mais facilidade.

## Implementar verificações de integridade nos serviços do ASP.NET Core

Ao desenvolver um ASP.NET Core Microservice ou aplicativo Web, você pode usar o recurso de verificações de integridade internas que foi lançado no ASP .NET Core 2,2 ([Microsoft.Extensions.Diagnostics.HealthChecks](#)). Assim como muitas funcionalidades do ASP.NET Core, as verificações de integridade são fornecidas com um conjunto de serviços e um middleware.

O middleware e os serviços de verificação de integridade são fáceis de usar e fornecem funcionalidades que permitem validar se algum recurso externo necessário para seu aplicativo (como um banco de dados do SQL Server ou uma API remota) está funcionando corretamente. Quando você usa essa funcionalidade, também pode decidir o que significa se o recurso está íntegro, como explicaremos mais adiante.

Para usar essa funcionalidade com eficiência, você precisará primeiro configurar serviços em seus microsserviços. Em segundo lugar, é necessário um aplicativo de front-end que consulte os relatórios de integridade. O aplicativo de front-end pode ser um aplicativo de relatório personalizado ou um orquestrador em si que pode reagir de acordo com os estados de integridade.

### Usar a funcionalidade HealthChecks nos microsserviços de back-end do ASP.NET

Nesta seção, você aprenderá a implementar o recurso HealthChecks em um aplicativo de API Web do 3,1 de exemplo ASP.NET Core ao usar o pacote [Microsoft.Extensions.Diagnostics.HealthChecks](#). A implementação desse recurso em um microsserviço de grande escala, como o eShopOnContainers, é explicado na próxima seção.

Para começar, você precisa definir o que constitui o status íntegro para cada microsserviço. No aplicativo de exemplo, definimos que o Microservice é íntegro se sua API estiver acessível via HTTP e seu banco de dados de SQL Server relacionado também estiver disponível.

No .NET Core 3,1, com as APIs internas, você pode configurar os serviços, adicionar uma verificação de integridade para o microsserviço e seu banco de dados dependente SQL Server dessa maneira:

```
// Startup.cs from .NET Core 3.1 Web API sample
//
public void ConfigureServices(IServiceCollection services)
{
    //...
    // Registers required services for health checks
    services.AddHealthChecks()
        // Add a health check for a SQL Server database
        .AddCheck(
            "OrderingDB-check",
            new SqlConnectionHealthCheck(Configuration["ConnectionString"]),
            HealthStatus.Unhealthy,
            new string[] { "orderingdb" });
}
```

No código anterior, o `services.AddHealthChecks()` método configura uma verificação http básica que retorna um código de status 200 com "íntegro". Além disso, o `AddCheck()` método de extensão configura um personalizado `SqlConnectionHealthCheck` que verifica a integridade do banco de dados SQL relacionado.

O método `AddCheck()` adiciona uma nova verificação de integridade com um nome especificado e a implementação do tipo `IHealthCheck`. Você pode adicionar várias verificações de integridade usando o método `addcheck`, para que um microserviço não forneça um status "íntegro" até que todas as suas verificações estejam íntegras.

`SqlConnectionHealthCheck` é uma classe personalizada que implementa `IHealthCheck`, que usa uma cadeia de conexão como um parâmetro de construtor e executa uma consulta simples para verificar se a conexão com o Banco de Dados SQL foi bem-sucedida. Ela retornará `HealthCheckResult.Healthy()` se a consulta for executada com êxito e um `FailureStatus` com a exceção real em caso de falha.

```

// Sample SQL Connection Health Check
public class SqlConnectionHealthCheck : IHealthCheck
{
    private static readonly string DefaultTestQuery = "Select 1";

    public string ConnectionString { get; }

    public string TestQuery { get; }

    public SqlConnectionHealthCheck(string connectionString)
        : this(connectionString, testQuery: DefaultTestQuery)
    {
    }

    public SqlConnectionHealthCheck(string connectionString, string testQuery)
    {
        ConnectionString = connectionString ?? throw new ArgumentNullException(nameof(connectionString));
        TestQuery = testQuery;
    }

    public async Task<HealthCheckResult> CheckHealthAsync(HealthCheckContext context, CancellationToken cancellationToken = default(CancellationToken))
    {
        using (var connection = new SqlConnection(ConnectionString))
        {
            try
            {
                await connection.OpenAsync(cancellationToken);

                if (TestQuery != null)
                {
                    var command = connection.CreateCommand();
                    command.CommandText = TestQuery;

                    await command.ExecuteNonQueryAsync(cancellationToken);
                }
            }
            catch (DbException ex)
            {
                return new HealthCheckResult(status: context.Registration.FailureStatus, exception: ex);
            }
        }

        return HealthCheckResult.Healthy();
    }
}

```

Observe que, no código anterior, `Select 1` é a consulta usada para verificar a integridade do banco de dados. Para monitorar a disponibilidade de seus microserviços, os orquestradores como o Kubernetes realizam periodicamente verificações de integridade enviando solicitações para testar os microserviços. É importante manter suas consultas de banco de dados eficientes para que essas operações sejam rápidas e não resultem em uma maior utilização de recursos.

Por fim, adicione um middleware que responda ao caminho da URL `/hc` :

```

// Startup.cs from .NET Core 3.1 Web API sample
//
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    //...
    app.UseEndpoints(endpoints =>
    {
        //...
        endpoints.MapHealthChecks("/hc");
        //...
    });
    //...
}

```

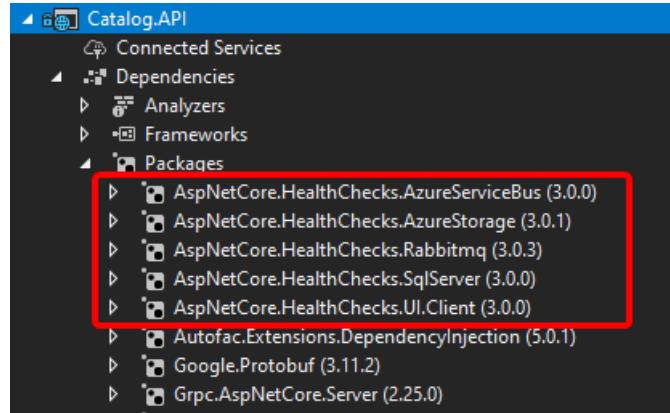
Quando o ponto de extremidade `<yourmicroservice>/hc` é invocado, ele executa todas as verificações de integridade configuradas no método `AddHealthChecks()` na classe de Inicialização e mostra o resultado.

### Implementação de HealthChecks no eShopOnContainers

Os microsserviços do eShopOnContainers dependem de vários serviços para realizar suas tarefas. Por exemplo, o microsserviço `Catalog.API` do eShopOnContainers depende de muitos serviços, como Armazenamento de Blobs do Azure, SQL Server e RabbitMQ. Portanto, ele tem várias verificações de integridade adicionadas usando o método `AddCheck()`. Para cada serviço dependente, uma `IHealthCheck` implementação personalizada que define seu respectivo status de integridade precisaria ser adicionada.

O projeto de software livre [AspNetCore.Diagnostics.HealthChecks](#) resolve esse problema fornecendo implementações de verificação de integridade personalizadas para cada um desses serviços corporativos, que são criados com base no .net Core 3.1. Cada verificação de integridade está disponível como um pacote NuGet individual que pode ser adicionado ao projeto com facilidade. O eShopOnContainers os usa extensivamente em todos os seus microsserviços.

