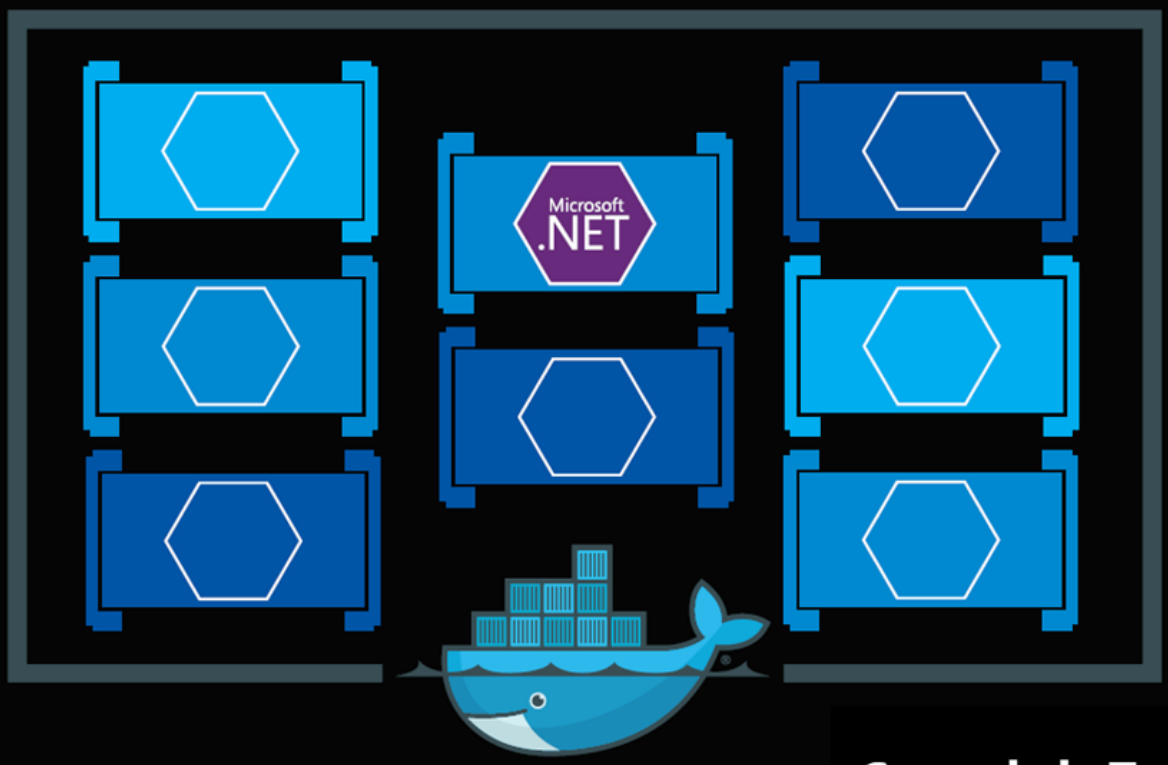
Microsoft

.NET Microservices: Architecture for Containerized .NET Applications



Cesar de la Torre
Bill Wagner
Mike Rousos

Microsoft Corporation

EDITION v7.0 - Updated to ASP.NET Core 7.0

Refer changelog for the book updates and community contributions.

This guide is an introduction to developing microservices-based applications and managing them using containers. It discusses architectural design and implementation approaches using .NET and Docker containers.

To make it easier to get started, the guide focuses on a reference containerized and microservice-based application that you can explore. The reference application is available at the eShopOnContainers GitHub repo.

Action links

- This e-book is also available in a PDF format (English version only) Download
- Clone/Fork the reference application eShopOnContainers on GitHub
- Watch the introductory video
- Get to know the Microservices Architecture right away

Introduction

Enterprises are increasingly realizing cost savings, solving deployment problems, and improving DevOps and production operations by using containers. Microsoft has been releasing container innovations for Windows and Linux by creating products like Azure Kubernetes Service and Azure Service Fabric, and by partnering with industry leaders like Docker, Mesosphere, and Kubernetes. These products deliver container solutions that help companies build and deploy applications at cloud speed and scale, whatever their choice of platform or tools.

Docker is becoming the de facto standard in the container industry, supported by the most significant vendors in the Windows and Linux ecosystems. (Microsoft is one of the main cloud vendors supporting Docker). In the future, Docker will probably be ubiquitous in any datacenter in the cloud or on-premises.

In addition, the microservices architecture is emerging as an important approach for distributed mission-critical applications. In a microservice-based architecture, the application is built on a collection of services that can be developed, tested, deployed, and versioned independently.

About this guide

This guide is an introduction to developing microservices-based applications and managing them using containers. It discusses architectural design and implementation approaches using .NET and Docker containers. To make it easier to get started with containers and microservices, the guide focuses on a reference containerized and microservice-based application that you can explore. The sample application is available at the eShopOnContainers GitHub repo.