Por exemplo, no microsserviço `Catalog.API`, os seguintes pacotes NuGet foram adicionados:



**Figura 8-7.** Verificações de Integridade personalizadas implementadas em Catalog.API usando `AspNetCore.Diagnostics.HealthChecks`

No seguinte código, as implementações de verificação de integridade são adicionadas a cada serviço dependente e, em seguida, o middleware é configurado:

```

// Startup.cs from Catalog.api microservice
//
public static IServiceCollection AddCustomHealthCheck(this IServiceCollection services, IConfiguration
configuration)
{
    var accountName = configuration.GetValue<string>("AzureStorageAccountName");
    var accountKey = configuration.GetValue<string>("AzureStorageAccountKey");

    var hcBuilder = services.AddHealthChecks();

    hcBuilder
        .AddSqlServer(
            configuration["ConnectionString"],
            name: "CatalogDB-check",
            tags: new string[] { "catalogdb" });

    if (!string.IsNullOrEmpty(accountName) && !string.IsNullOrEmpty(accountKey))
    {
        hcBuilder
            .AddAzureBlobStorage(
                $"DefaultEndpointsProtocol=https;AccountName={accountName};AccountKey=
{accountKey};EndpointSuffix=core.windows.net",
                name: "catalog-storage-check",
                tags: new string[] { "catalogstorage" });
    }
    if (configuration.GetValue<bool>("AzureServiceBusEnabled"))
    {
        hcBuilder
            .AddAzureServiceBusTopic(
                configuration["EventBusConnection"],
                topicName: "eshop_event_bus",
                name: "catalog-servicebus-check",
                tags: new string[] { "servicebus" });
    }
    else
    {
        hcBuilder
            .AddRabbitMQ(
                $"amqp://{{configuration["EventBusConnection"]}}",
                name: "catalog-rabbitmqbus-check",
                tags: new string[] { "rabbitmqbus" });
    }
}

return services;
}

```

Por fim, adicione o middleware HealthCheck para escutar o ponto de extremidade "/HC":

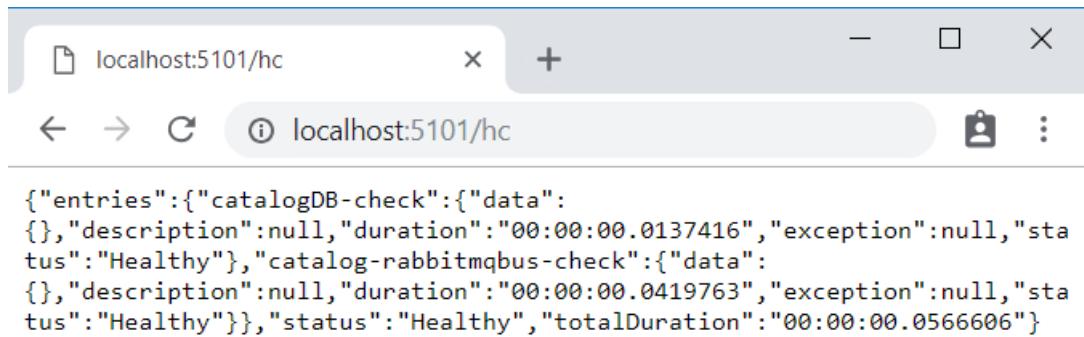
```

// HealthCheck middleware
app.UseHealthChecks("/hc", new HealthCheckOptions()
{
    Predicate = _ => true,
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});

```

## Consultar os microsserviços para relatar o status de integridade

Depois de configurar as verificações de integridade, como descrito neste artigo e colocar o microsserviço em execução no Docker, você poderá verificar diretamente se ele está íntegro usando um navegador. É necessário publicar a porta do contêiner no host do Docker, para que você possa acessar o contêiner por meio do IP externo do host do Docker ou por meio de `localhost`, como mostra a Figura 8-8.



```
{"entries": {"catalogDB-check": {"data": {}, "description": null, "duration": "00:00:00.0137416", "exception": null, "status": "Healthy"}, "catalog-rabbitmqbus-check": {"data": {}, "description": null, "duration": "00:00:00.0419763", "exception": null, "status": "Healthy"}}, "status": "Healthy", "totalDuration": "00:00:00.0566606"}
```

Figura 8-8. Verificando o status da integridade de um único serviço em um navegador

Nesse teste, você pode ver que o microsserviço `Catalog.API` (em execução na porta 5101) está íntegro, retornando o status HTTP 200 e informações de status em JSON. O serviço também verificou a integridade de sua dependência do banco de dados do SQL Server e do RabbitMQ e, portanto, a verificação de integridade relatou a si própria como íntegra.

## Usar watchdogs

Um watchdog é um serviço separado que pode inspecionar a integridade e a carga entre serviços e relatar a integridade dos microsserviços consultando a biblioteca `HealthChecks` já apresentada. Isso pode ajudar a evitar erros que não seriam detectados com base no modo de exibição de um único serviço. Os watchdogs também são um bom local para hospedar os códigos que podem realizar ações de correção para condições conhecidas sem qualquer interação com o usuário.

O exemplo de eShopOnContainers contém uma página da Web que exibe os relatórios de verificação de integridade de exemplo, como mostra a Figura 8-9. Este é o watchdog mais simples que você pode ter, pois ele mostra simplesmente o estado dos aplicativos Web e dos microsserviços no eShopOnContainers. Geralmente, um watchdog também executa ações quando detecta estados não íntegro.

Uma boa notícia é que `AspNetCore.Diagnostics.HealthChecks` também fornece o pacote NuGet `AspNetCore.HealthChecks.UI`, que pode ser usado para exibir os resultados da verificação de integridade dos URLs configurados.

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.** Relatório de verificação de integridade de exemplo em eShopOnContainers

Em resumo, esse serviço de Watchdog consulta o ponto de extremidade "/HC" de cada microserviço. Isso executará todas as verificações de integridade definidas dentro dele e retornará um estado de integridade geral dependendo todas essas verificações. O HealthChecksUI é fácil de consumir com algumas entradas de configuração e duas linhas de código que precisam ser adicionadas ao *Startup.cs* do serviço de Watchdog.

Arquivo de configuração de exemplo para interface do usuário da verificação de integridade:

```
// Configuration
{
  "HealthChecks-UI": {
    "HealthChecks": [
      {
        "Name": "Ordering HTTP Check",
        "Uri": "http://localhost:5102/hc"
      },
      {
        "Name": "Ordering HTTP Background Check",
        "Uri": "http://localhost:5111/hc"
      },
      //...
    ]
  }
}
```

Arquivo *Startup.cs* que adiciona HealthChecksUI:

```

// Startup.cs from WebStatus(Watch Dog) service
//
public void ConfigureServices(IServiceCollection services)
{
    ...
    // Registers required services for health checks
    services.AddHealthChecksUI();
}
...
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...
    app.UseHealthChecksUI(config => config.UIPath = "/hc-ui");
    ...
}

```

## Verificações de integridade ao usar orquestradores

Para monitorar a disponibilidade de seus microsserviços, orquestradores como Kubernetes e Service Fabric realizam verificações de integridade periodicamente enviando solicitações para testar os microsserviços. Quando um orquestrador determina que um serviço/contêiner não está íntegro, ele para de rotear solicitações para aquela instância. Ele também geralmente cria uma nova instância do contêiner.

Por exemplo, a maioria dos orquestradores pode usar verificações de integridade para gerenciar implantações de tempo de inatividade zero. Somente quando o status de um serviço/contêiner é alterado para íntegro o orquestrador começa a rotear o tráfego para instâncias de serviço/contêiner.

O monitoramento de integridade é especialmente importante quando um orquestrador executa uma atualização do aplicativo. Alguns orquestradores (como o Azure Service Fabric) atualizam os serviços em fases, por exemplo, podem atualizar um quinto da superfície de cluster para cada upgrade de aplicativo. O conjunto de nós que é atualizado ao mesmo tempo é mencionado como um *domínio de atualização*. Depois de cada domínio de atualização ter sido atualizado e estar disponível para os usuários, esse domínio de atualização deverá passar por verificações de integridade antes que a implantação passe para o próximo domínio de atualização.

Outro aspecto da integridade do serviço são os relatórios de métrica do serviço. Este é uma funcionalidade avançada do modelo de integridade de alguns orquestradores, como o Service Fabric. As métricas são importantes ao usar um orquestrador, pois elas são usadas para equilibrar o uso de recursos. As métricas também podem ser um indicador de integridade do sistema. Por exemplo, você pode ter um aplicativo com muitos microsserviços, e cada instância relata uma métrica de RPS (solicitações por segundo). Se um serviço estiver usando mais recursos (memória, processador etc.) que outro, o orquestrador poderá mover instâncias de serviço do cluster para tentar manter até mesmo a utilização de recursos.

Observe que o Azure Service Fabric fornece seu próprio [modelo de monitoramento de integridade](#), que é mais avançado do que as verificações de integridade simples.

## Monitoramento avançado: visualização, análise e alertas

A parte final do monitoramento é visualizar o fluxo de eventos, informar sobre o desempenho do serviço e alertar ao detectar um problema. Você pode usar diferentes soluções para esse aspecto do monitoramento.

Você pode usar aplicativos personalizados simples que mostram o estado dos serviços, como a página personalizada mostrada na explicação do [AspNetCore.Diagnostics.HealthChecks](#). Ou você pode usar ferramentas mais avançadas, como o [Azure Monitor](#) para gerar alertas com base em fluxo de eventos.

Por fim, se você estiver armazenando todos os fluxos de eventos, poderá usar o Microsoft Power BI ou uma solução de terceiros, como o Kibana ou o Splunk, para visualizar os dados.

## Recursos adicionais

- Interface do usuário do HealthChecks e do HealthChecks para ASP.NET Core  
<https://github.com/Xabril/AspNetCore.Diagnostics.HealthChecks>
- Introdução ao monitoramento de integridade Service Fabric  
<https://docs.microsoft.com/azure/service-fabric/service-fabric-health-introduction>
- Azure Monitor  
<https://azure.microsoft.com/services/monitor/>

[ANTERIOR](#)

[AVANÇAR](#)

# Proteger microsserviços .NET e aplicativos Web

10/09/2020 • 20 minutes to read • [Edit Online](#)

Há muitos aspectos sobre segurança em microsserviços e aplicativos Web que o tópico poderia facilmente pegar vários livros como este. Portanto, nesta seção, nos concentraremos na autenticação, na autorização e nos segredos do aplicativo.

## Implementar a autenticação em microsserviços .NET e aplicativos Web

Geralmente é necessário que os recursos e as APIs publicados por um serviço sejam limitados a determinados usuários ou clientes confiáveis. A primeira etapa para tomar esses tipos de decisões de confiança no nível da API é a autenticação. A autenticação é o processo de verificar a identidade de um usuário de forma confiável.

Em cenários de microsserviço, a autenticação é normalmente feita de forma centralizada. Se você estiver usando um Gateway de API, o gateway será um bom lugar para fazer a autenticação, conforme é mostrado na Figura 9-1. Se você usar esta abordagem, verifique se os microsserviços individuais não podem ser acessados diretamente (sem o Gateway de API), a não ser que haja uma segurança adicional em vigor para autenticar mensagens que entram ou não pelo gateway.

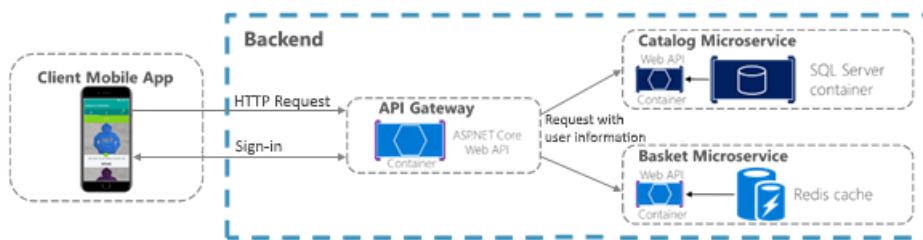


Figura 9-1. Autenticação centralizada com um Gateway de API

Quando o Gateway de API centraliza a autenticação, ele adiciona informações do usuário ao encaminhar solicitações para os microsserviços. Se os serviços puderem ser acessados diretamente, um serviço de autenticação, como o Azure Active Directory ou um microsserviço de autenticação dedicado atuando como um serviço de token de segurança (STS), poderá ser usado para autenticar os usuários. As decisões de confiança são compartilhadas entre os serviços com tokens de segurança ou cookies. (Esses tokens podem ser compartilhados entre ASP.NET Core aplicativos, se necessário, implementando o [compartilhamento de cookies](#).) Esse padrão é ilustrado na Figura 9-2.

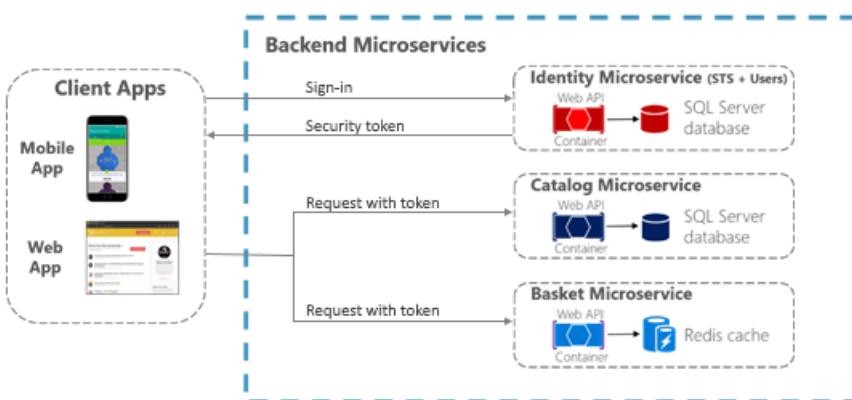


Figura 9-2. Autenticação por microsserviço de identidade; a confiança é compartilhada usando um token de autorização

Quando os microsserviços são acessados diretamente, a confiança, que inclui autenticação e autorização, é tratada

por um token de segurança emitido por um microsserviço dedicado, compartilhado entre os microsserviços.

## Autenticar usando o ASP.NET Core Identity

O mecanismo principal no ASP.NET Core para identificar os usuários de um aplicativo é o sistema de associação de [identidade ASP.NET Core](#). A Identidade do ASP.NET Core armazena informações de usuário (incluindo informações de logon, funções e declarações) em um repositório de dados configurado pelo desenvolvedor. Normalmente, o armazenamento de dados do ASP.NET Core Identity é um repositório do Entity Framework fornecido no pacote `Microsoft.AspNetCore.Identity.EntityFrameworkCore`. No entanto, os repositórios personalizados ou outros pacotes de terceiros podem ser usados para armazenar informações de identidade no Armazenamento de Tabelas do Azure, no CosmosDB ou em outros locais.

### TIP

ASP.NET Core 2,1 e posterior fornece [ASP.NET Core identidade](#) como uma [biblioteca de classes Razor](#), de modo que você não verá grande parte do código necessário em seu projeto, como foi o caso de versões anteriores. Para obter detalhes sobre como personalizar o código de identidade para atender às suas necessidades, consulte [Scaffold Identity in ASP.NET Core Projects](#).