This guide provides foundational development and architectural guidance primarily at a development environment level with a focus on two technologies: Docker and .NET. Our intention is that you read this guide when thinking about your application design without focusing on the infrastructure (cloud or on-premises) of your production environment. You will make decisions about your infrastructure later, when you create your production-ready applications. Therefore, this guide is intended to be infrastructure agnostic and more development-environment-centric.

After you have studied this guide, your next step would be to learn about production-ready microservices on Microsoft Azure.

Version

This guide has been revised to cover **.NET 7** version along with many additional updates related to the same "wave" of technologies (that is, Azure and additional third-party technologies) coinciding in time with the .NET 7 release. That's why the book version has also been updated to version **7.0**.

What this guide does not cover

This guide does not focus on the application lifecycle, DevOps, CI/CD pipelines, or team work. The complementary guide Containerized Docker Application Lifecycle with Microsoft Platform and Tools focuses on that subject. The current guide also does not provide implementation details on Azure infrastructure, such as information on specific orchestrators.

Additional resources

- Containerized Docker Application Lifecycle with Microsoft Platform and Tools (downloadable e-book)
  https://aka.ms/dockerlifecycleebook

Who should use this guide

We wrote this guide for developers and solution architects who are new to Docker-based application development and to microservices-based architecture. This guide is for you if you want to learn how to architect, design, and implement proof-of-concept applications with Microsoft development technologies (with special focus on .NET) and with Docker containers.

You will also find this guide useful if you are a technical decision maker, such as an enterprise architect, who wants an architecture and technology overview before you decide on what approach to select for new and modern distributed applications.

How to use this guide

The first part of this guide introduces Docker containers, discusses how to choose between .NET 7 and the .NET Framework as a development framework, and provides an overview of microservices. This content is for architects and technical decision makers who want an overview but don't need to focus on code implementation details.

The second part of the guide starts with the Development process for Docker based applications section. It focuses on the development and microservice patterns for implementing applications using .NET and Docker. This section will be of most interest to developers and architects who want to focus on code and on patterns and implementation details.

Related microservice and container-based reference application: eShopOnContainers

The eShopOnContainers application is an open-source reference app for .NET and microservices that is designed to be deployed using Docker containers. The application consists of multiple subsystems, including several e-store UI front-ends (a Web MVC app, a Web SPA, and a native mobile app). It also includes the back-end microservices and containers for all required server-side operations.

The purpose of the application is to showcase architectural patterns. **IT IS NOT A PRODUCTION-READY TEMPLATE** to start real-world applications. In fact, the application is in a permanent beta state, as it's also used to test new potentially interesting technologies as they show up.

Credits

Co-Authors:

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

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

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

Editors:

Mike Pope

Steve Hoag

Participants and reviewers:

Jeffrey Richter, Partner Software Eng, Azure team, Microsoft

Jimmy Bogard, Chief Architect at Headspring

Udi Dahan, Founder & CEO, Particular Software

Jimmy Nilsson, Co-founder and CEO of Factor10

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

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

Diego Vega, PM Lead, Entity Framework team, Microsoft

Barry Dorrans, Sr. Security Program Manager

Rowan Miller, Sr. Program Manager, Microsoft

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

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

Nish Anil, Sr. Program Manager, .NET team, Microsoft

Dylan Reisenberger, Architect and Dev Lead at Polly

Steve "ardalis" Smith - Software Architect and Trainer - Ardalis.com

Ian Cooper, Coding Architect at Brighter

Unai Zorrilla, Architect and Dev Lead at Plain Concepts

Eduard Tomas, Dev Lead at Plain Concepts

Ramon Tomas, Developer at Plain Concepts

David Sanz, Developer at Plain Concepts

Javier Valero, Chief Operating Officer at Grupo Solutio

Pierre Millet, Sr. Consultant, Microsoft

Michael Friis, Product Manager, Docker Inc

Charles Lowell, Software Engineer, VS CAT team, Microsoft

Miguel Veloso, Software Development Engineer at Plain Concepts

Sumit Ghosh, Principal Consultant at Neudesic

Copyright

PUBLISHED BY

Microsoft Developer Division, .NET and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2023 by Microsoft Corporation

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

This book is provided "as-is" and expresses the author's views and opinions. The views, opinions and information expressed in this book, including URL and other Internet website references, may change without notice.

Some examples depicted herein are provided for illustration only and are fictitious. No real association or connection is intended or should be inferred.

Microsoft and the trademarks listed at https://www.microsoft.com on the “Trademarks” webpage are trademarks of the Microsoft group of companies.

Mac and macOS are trademarks of Apple Inc.

The Docker whale logo is a registered trademark of Docker, Inc. Used by permission.