O código a seguir é obtido do modelo de projeto ASP.NET Core aplicativo Web MVC 3,1 com a autenticação de conta de usuário individual selecionada. Ele mostra como configurar ASP.NET Core identidade usando Entity Framework Core no `Startup.ConfigureServices` método.

```
public void ConfigureServices(IServiceCollection services)
{
    //...
    services.AddDbContext<ApplicationContext>(options =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationContext>();
    services.AddRazorPages();
    //...
}
```

Quando ASP.NET Core identidade estiver configurada, você a habilitará adicionando o `app.UseAuthentication()` e `endpoints.MapRazorPages()` conforme mostrado no código a seguir no método do serviço `Startup.Configure`:

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    //...
    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
    //...
}
```

## IMPORTANT

As linhas no código anterior devem estar na ordem mostrada para que a identidade funcione corretamente.

Usar o ASP.NET Core Identity permite vários cenários:

- Criar informações de novo usuário usando o tipo `UserManager` (`userManager.CreateAsync`).
- Autenticar usuários usando o tipo `SignInManager`. Você pode usar o `signInManager.SignInAsync` para entrar diretamente ou `signInManager.PasswordSignInAsync` para confirmar se a senha do usuário está correta e, em seguida, conectá-la.
- Identifique um usuário com base nas informações armazenadas em um cookie (que é lido por ASP.NET Core middleware de identidade) para que as solicitações subsequentes de um navegador incluam uma identidade e declarações do usuário conectado.

O ASP.NET Core Identity também dá suporte à [autenticação de dois fatores](#).

Para cenários de autenticação que usam um armazenamento de dados de usuário local e que persistem a identidade entre solicitações usando cookies (como é comum para aplicativos Web MVC), o ASP.NET Core Identity é uma solução recomendada.

### Autenticar com provedores externos

O ASP.NET Core também dá suporte ao uso de [provedores de autenticação externos](#) para permitir que os usuários entrem por meio de fluxos OAuth 2.0. Isso significa que os usuários podem entrar usando os processos de autenticação existentes de provedores como Microsoft, Google, Facebook ou Twitter e associar essas identidades a um ASP.NET Core Identity no seu aplicativo.

Para usar a autenticação externa, além de incluir o middleware de autenticação, conforme mencionado anteriormente, usando o `app.UseAuthentication()` método, você também precisa registrar o provedor externo no `Startup`, conforme mostrado no exemplo a seguir:

```
public void ConfigureServices(IServiceCollection services)
{
    //...
    services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationContext>();

    services.AddAuthentication()
        .AddMicrosoftAccount(microsoftOptions =>
    {
        microsoftOptions.ClientId = Configuration["Authentication:Microsoft:ClientId"];
        microsoftOptions.ClientSecret = Configuration["Authentication:Microsoft:ClientSecret"];
    })
        .AddGoogle(googleOptions => { ... })
        .AddTwitter(twitterOptions => { ... })
        .AddFacebook(facebookOptions => { ... });
    //...
}
```

Os provedores de autenticação externos populares e seus pacotes NuGet associados são mostrados na tabela a seguir:

PROVEDOR	PACOTE
Microsoft	<code>Microsoft.AspNetCore.Authentication.MicrosoftAccount</code>

PROVEDOR	PACOTE
Google	<code>Microsoft.AspNetCore.Authentication.Google</code>
Facebook	<code>Microsoft.AspNetCore.Authentication.Facebook</code>
Twitter	<code>Microsoft.AspNetCore.Authentication.Twitter</code>

Em todos os casos, você deve concluir um procedimento de registro de aplicativo que é dependente do fornecedor e que geralmente envolve:

1. Obtendo uma ID do aplicativo cliente.
2. Obtendo um segredo do aplicativo cliente.
3. Configurando uma URL de redirecionamento, que é tratada pelo middleware de autorização e pelo provedor registrado
4. Opcionalmente, a configuração de uma URL de saída para tratar corretamente a saída em um cenário de logon único (SSO).

Para obter detalhes sobre como configurar seu aplicativo para um provedor externo, consulte a [autenticação do provedor externo na documentação do ASP.NET Core](#).

#### TIP

Todos os detalhes são tratados pelo middleware de autorização e pelos serviços mencionados anteriormente. Portanto, basta escolher a opção de autenticação de **conta de usuário individual** ao criar o projeto de aplicativo Web de código ASP.net no Visual Studio, como mostrado na Figura 9-3, além de registrar os provedores de autenticação mencionados anteriormente.

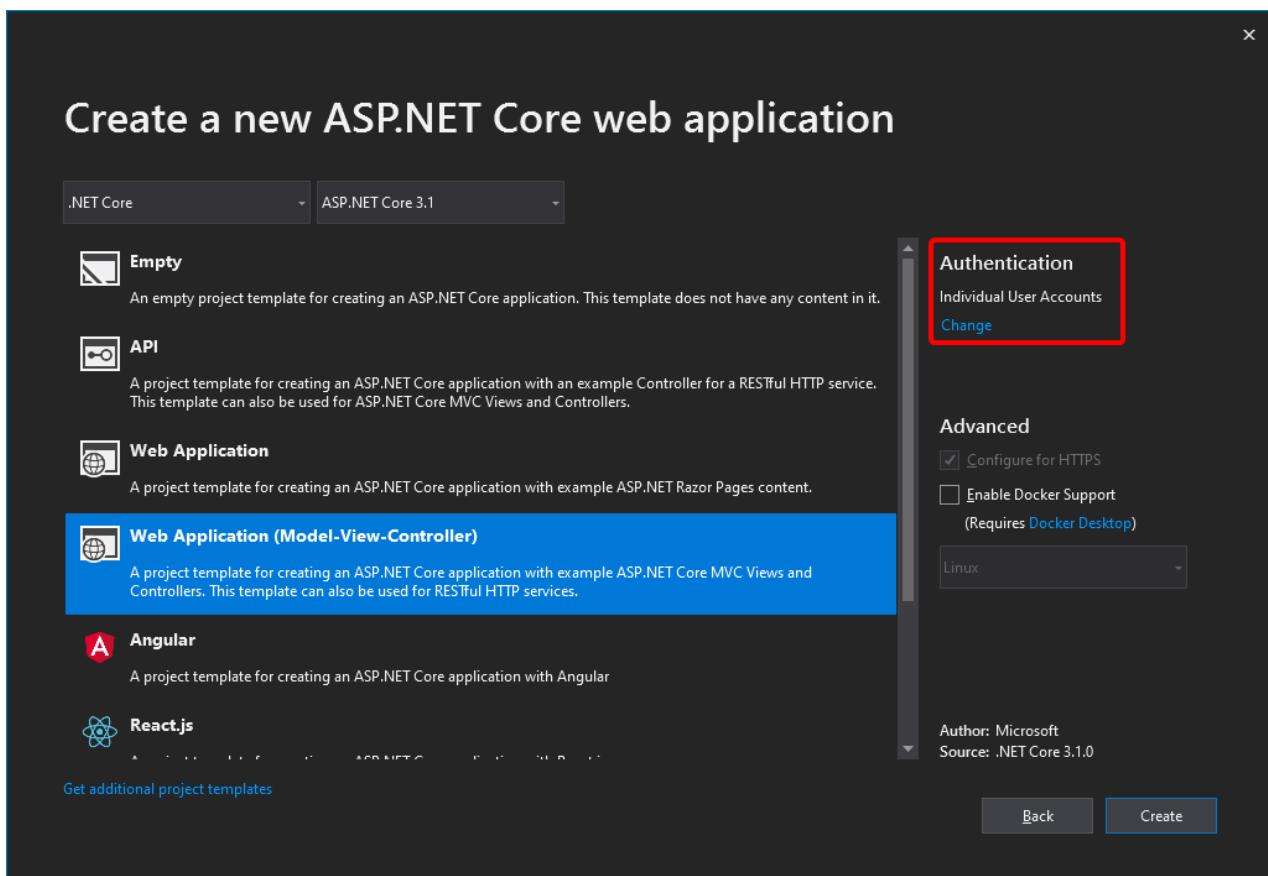


Figura 9-3. Selecionando a opção contas de usuário individuais, para usar a autenticação externa, ao criar um projeto de aplicativo Web no Visual Studio 2019.

Além dos provedores de autenticação externos listados anteriormente, estão disponíveis pacotes de terceiros que fornecem middleware para o uso de vários outros provedores de autenticação externos. Para obter uma lista, consulte o repositório [ASP.NET Security OAuth Providers](#) no github.

Você também pode criar seu próprio middleware de autenticação externa para resolver alguma necessidade especial.

### **Autenticar com tokens de portador**

Autenticar com o ASP.NET Core Identity (ou com Identity e também com provedores de autenticação externos) funciona bem para vários cenários de aplicativo Web nos quais é apropriado armazenar informações do usuário em um cookie. Em outros cenários, no entanto, os cookies não são uma maneira natural de persistir e transmitir dados.

Por exemplo, em uma API Web do ASP.NET Core que expõe pontos de extremidade RESTful que podem ser acessados por SPAs (Aplicativos de Única Página), por clientes nativos ou até mesmo por outras APIs Web, geralmente é melhor usar a autenticação de token de portador. Esses tipos de aplicativos não funcionam com cookies, mas podem facilmente recuperar um token de portador e incluí-lo no cabeçalho de autorização das próximas solicitações. Para habilitar a autenticação de token, o ASP.NET Core dá suporte a várias opções para usar [OAuth 2.0](#) e [OpenID Connect](#).

### **Autenticar com um provedor de identidade OAuth 2.0 ou OpenID Connect**

Se as informações de usuário estiverem armazenadas no Azure Active Directory ou em outra solução de identidade que dê suporte a OpenID Connect ou OAuth 2.0, você poderá usar o pacote [Microsoft.AspNetCore.Authentication.OpenIdConnect](#) para autenticar usando o fluxo de trabalho do OpenID Connect. Por exemplo, para autenticar no microserviço Identity.Api em eShopOnContainers, um aplicativo Web do ASP.NET Core pode usar o middleware desse pacote, conforme mostrado no seguinte exemplo simplificado em `Startup.cs`:

```

// Startup.cs

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    //...
    app.UseAuthentication();
    //...
    app.UseEndpoints(endpoints =>
    {
        //...
    });
}

public void ConfigureServices(IServiceCollection services)
{
    var identityUrl = Configuration.GetValue<string>("IdentityUrl");
    var callBackUrl = Configuration.GetValue<string>("CallBackUrl");
    var sessionCookieLifetime = Configuration.GetValue("SessionCookieLifetimeMinutes", 60);

    // Add Authentication services

    services.AddAuthentication(options =>
    {
        options.DefaultScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme = JwtBearerDefaults.AuthenticationScheme;
    })
    .AddCookie(setup => setup.ExpireTimeSpan = TimeSpan.FromMinutes(sessionCookieLifetime))
    .AddOpenIdConnect(options =>
    {
        options.SignInScheme = CookieAuthenticationDefaults.AuthenticationScheme;
        options.Authority = identityUrl.ToString();
        options.SignedOutRedirectUri = callBackUrl.ToString();
        options.ClientId = useLoadTest ? "mvctest" : "mvc";
        options.ClientSecret = "secret";
        options.ResponseType = useLoadTest ? "code id_token token" : "code id_token";
        options.SaveTokens = true;
        options.GetClaimsFromUserInfoEndpoint = true;
        options.RequireHttpsMetadata = false;
        options.Scope.Add("openid");
        options.Scope.Add("profile");
        options.Scope.Add("orders");
        options.Scope.Add("basket");
        options.Scope.Add("marketing");
        options.Scope.Add("locations");
        options.Scope.Add("webshoppingagg");
        options.Scope.Add("orders.signalrhub");
    });
}

```

Observe que quando você usa este fluxo de trabalho, o middleware do ASP.NET Core Identity não é necessário, porque todo o armazenamento de informações de usuário e toda a autenticação são manipulados pelo serviço de identidade.

### **Emitir tokens de segurança de um serviço do ASP.NET Core**

Se preferir emitir tokens de segurança para usuários do ASP.NET Core Identity local em vez de usar um provedor de identidade externo, você poderá usufruir de algumas boas bibliotecas de terceiros.

[IdentityServer4](#) e [OpenIddict](#) são provedores do OpenID Connect que se integram facilmente ao ASP.NET Core Identity para permitir que você emita tokens de segurança de um serviço do ASP.NET Core. A [IdentityServer4 documentation](#) (Documentação do IdentityServer4) tem instruções detalhadas para usar a biblioteca. No entanto, as etapas básicas para usar o IdentityServer4 para emitir tokens são as seguintes.

1. Você chama o aplicativo. UselidentityServer no método Startup.Configuar para adicionar IdentityServer4 ao

pipeline de processamento de solicitação HTTP do aplicativo. Isso permite que a biblioteca atenda solicitações dos pontos de extremidade do OpenID Connect e do OAuth2 como /connect/token.

2. Você pode configurar o IdentityServer4 no Startup.ConfigureServices fazendo uma chamada para services.AddIdentityServer.
3. Configure o servidor de identidade definindo os seguintes dados:
  - As [credenciais](#) a serem usadas para assinar.
  - Os [recursos de identidade e da API](#) aos quais os usuários podem solicitar acesso:
    - Os recursos da API representam dados ou funcionalidades protegidos que o usuário pode acessar com um token de acesso. Um exemplo de um recurso da API seria uma API Web (ou conjunto de APIs) que exige autorização.
    - Os recursos de identidade representam informações (declarações) que são concedidas a um cliente para identificar um usuário. As declarações podem incluir o nome de usuário, o endereço de email e assim por diante.
  - Os [clientes](#) que se conectarão para solicitar tokens.
  - O mecanismo de armazenamento de informações do usuário, como o [ASP.NET Core Identity](#) ou um outro.

Ao especificar os clientes e os recursos para o IdentityServer4 usar, você pode passar uma coleção [IEnumerable<T>](#) do tipo apropriado aos métodos que usam os repositórios de clientes ou de recursos na memória. Ou, para cenários mais complexos, você pode fornecer os tipos de provedor de clientes ou de recursos por meio de injeção de dependência.

Um exemplo de configuração para o IdentityServer4 usar recursos e clientes na memória fornecidos por um tipo IClientStore personalizado pode ser semelhante ao exemplo a seguir:

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    //...
    services.AddSingleton<IClientStore, CustomClientStore>();
    services.AddIdentityServer()
        .AddSigningCredential("CN=sts")
        .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
        .AddAspNetIdentity<ApplicationUser>();
    //...
}
```

## Consumir tokens de segurança

Autenticar em relação a um ponto de extremidade do OpenID Connect ou emitir seus próprios tokens de segurança cobre alguns cenários. Mas, e um serviço que apenas precisa limitar o acesso a esses usuários que têm tokens de segurança válidos que foram fornecidos por um serviço diferente?