All other marks and logos are property of their respective owners.

Contents

Introduction to Containers and Docker 1

What is Docker? 2

Comparing Docker containers with virtual machines 3

A simple analogy 4

Docker terminology 5

Docker containers, images, and registries 7

Choosing Between .NET and .NET Framework for Docker Containers 9

General guidance 9

When to choose .NET for Docker containers 10

Developing and deploying cross platform 10

Using containers for new ("green-field") projects 11

Create and deploy microservices on containers 11

Deploying high density in scalable systems 11

When to choose .NET Framework for Docker containers 12

Migrating existing applications directly to a Windows Server container 12

Using third-party .NET libraries or NuGet packages not available for .NET 7 12

Using .NET technologies not available for .NET 7 12

Using a platform or API that doesn't support .NET 7 13

Porting existing ASP.NET application to .NET 7 13

Decision table: .NET implementations to use for Docker 13

What OS to target with .NET containers 14

Official .NET Docker images 16

.NET and Docker image optimizations for development versus production 16

Architecting container and microservice-based applications 18

Container design principles 18

Containerizing monolithic applications 19

Deploying a monolithic application as a container 21

Publishing a single-container-based application to Azure App Service 21

Manage state and data in Docker applications 22
Service-oriented architecture 25
Microservices architecture 25
Additional resources 27
Data sovereignty per microservice 27
The relationship between microservices and the Bounded Context pattern 29
Logical architecture versus physical architecture 30
Challenges and solutions for distributed data management 31
Challenge #1: How to define the boundaries of each microservice 31
Challenge #2: How to create queries that retrieve data from several microservices 32
Challenge #3: How to achieve consistency across multiple microservices 33
Challenge #4: How to design communication across microservice boundaries 35
Additional resources 36
Identify domain-model boundaries for each microservice 36
The API gateway pattern versus the Direct client-to-microservice communication 40
Direct client-to-microservice communication 40
Why consider API Gateways instead of direct client-to-microservice communication 41
What is the API Gateway pattern? 42
Main features in the API Gateway pattern 44
Using products with API Gateway features 45
Drawbacks of the API Gateway pattern 47
Additional resources 48
Communication in a microservice architecture 48
Communication types 49
Asynchronous microservice integration enforces microservice's autonomy 50
Communication styles 52
Asynchronous message-based communication 54
Single-receiver message-based communication 55
Multiple-receivers message-based communication 56
Asynchronous event-driven communication 56
A note about messaging technologies for production systems 57
Resiliently publishing to the event bus 58

Additional resources ... 58
Creating, evolving, and versioning microservice APIs and contracts ... 59
Additional resources ... 59
Microservices addressability and the service registry ... 60
Additional resources ... 60
Creating composite UI based on microservices ... 60
Additional resources ... 62
Resiliency and high availability in microservices ... 63
Health management and diagnostics in microservices ... 63
Additional resources ... 65
Orchestrate microservices and multi-container applications for high scalability and availability ... 66
Software platforms for container clustering, orchestration, and scheduling ... 68
Using container-based orchestrators in Microsoft Azure ... 68
Using Azure Kubernetes Service ... 69
Development environment for Kubernetes ... 70
Getting started with Azure Kubernetes Service (AKS) ... 70
Deploy with Helm charts into Kubernetes clusters ... 71
Additional resources ... 71
Development process for Docker-based applications ... 72
Development environment for Docker apps ... 72
Development tool choices: IDE or editor ... 72
Additional resources ... 73
.NET languages and frameworks for Docker containers ... 73
Development workflow for Docker apps ... 73
Workflow for developing Docker container-based applications ... 73
Step 1. Start coding and create your initial application or service baseline ... 75
Step 2. Create a Dockerfile related to an existing .NET base image ... 76
Step 3. Create your custom Docker images and embed your application or service in them ... 83
Step 4. Define your services in docker-compose.yml when building a multi-container Docker application ... 84
Step 5. Build and run your Docker application ... 87
Step 6. Test your Docker application using your local Docker host ... 89

Simplified workflow when developing containers with Visual Studio 90
Using PowerShell commands in a Dockerfile to set up Windows Containers 91
Designing and Developing Multi-Container and Microservice-Based .NET Applications 93
Design a microservice-oriented application 93
Application specifications 93
Development team context 94
Choosing an architecture 94
Benefits of a microservice-based solution 97
Downsides of a microservice-based solution 98
External versus internal architecture and design patterns 99
The new world: multiple architectural patterns and polyglot microservices 100
Creating a simple data-driven CRUD microservice 102
Designing a simple CRUD microservice 102
Implementing a simple CRUD microservice with ASP.NET Core 103
The DB connection string and environment variables used by Docker containers 109
Generating Swagger description metadata from your ASP.NET Core Web API 111
Defining your multi-container application with docker-compose.yml 116
Use a database server running as a container 127
SQL Server running as a container with a microservice-related database 128
Seeding with test data on Web application startup 129
EF Core InMemory database versus SQL Server running as a container 132
Using a Redis cache service running in a container 132
Implementing event-based communication between microservices (integration events) 133
Using message brokers and service buses for production systems 134
Integration events 135
The event bus 136
Additional resources 138
Implementing an event bus with RabbitMQ for the development or test environment 138
Implementing a simple publish method with RabbitMQ 139
Implementing the subscription code with the RabbitMQ API 140
Additional resources 141

Subscribing to events 141
Publishing events through the event bus 142
Idempotency in update message events 149
Deduplicating integration event messages 150
Testing ASP.NET Core services and web apps 152
Testing in eShopOnContainers 155
Implement background tasks in microservices with IHostedService and the BackgroundService class 157
Registering hosted services in your WebHost or Host 159
The IHostedService interface 159
Implementing IHostedService with a custom hosted service class deriving from the BackgroundService base class 160
Additional resources 163
Implement API Gateways with Ocelot 163
Architect and design your API Gateways 163
Implementing your API Gateways with Ocelot 168
Using Kubernetes Ingress plus Ocelot API Gateways 180
Additional cross-cutting features in an Ocelot API Gateway 181
Tackle Business Complexity in a Microservice with DDD and CQRS Patterns 182
Apply simplified CQRS and DDD patterns in a microservice 184
Additional resources 186
Apply CQRS and CQS approaches in a DDD microservice in eShopOnContainers 186
CQRS and DDD patterns are not top-level architectures 187
Implement reads/queries in a CQRS microservice 188
Use ViewModels specifically made for client apps, independent from domain model constraints 189
Use Dapper as a micro ORM to perform queries 189
Dynamic versus static ViewModels 190
Additional resources 193
Design a DDD-oriented microservice 194
Keep the microservice context boundaries relatively small 194
Layers in DDD microservices 195

Design a microservice domain model 199
The Domain Entity pattern 199
Implement a microservice domain model with .NET 204
Domain model structure in a custom .NET Standard Library 204
Structure aggregates in a custom .NET Standard library 205
Implement domain entities as POCO classes 206
Encapsulate data in the Domain Entities 207
Seedwork (reusable base classes and interfaces for your domain model) 210
The custom Entity base class 211
Repository contracts (interfaces) in the domain model layer 212
Additional resources 213
Implement value objects 213
Important characteristics of value objects 214
Value object implementation in C# 215
How to persist value objects in the database with EF Core 2.0 and later 217
Persist value objects as owned entity types in EF Core 2.0 and later 218
Additional resources 221
Use enumeration classes instead of enum types 221
Implement an Enumeration base class 222
Additional resources 223
Design validations in the domain model layer 223
Implement validations in the domain model layer 224
Additional resources 225
Client-side validation (validation in the presentation layers) 226
Additional resources 227
Domain events: Design and implementation 227
What is a domain event? 228
Domain events versus integration events 228
Domain events as a preferred way to trigger side effects across multiple aggregates within the same domain 229
Implement domain events 231
Conclusions on domain events 237

Additional resources 238
Design the infrastructure persistence layer 238
The Repository pattern 238
Additional resources 243
Implement the infrastructure persistence layer with Entity Framework Core 243
Introduction to Entity Framework Core 244
Infrastructure in Entity Framework Core from a DDD perspective 244
Implement custom repositories with Entity Framework Core 246
EF DbContext and IUnitOfWork instance lifetime in your IoC container 248
The repository instance lifetime in your IoC container 249
Table mapping 250
Implement the Query Specification pattern 253
Use NoSQL databases as a persistence infrastructure 255
Introduction to Azure Cosmos DB and the native Cosmos DB API 256
Implement .NET code targeting MongoDB and Azure Cosmos DB 258
Design the microservice application layer and Web API 266
Use SOLID principles and Dependency Injection 266
Implement the microservice application layer using the Web API 267
Use Dependency Injection to inject infrastructure objects into your application layer 267
Implement the Command and Command Handler patterns 271
The Command process pipeline: how to trigger a command handler 278
Implement the command process pipeline with a mediator pattern (MediatR) 281
Apply cross-cutting concerns when processing commands with the Behaviors in MediatR 287
Implement resilient applications 291
Handle partial failure 292
Strategies to handle partial failure 294
Additional resources 295
Implement retries with exponential backoff 295
Implement resilient Entity Framework Core SQL connections 295
Execution strategies and explicit transactions using BeginTransaction and multiple DbContexts 296
Additional resources 298
Use IHttpClientFactory to implement resilient HTTP requests 298