Para esse cenário, o middleware de autenticação que manipula os tokens JWT está disponível no pacote [Microsoft.AspNetCore.Authentication.JwtBearer](#). JWT representa "Token Web JSON" e é um formato de token de segurança comum (definido pelo RFC 7519) para comunicação de declarações de segurança. Um exemplo simplificado de como usar o middleware para consumir esses tokens pode parecer com este fragmento de código, tirado do microsserviço Ordering.API de eShopOnContainers.

```

// Startup.cs

public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    //...
    // Configure the pipeline to use authentication
    app.UseAuthentication();
    //...
    app.UseEndpoints(endpoints =>
    {
        //...
    });
}

public void ConfigureServices(IServiceCollection services)
{
    var identityUrl = Configuration.GetValue<string>("IdentityUrl");

    // Add Authentication services

    services.AddAuthentication(options =>
    {
        options.DefaultAuthenticateScheme =
            AspNetCore.Authentication.JwtBearer.JwtBearerDefaults.AuthenticationScheme;
        options.DefaultChallengeScheme =
            AspNetCore.Authentication.JwtBearer.JwtBearerDefaults.AuthenticationScheme;

    }).AddJwtBearer(options =>
    {
        options.Authority = identityUrl;
        options.RequireHttpsMetadata = false;
        options.Audience = "orders";
    });
}

```

Os parâmetros nesse tipo de uso são:

- `Audience` representa o receptor do token ou do recurso de entrada ao qual o token concede acesso. Se o valor especificado nesse parâmetro não corresponder ao parâmetro no token, o token será rejeitado.
- `Authority` é o endereço do servidor de autenticação que emite o token. O middleware de portador do JWT usa esse URI para obter a chave pública que pode ser usada para validar a assinatura do token. O middleware também confirma se o parâmetro `iss` no token corresponde a esse URI.

Outro parâmetro, `RequireHttpsMetadata`, é útil para testes. Defina esse parâmetro como `false` para poder testar em ambientes nos quais você não tenha certificados. Em implantações reais, os tokens de portador do JWT sempre devem ser passados apenas por HTTPS.

Com este middleware em vigor, os tokens JWT são extraídos automaticamente dos cabeçalhos de autorização. Eles são desserializados, validados (usando os valores nos parâmetros `Audience` e `Authority`) e armazenados como informações de usuário a serem referenciadas posteriormente por ações de MVC ou filtros de autorização.

O middleware de autenticação de portador do JWT também pode dar suporte a cenários mais avançados, como o uso de um certificado local para validar um token se a autoridade não estiver disponível. Para este cenário, você pode especificar um objeto `TokenValidationParameters` no objeto `JwtBearerOptions`.

## Recursos adicionais

- Compartilhando cookies entre aplicativos  
<https://docs.microsoft.com/aspnet/core/security/cookie-sharing>
- Introdução à identidade

<https://docs.microsoft.com/aspnet/core/security/authentication/identity>

- **Rick Anderson.** Autenticação de dois fatores com SMS  
<https://docs.microsoft.com/aspnet/core/security/authentication/2fa>
- **Habilitando a autenticação usando o Facebook, o Google e outros provedores externos**  
<https://docs.microsoft.com/aspnet/core/security/authentication/social/>
- **Michell Anicas.** Uma introdução ao OAuth 2  
<https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2>
- **AspNet.Security.OAuth.Providers** (repositório do GitHub para provedores do ASP.NET OAuth)  
<https://github.com/aspnet-contrib/AspNet.Security.OAuth.Providers/tree/dev/src>
- **IdentityServer4.** Documentação oficial  
<https://identityserver4.readthedocs.io/en/latest/>

[ANTERIOR](#)

[AVANÇAR](#)

# Sobre a autorização em aplicativos Web e em microsserviços .NET

09/04/2020 • 8 minutes to read • [Edit Online](#)

Após a autenticação, as APIs Web do ASP.NET Core precisam autorizar o acesso. Esse processo permite que um serviço disponibilize APIs para alguns usuários autenticados, mas não para todos. A [autorização](#) pode ser feita com base nas funções dos usuários ou com base em políticas personalizadas, que podem incluir a inspeção de sinistros ou outras heurísticas.

Restringir o acesso a uma rota mvc ASP.NET é tão fácil quanto aplicar um atributo Authorize ao método de ação (ou à classe do controlador se todas as ações do controlador exigirem autorização), como mostrado no exemplo a seguir:

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

Por padrão, adicionar um atributo Authorize sem parâmetros limitará o acesso aos usuários autenticados do controlador ou ação. Para restringir a disponibilidade de uma API apenas para usuários específicos, o atributo pode ser expandido para especificar funções necessárias ou políticas que os usuários devem atender.

## Implementar a autorização baseada em função

A identidade do ASP.NET Core tem um conceito interno de funções. Além dos usuários, a identidade do ASP.NET Core armazena informações sobre as diferentes funções usadas pelo aplicativo e controla quais usuários são atribuídos a quais funções. Essas atribuições podem ser alteradas de forma programática com o tipo `RoleManager`, que atualiza funções no armazenamento persistente, e o tipo `UserManager`, que pode conceder ou revogar as funções de usuários.

Se você estiver autenticando com tokens portadores JWT, o ASP.NET middleware de autenticação do portador Do Core JWT preencherá as funções de um usuário com base em alegações de função encontradas no token. Para limitar o acesso a uma ação ou um controlador MVC a usuários em funções específicas, você pode incluir um parâmetro de funções na anotação Authorize (atributo), como mostra o seguinte fragmento de código:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

Neste exemplo, apenas os usuários nas funções Administrator ou PowerUser podem acessar APIs no controlador ControlPanel (como executar a ação SetTime). A API ShutDown é mais restrita para permitir o acesso somente aos usuários na função Administrator.

Para exigir que um usuário esteja em várias funções, use vários atributos Authorize, conforme mostrado no exemplo a seguir:

```
[Authorize(Roles = "Administrator, PowerUser")]
[Authorize(Roles = "RemoteEmployee")]
[Authorize(Policy = "CustomPolicy")]
public ActionResult API1 ()
{
}
```

Neste exemplo, para chamar API1, um usuário deve:

- estar na função Administrator *ou* PowerUser
- estar na função RemoteEmployee *e*
- satisfazer a um manipulador personalizado da autorização CustomPolicy.

## Implementar a autorização baseada em política

Regras de autorização personalizadas também podem ser gravadas usando [políticas de autorização](#). Esta seção fornece uma visão geral. Para obter mais informações, confira o [Workshop de autorização do ASP.NET](#).

Políticas de autorização personalizadas são registradas no método Startup.ConfigureServices usando o método service.AddAuthorization. Esse método usa um delegado que configura um argumento AuthorizationOptions.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("AdministratorsOnly", policy =>
        policy.RequireRole("Administrator"));

    options.AddPolicy("EmployeesOnly", policy =>
        policy.RequireClaim("EmployeeNumber"));