Issues with the original HttpClient class available in .NET 298
Benefits of using IHttpClientFactory 299
Multiple ways to use IHttpClientFactory 300
How to use Typed Clients with IHttpClientFactory 300
Additional resources 304
Implement HTTP call retries with exponential backoff with IHttpClientFactory and Polly policies 304
Add a jitter strategy to the retry policy 305
Additional resources 306
Implement the Circuit Breaker pattern 306
Implement Circuit Breaker pattern with IHttpClientFactory and Polly 307
Test Http retries and circuit breakers in eShopOnContainers 308
Additional resources 310
Health monitoring 310
Implement health checks in ASP.NET Core services 311
Use watchdogs 315
Health checks when using orchestrators 317
Advanced monitoring: visualization, analysis, and alerts 317
Additional resources 318
Make secure .NET Microservices and Web Applications 319
Implement authentication in .NET microservices and web applications 319
Authenticate with ASP.NET Core Identity 320
Authenticate with external providers 321
Authenticate with bearer tokens 323
Authenticate with an OpenID Connect or OAuth 2.0 Identity provider 324
Issue security tokens from an ASP.NET Core service 325
Consume security tokens 326
Additional resources 327
About authorization in .NET microservices and web applications 327
Implement role-based authorization 328
Implement policy-based authorization 329
Authorization and minimal apis 330
Additional resources 330

Store application secrets safely during development 330
Store secrets in environment variables 331
Store secrets with the ASP.NET Core Secret Manager 331
Use Azure Key Vault to protect secrets at production time 332
Additional resources 333
.NET Microservices Architecture key takeaways 334

Introduction to Containers and Docker

Containerization is an approach to software development in which an application or service, its dependencies, and its configuration (abstracted as deployment manifest files) are packaged together as a container image. The containerized application can be tested as a unit and deployed as a container image instance to the host operating system (OS).

Just as shipping containers allow goods to be transported by ship, train, or truck regardless of the cargo inside, software containers act as a standard unit of software deployment that can contain different code and dependencies. Containerizing software this way enables developers and IT professionals to deploy them across environments with little or no modification.

Containers also isolate applications from each other on a shared OS. Containerized applications run on top of a container host that in turn runs on the OS (Linux or Windows). Containers therefore have a significantly smaller footprint than virtual machine (VM) images.

Each container can run a whole web application or a service, as shown in Figure 2-1. In this example, Docker host is a container host, and App1, App2, Svc 1, and Svc 2 are containerized applications or services.

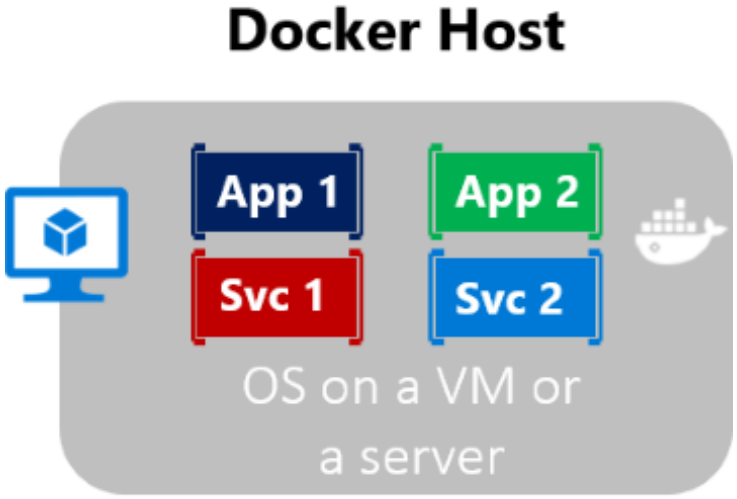


Figure 2-1. Multiple containers running on a container host

Another benefit of containerization is scalability. You can scale out quickly by creating new containers for short-term tasks. From an application point of view, instantiating an image (creating a container) is similar to instantiating a process like a service or a web app. For reliability, however, when you run multiple instances of the same image across multiple host servers, you typically want each container (image instance) to run in a different host server or VM in different fault domains.

In short, containers offer the benefits of isolation, portability, agility, scalability, and control across the whole application lifecycle workflow. The most important benefit is the environment's isolation provided between Dev and Ops.

What is Docker?

Docker is an open-source project for automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises. Docker is also a company that promotes and evolves this technology, working in collaboration with cloud, Linux, and Windows vendors, including Microsoft.

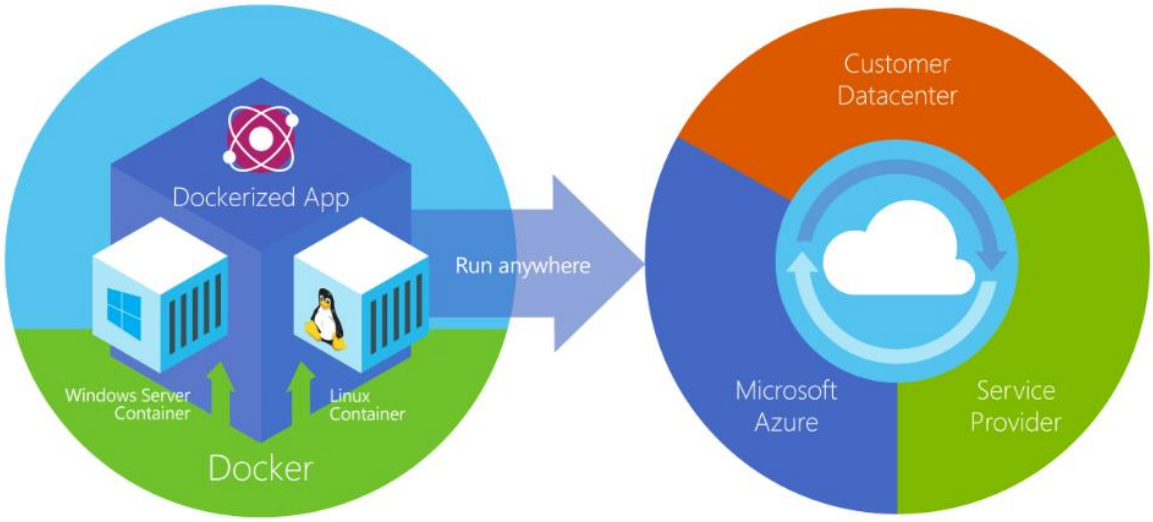


Figure 2-2. Docker deploys containers at all layers of the hybrid cloud.

Docker containers can run anywhere, on-premises in the customer datacenter, in an external service provider or in the cloud, on Azure. Docker image containers can run natively on Linux and Windows. However, Windows images can run only on Windows hosts and Linux images can run on Linux hosts and Windows hosts (using a Hyper-V Linux VM, so far), where host means a server or a VM.

Developers can use development environments on Windows, Linux, or macOS. On the development computer, the developer runs a Docker host where Docker images are deployed, including the app and its dependencies. Developers who work on Linux or on macOS use a Docker host that is Linux based, and they can create images only for Linux containers. (Developers working on macOS can edit code or run the Docker CLI from macOS, but as of the time of this writing, containers don't run

directly on macOS.) Developers who work on Windows can create images for either Linux or Windows Containers.

To host containers in development environments and provide additional developer tools, Docker ships Docker Desktop for Windows or for macOS. These products install the necessary VM (the Docker host) to host the containers.

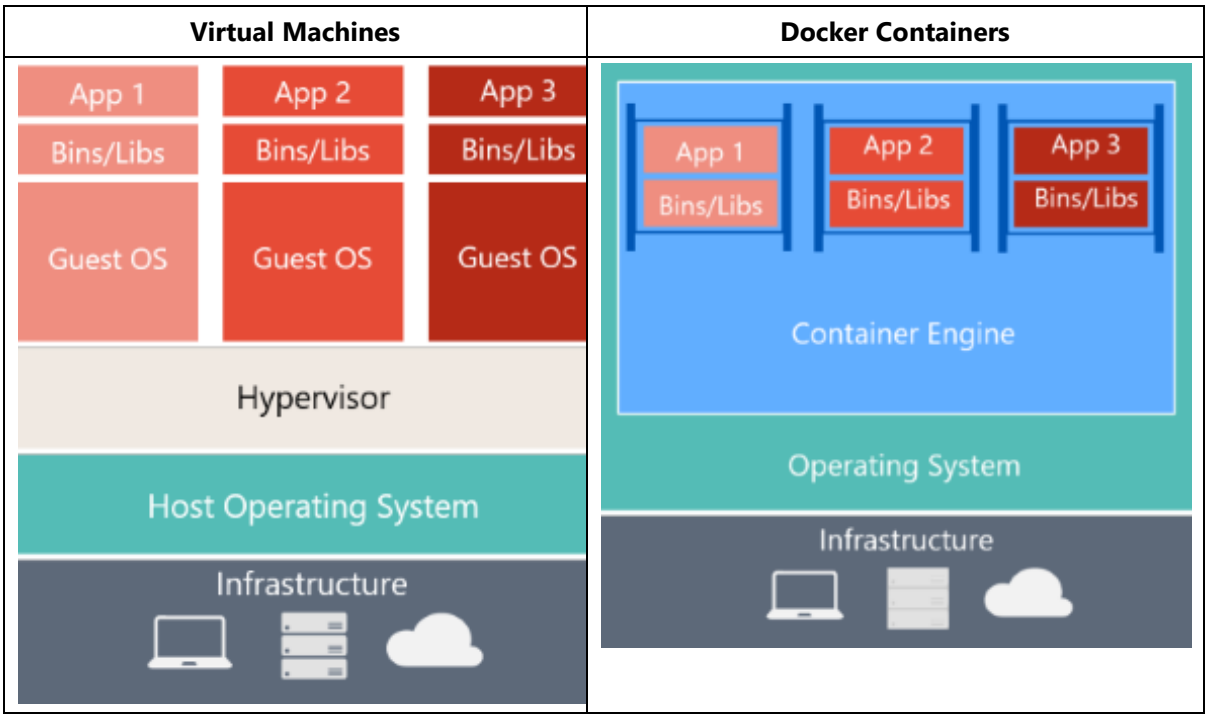
To run Windows Containers, there are two types of runtimes:

- Windows Server Containers provide application isolation through process and namespace isolation technology. A Windows Server Container shares a kernel with the container host and with all containers running on the host.
- Hyper-V Containers expand on the isolation provided by Windows Server Containers by running each container in a highly optimized virtual machine. In this configuration, the kernel of the container host isn't shared with the Hyper-V Containers, providing better isolation.

The images for these containers are created the same way and function the same. The difference is in how the container is created from the image running a Hyper-V Container requires an extra parameter. For details, see Hyper-V Containers.

Comparing Docker containers with virtual machines

Figure 2-3 shows a comparison between VMs and Docker containers.



Virtual Machines	Docker Containers
Virtual machines include the application, the required libraries or binaries, and a full guest operating system. Full virtualization requires more resources than containerization.	Containers include the application and all its dependencies. However, they share the OS kernel with other containers, running as isolated processes in user space on the host operating system. (Except in Hyper-V containers, where each container runs inside of a special virtual machine per container.)

Figure 2-3. Comparison of traditional virtual machines to Docker containers

For VMs, there are three base layers in the host server, from the bottom-up: infrastructure, Host Operating System and a Hypervisor and on top of all that each VM has its own OS and all necessary libraries. For Docker, the host server only has the infrastructure and the OS and on top of that, the container engine, that keeps container isolated but sharing the base OS services.

Because containers require far fewer resources (for example, they don't need a full OS), they're easy to deploy and they start fast. This allows you to have higher density, meaning that it allows you to run more services on the same hardware unit, thereby reducing costs.

As a side effect of running on the same kernel, you get less isolation than VMs.

The main goal of an image is that it makes the environment (dependencies) the same across different deployments. This means that you can debug it on your machine and then deploy it to another machine with the same environment guaranteed.

A container image is a way to package an app or service and deploy it in a reliable and reproducible way. You could say that Docker isn't only a technology but also a philosophy and a process.

When using Docker, you won't hear developers say, "It works on my machine, why not in production?" They can simply say, "It runs on Docker", because the packaged Docker application can be executed on any supported Docker environment, and it runs the way it was intended to on all deployment targets (such as Dev, QA, staging, and production).

A simple analogy

Perhaps a simple analogy can help getting the grasp of the core concept of Docker.

Let's go back in time to the 1950s for a moment. There were no word processors, and the photocopiers were used everywhere (kind of).

Imagine you're responsible for quickly issuing batches of letters as required, to mail them to customers, using real paper and envelopes, to be delivered physically to each customer's address (there was no email back then).

At some point, you realize the letters are just a composition of a large set of paragraphs, which are picked and arranged as needed, according to the purpose of the letter, so you devise a system to issue letters quickly, expecting to get a hefty raise.

The system is simple:

1. You begin with a deck of transparent sheets containing one paragraph each.

2. To issue a set of letters, you pick the sheets with the paragraphs you need, then you stack and align them so they look and read fine.
3. Finally, you place the set in the photocopier and press start to produce as many letters as required.

So, simplifying, that's the core idea of Docker.

In Docker, each layer is the resulting set of changes that happen to the filesystem after executing a command, such as, installing a program.

So, when you "look" at the filesystem after the layer has been copied, you see all the files, included in the layer when the program was installed.

You can think of an image as an auxiliary read-only hard disk ready to be installed in a "computer" where the operating system is already installed.

Similarly, you can think of a container as the "computer" with the image hard disk installed. The container, just like a computer, can be powered on or off.

Docker terminology

This section lists terms and definitions you should be familiar with before getting deeper into Docker. For further definitions, see the extensive glossary provided by Docker.

Container image: A package with all the dependencies and information needed to create a container. An image includes all the dependencies (such as frameworks) plus deployment and execution configuration to be used by a container runtime. Usually, an image derives from multiple base images that are layers stacked on top of each other to form the container's filesystem. An image is immutable once it has been created.

Dockerfile: A text file that contains instructions for building a Docker image. It's like a batch script, the first line states the base image to begin with and then follow the instructions to install required programs, copy files, and so on, until you get the working environment you need.