    options.AddPolicy("Over21", policy =>
        policy.Requirements.Add(new MinimumAgeRequirement(21)));
});
```

Conforme mostrado no exemplo, políticas podem ser associadas a diferentes tipos de requisitos. Depois que as políticas são registradas, elas podem ser aplicadas a uma ação ou controlador, passando [Authorize(Policy="EmployeesOnly")] o nome da política como o argumento de Política do atributo Authorize (por exemplo, ). As políticas podem ter vários requisitos, não apenas um (como mostrado nestes exemplos).

No exemplo anterior, a primeira chamada `AddPolicy` é apenas uma maneira alternativa de autorizar pela função. Se `[Authorize(Policy="AdministratorsOnly")]` for aplicado a uma API, somente os usuários na função `Administrator` poderão acessá-lo.

A segunda chamada de `AddPolicy` demonstra uma maneira fácil de exigir que uma determinada declaração esteja presente para o usuário. O método `RequireClaim` também usa os valores esperados para a declaração. Se os valores são especificados, o requisito é atendido somente se o usuário tem uma declaração do tipo correto e um dos valores especificados. Se você estiver usando o middleware de autenticação de portador do JWT, todas as propriedades do JWT estarão disponíveis como declarações de usuário.

A política mais interessante mostrada aqui está no terceiro método `AddPolicy`, porque ele usa um requisito de autorização personalizado. Usando os requisitos de autorização personalizados, você pode ter grande controle sobre como a autorização é realizada. Para que isso funcione, você deve implementar esses tipos:

- Um tipo `Requirements` que é derivado de `IAuthorizationRequirement` e contém campos para especificar os detalhes do requisito. No exemplo, esse é um campo de idade para o tipo `MinimumAgeRequirement` de exemplo.
- Um manipulador que implementa `AuthorizationHandler<TRequirement>`, em que `T` é o tipo de `IAuthorizationRequirement` que o manipulador pode atender. O manipulador precisa implementar o método `HandleRequirementAsync`, que verifica se um contexto especificado que contém informações sobre o usuário atende ao requisito.

Se o usuário atender ao requisito, uma chamada para `context.Succeed` indicará que o usuário está autorizado. Se houver várias maneiras pelas quais um usuário pode satisfazer a um requisito de autorização, vários manipuladores poderão ser criados.

Além de registrar requisitos de política personalizada com chamadas de `AddPolicy`, você também precisa registrar manipuladores de requisito personalizados por meio da injeção de dependência (`services.AddTransient<IAuthorizationHandler, MinimumAgeHandler>()`).

Um exemplo de um requisito de autorização personalizado e manipulador para `DateOfBirth` verificar a idade de um usuário (com base em uma reclamação) está disponível na [documentação](#) de autorização do ASP.NET Core .

## Recursos adicionais

- Autenticação do Núcleo ASP.NET  
<https://docs.microsoft.com/aspnet/core/security/authentication/identity>
- autorização do núcleo ASP.NET  
<https://docs.microsoft.com/aspnet/core/security/authorization/introduction>
- Autorização baseada em papéis  
<https://docs.microsoft.com/aspnet/core/security/authorization/roles>
- Autorização personalizada baseada em políticas  
<https://docs.microsoft.com/aspnet/core/security/authorization/policies>

[PRÓXIMO](#)

[ANTERIOR](#)

# Armazenar segredos de aplicativo com segurança durante o desenvolvimento

18/03/2020 • 5 minutes to read • [Edit Online](#)

Para se conectar com recursos protegidos e outros serviços, os aplicativos ASP.NET Core normalmente precisam usar cadeias de conexão, senhas ou outras credenciais que contêm informações confidenciais. Essas informações confidenciais são chamadas *segredos*. Uma prática recomendada é não incluir segredos no código-fonte e não armazená-los no controle do código-fonte. Em vez disso, você deve usar o modelo de configuração do ASP.NET Core para ler os segredos de locais mais seguros.

Você precisa separar os segredos para acessar os recursos de desenvolvimento e de preparo daqueles que são usados para acessar recursos de produção, pois diferentes pessoas precisarão acessar esses diferentes conjuntos de segredos. Para armazenar segredos usados durante o desenvolvimento, as abordagens comuns são armazenar segredos em variáveis de ambiente ou por meio do uso da ferramenta Secret Manager do ASP.NET Core. Para obter mais armazenamento seguro em ambientes de produção, os microsserviços podem armazenar segredos em um Azure Key Vault.

## Armazenar segredos em variáveis de ambiente

Uma maneira de manter segredos fora do código-fonte é para os desenvolvedores definirem segredos baseados em cadeia de caracteres como [variáveis de ambiente](#) em seus computadores de desenvolvimento. Ao usar variáveis de ambiente para armazenar segredos com nomes hierárquicos, como aqueles aninhados em seções de configuração, você precisa nomear as variáveis para incluir a hierarquia completa de suas seções, delimitada por dois-pontos (:).

Por exemplo, a definição de uma variável de ambiente `Logging:LogLevel:Default` para o valor `Debug` seria equivalente a um valor de configuração do arquivo JSON a seguir:

```
{  
  "Logging": {  
    "LogLevel": {  
      "Default": "Debug"  
    }  
  }  
}
```

Para acessar esses valores de variáveis de ambiente, o aplicativo precisa apenas chamar `AddEnvironmentVariables` em seu `ConfigurationBuilder` ao construir um objeto `IConfigurationRoot`.

Observe que as variáveis de ambiente geralmente são armazenadas como texto sem formatação. Portanto, se o computador ou o processo com variáveis de ambiente estiver comprometido, os valores das variáveis de ambiente estarão visíveis.

## Armazenar segredos com o Secret Manager do ASP.NET Core

A ferramenta ASP.NET Core [Secret Manager](#) fornece outro método de manter segredos fora do código-fonte durante o [desenvolvimento](#). Para usar a ferramenta Secret Manager, instale o pacote `Microsoft.Extensions.Configuration.SecretManager` no arquivo de projeto. Quando essa dependência estiver presente e tiver sido restaurada, o comando `dotnet user-secrets` poderá ser usado para definir o valor dos segredos na linha de comando. Esses segredos serão armazenados em um arquivo JSON no diretório do perfil do

usuário (os detalhes variam de acordo com o sistema operacional), fora do código-fonte.

Os segredos definidos pela ferramenta Secret Manager são organizados pela propriedade `UserSecretsId` do projeto que está usando os segredos. Portanto, você precisa definir a propriedade `UserSecretsId` em seu arquivo de projeto, como mostra o snippet a seguir. O valor padrão é um GUID atribuído pelo Visual Studio, mas a cadeia de caracteres real não é importante, desde que seja exclusiva no seu computador.

```
<PropertyGroup>
    <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

O uso de segredos armazenados com o Secret Manager em um aplicativo é realizado chamando `AddUserSecrets<T>` na instância ConfigurationBuilder para incluir os segredos do aplicativo em sua configuração. O parâmetro genérico `T` deve ser um tipo do assembly ao qual o `UserSecretId` foi aplicado. Geralmente, basta usar o `AddUserSecrets<Startup>`.  
O `AddUserSecrets<Startup>()` está incluído nas opções padrão do ambiente de desenvolvimento quando o método `CreateDefaultBuilder` é usado em `Program.cs`.

[PRÓXIMO](#)

[ANTERIOR](#)

# Usar o Azure Key Vault para proteger os segredos no tempo de produção

18/03/2020 • 3 minutes to read • [Edit Online](#)

Os segredos armazenados como variáveis de ambiente ou armazenados pela ferramenta Secret Manager ainda são armazenados localmente e descriptografados no computador. Uma opção mais segura para armazenar segredos é o [Azure Key Vault](#), que oferece um local seguro e central para armazenar chaves e segredos.

O pacote [Microsoft.Extensions.Configuration.AzureKeyVault](#) permite que um aplicativo do ASP.NET Core leia informações de configuração do Azure Key Vault. Para começar a usar os segredos de um Azure Key Vault, siga estas etapas:

1. Registre seu aplicativo como um aplicativo do Azure AD. (O acesso aos cofres-chave é gerenciado pelo Azure AD.) Isso pode ser feito através do portal de gerenciamento do Azure.\

Ou, se você desejar que seu aplicativo seja autenticado usando um certificado em vez de um segredo do cliente ou senha, use o cmdlet do PowerShell [New-AzADApplication](#). O certificado que você registrar com o Azure Key Vault precisa apenas de sua chave pública. O aplicativo usará a chave privada.

2. Permita que o aplicativo registrado acesse o Key Vault criando uma entidade de serviço. É possível fazer isso usando os seguintes comandos do PowerShell:

```
$sp = New-AzADServicePrincipal -ApplicationId "<Application ID guid>"  
Set-AzKeyVaultAccessPolicy -VaultName "<VaultName>" -ServicePrincipalName $sp.ServicePrincipalNames[0]  
-PermissionsToSecrets all -ResourceGroupName "<KeyVault Resource Group>"
```

3. Inclua o Key Vault como uma fonte de configuração em seu aplicativo chamando o método de extensão [AzureKeyVaultConfigurationExtensions.AddAzureKeyVault](#) ao criar uma instância de [IConfigurationRoot](#). Observe que a chamada de `AddAzureKeyVault` requer a ID do aplicativo que foi registrada e permitiu acesso ao Key Vault nas etapas anteriores.

Você também pode usar uma sobrecarga de `AddAzureKeyVault`, que usa um certificado no lugar do segredo do cliente, apenas incluindo uma referência ao pacote [Microsoft.IdentityModel.Clients.ActiveDirectory](#).

## IMPORTANT

Recomendamos que você registre o Azure Key Vault como o último provedor de configuração, para que ele possa substituir os valores de configuração de provedores anteriores.

## Recursos adicionais

- **Usando o Azure Key Vault para proteger segredos de aplicativos**  
<https://docs.microsoft.com/azure/guidance/guidance-multitenant-identity-keyvault>
- **Armazenamento seguro de segredos de aplicativos durante o desenvolvimento**  
<https://docs.microsoft.com/aspnet/core/security/app-secrets>
- **Configuração da proteção de dados**  
<https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/overview>
- **Gerenciamento de chaves de proteção de dados e vida útil em ASP.NET Core**

<https://docs.microsoft.com/aspnet/core/security/data-protection/configuration/default-settings>

- Repositório do GitHub **Microsoft.Extensions.Configuration.KeyPerFile**.

<https://github.com/dotnet/extensions/tree/master/src/Configuration/Config.KeyPerFile>

[PRÓXIMO](#)

[ANTERIOR](#)

# Principais observações

09/04/2020 • 8 minutes to read • [Edit Online](#)

A seguir, encontram-se as conclusões mais importantes deste guia, que servem como um resumo dos principais aspectos a serem lembrados.

**Benefícios do uso de contêineres.** As soluções baseadas em contêiner oferecem uma importante economia de custo porque ajudam a reduzir os problemas de implantação causados por dependências com falha em ambientes de produção. Portanto, os contêineres melhoraram significativamente as operações de produção e DevOps.

**Contêineres serão onipresentes.** Os contêineres baseados no Docker estão se tornando o verdadeiro padrão no setor, com suporte dos principais fornecedores nos ecossistemas do Windows e do Linux, como Microsoft, Amazon AWS, Google e IBM. Provavelmente, em breve o Docker será predominante nos data centers de nuvem e locais.

**Contêineres como uma unidade de implantação.** Um contêiner do Docker está se tornando a unidade padrão de implantação para qualquer serviço ou aplicativo baseado em servidor.

**Microserviços.** A arquitetura de microserviços está se tornando a abordagem preferencial para aplicativos críticos grandes ou complexos e distribuídos baseados em diversos subsistemas independentes na forma de serviços autônomos. Em uma arquitetura baseada em microserviço, o aplicativo é criado em uma coleção de serviços que podem ser desenvolvidos, testados, implantados e ter as versões controladas de forma independente. Cada serviço pode incluir qualquer banco de dados autônomo relacionado.

**Design controlado por domínio e SOA.** Os padrões de arquitetura de microserviços derivam da SOA (arquitetura orientada a serviços) e do DDD (design controlado por domínio). Ao projetar e desenvolver microserviços para ambientes com necessidades e regras dos negócios em evolução, é importante considerar as abordagens e os padrões de DDD.

**Desafios dos microserviços.** Os microserviços oferecem muitos recursos avançados, como implantação independente, fortes limites de subsistema e diversidade tecnológica. No entanto, eles também apresentam vários novos desafios relacionados com o desenvolvimento de aplicativos distribuídos, como modelos de dados fragmentados e independentes, comunicação resiliente entre microserviços, consistência eventual e complexidade operacional como resultado da combinação de registros em log e monitoramento de informações de diversos microserviços. Esses aspectos geram um nível de complexidade muito maior do que um aplicativo monolítico tradicional. Assim, apenas cenários específicos são adequados para aplicativos baseados em microserviço. Isso inclui aplicativos grandes e complexos com vários subsistemas em evolução. Nesses casos, vale a pena investir em uma arquitetura de software mais complexa, pois ela fornecerá uma agilidade e uma manutenção de aplicativos melhores a longo prazo.

**Contêineres para qualquer aplicativo.** Os contêineres são convenientes para os microserviços, mas também podem ser úteis para aplicativos monolíticos baseados no .NET Framework tradicional, quando são usados contêineres do Windows. Os benefícios de usar o Docker, como resolver vários problemas entre as fases de implantação e produção e fornecer ambientes de desenvolvimento e teste de última geração, aplicam-se a vários tipos de aplicativos diferentes.

**CLI versus IDE.** Com as ferramentas da Microsoft, é possível desenvolver aplicativos .NET em contêineres usando sua abordagem preferencial. Você pode desenvolver com uma CLI e um ambiente baseado em editor usando a CLI do Docker e o Visual Studio Code. Outra opção é usar uma abordagem centrada em IDE com o Visual Studio e seus recursos exclusivos para Docker, como a depuração de aplicativos de vários contêineres.

**Aplicativos de nuvem resilientes.** Em sistemas baseados em nuvem e sistemas distribuídos em geral, há

sempre o risco de falha parcial. Como clientes e serviços são processos separados (contêineres), um serviço pode não ser capaz de responder em tempo hárum à solicitação de um cliente. Por exemplo, um serviço pode estar inativo por causa de uma falha parcial ou de manutenção, estar sobrecarregado e respondendo a solicitações de maneira extremamente lenta ou simplesmente não estar acessível durante um curto período devido a problemas de rede. Portanto, um aplicativo baseado em nuvem deve adotar essas falhas e ter uma estratégia para responder a elas. Essas estratégias podem incluir políticas de repetição (reenviar mensagens ou repetir solicitações) e padrões de implementação disjuntor para evitar uma carga exponencial de solicitações repetidas. Basicamente, os aplicativos baseados em nuvem precisam ter mecanismos resilientes, seja com base na infraestrutura de nuvem ou personalizados, como os de alto nível fornecidos pelos orquestradores ou barramentos de serviço.

**Segurança.** O mundo moderno de contêineres e microsserviços pode expor novas vulnerabilidades. Há várias maneiras de implementar a segurança básica de aplicativo, com base em autenticação e autorização. No entanto, a segurança do contêiner precisa considerar os principais componentes adicionais que resultam em aplicativos inherentemente mais seguros. Um elemento crítico da criação de aplicativos mais seguros é implantar uma maneira segura de comunicar-se com outros aplicativos e sistemas, algo que geralmente requer credenciais, tokens, senhas e assim por diante, conhecidos como segredos do aplicativo. Toda solução de segurança precisa seguir as práticas recomendadas de segurança, como criptografia de segredos quando em trânsito e em repouso, e impedir que os segredos vazem ao serem consumidos pelo aplicativo final. Esses segredos precisam ser armazenados e mantidos em segurança, como quando o Azure Key Vault é usado.

**Orquestradores.** Os orquestradores baseados em contêiner, como o Serviço de Kubernetes do Azure e o Azure Service Fabric são uma parte fundamental de qualquer microsserviço e aplicativo baseado em contêiner significativo. Esses aplicativos apresentam alta complexidade, necessidades de escalabilidade e constante evolução. Este guia apresentou os orquestradores e sua função em soluções baseadas em microsserviços e contêineres. Se suas necessidades de aplicativo estão exigindo aplicativos complexos em contêineres, pode ser útil procurar outros recursos para saber mais sobre os orquestradores.

[ANTERIOR](#)