Build: The action of building a container image based on the information and context provided by its Dockerfile, plus additional files in the folder where the image is built. You can build images with the following Docker command:

```
docker build
```

Container: An instance of a Docker image. A container represents the execution of a single application, process, or service. It consists of the contents of a Docker image, an execution environment, and a standard set of instructions. When scaling a service, you create multiple instances of a container from the same image. Or a batch job can create multiple containers from the same image, passing different parameters to each instance.

Volumes: Offer a writable filesystem that the container can use. Since images are read-only but most programs need to write to the filesystem, volumes add a writable layer, on top of the container image, so the programs have access to a writable filesystem. The program doesn't know it's accessing a

layered filesystem, it's just the filesystem as usual. Volumes live in the host system and are managed by Docker.

Tag: A mark or label you can apply to images so that different images or versions of the same image (depending on the version number or the target environment) can be identified.

Multi-stage Build: Is a feature, since Docker 17.05 or higher, that helps to reduce the size of the final images. For example, a large base image, containing the SDK can be used for compiling and publishing and then a small runtime-only base image can be used to host the application.

Repository (repo): A collection of related Docker images, labeled with a tag that indicates the image version. Some repos contain multiple variants of a specific image, such as an image containing SDKs (heavier), an image containing only runtimes (lighter), etc. Those variants can be marked with tags. A single repo can contain platform variants, such as a Linux image and a Windows image.

Registry: A service that provides access to repositories. The default registry for most public images is Docker Hub (owned by Docker as an organization). A registry usually contains repositories from multiple teams. Companies often have private registries to store and manage images they've created. Azure Container Registry is another example.

Multi-arch image: For multi-architecture (or multi-platform), it's a Docker feature that simplifies the selection of the appropriate image, according to the platform where Docker is running. For example, when a Dockerfile requests a base image **FROM mcr.microsoft.com/dotnet/sdk:7.0** from the registry, it actually gets **7.0-nanoserver-ltsc2022**, **7.0-nanoserver-1809** or **7.0-bullseye-slim**, depending on the operating system and version where Docker is running.

Docker Hub: A public registry to upload images and work with them. Docker Hub provides Docker image hosting, public or private registries, build triggers and web hooks, and integration with GitHub and Bitbucket.

Azure Container Registry: A public resource for working with Docker images and its components in Azure. This provides a registry that's close to your deployments in Azure and that gives you control over access, making it possible to use your Azure Active Directory groups and permissions.

Docker Trusted Registry (DTR): A Docker registry service (from Docker) that can be installed on-premises so it lives within the organization's datacenter and network. It's convenient for private images that should be managed within the enterprise. Docker Trusted Registry is included as part of the Docker Datacenter product.

Docker Desktop: Development tools for Windows and macOS for building, running, and testing containers locally. Docker Desktop for Windows provides development environments for both Linux and Windows Containers. The Linux Docker host on Windows is based on a Hyper-V virtual machine. The host for Windows Containers is directly based on Windows. Docker Desktop for Mac is based on the Apple Hypervisor framework and the xhyve hypervisor, which provides a Linux Docker host virtual machine on macOS. Docker Desktop for Windows and for Mac replaces Docker Toolbox, which was based on Oracle VirtualBox.

Compose: A command-line tool and YAML file format with metadata for defining and running multi-container applications. You define a single application based on multiple images with one or more .yml files that can override values depending on the environment. After you've created the definitions,

you can deploy the whole multi-container application with a single command (docker-compose up) that creates a container per image on the Docker host.

Cluster: A collection of Docker hosts exposed as if it were a single virtual Docker host, so that the application can scale to multiple instances of the services spread across multiple hosts within the cluster. Docker clusters can be created with Kubernetes, Azure Service Fabric, Docker Swarm and Mesosphere DC/OS.

Orchestrator: A tool that simplifies the management of clusters and Docker hosts. Orchestrators enable you to manage their images, containers, and hosts through a command-line interface (CLI) or a graphical UI. You can manage container networking, configurations, load balancing, service discovery, high availability, Docker host configuration, and more. An orchestrator is responsible for running, distributing, scaling, and healing workloads across a collection of nodes. Typically, orchestrator products are the same products that provide cluster infrastructure, like Kubernetes and Azure Service Fabric, among other offerings in the market.

Docker containers, images, and registries

When using Docker, a developer creates an app or service and packages it and its dependencies into a container image. An image is a static representation of the app or service and its configuration and dependencies.

To run the app or service, the app's image is instantiated to create a container, which will be running on the Docker host. Containers are initially tested in a development environment or PC.

Developers should store images in a registry, which acts as a library of images and is needed when deploying to production orchestrators. Docker maintains a public registry via Docker Hub; other vendors provide registries for different collections of images, including Azure Container Registry. Alternatively, enterprises can have a private registry on-premises for their own Docker images.

Figure 2-4 shows how images and registries in Docker relate to other components. It also shows the multiple registry offerings from vendors.

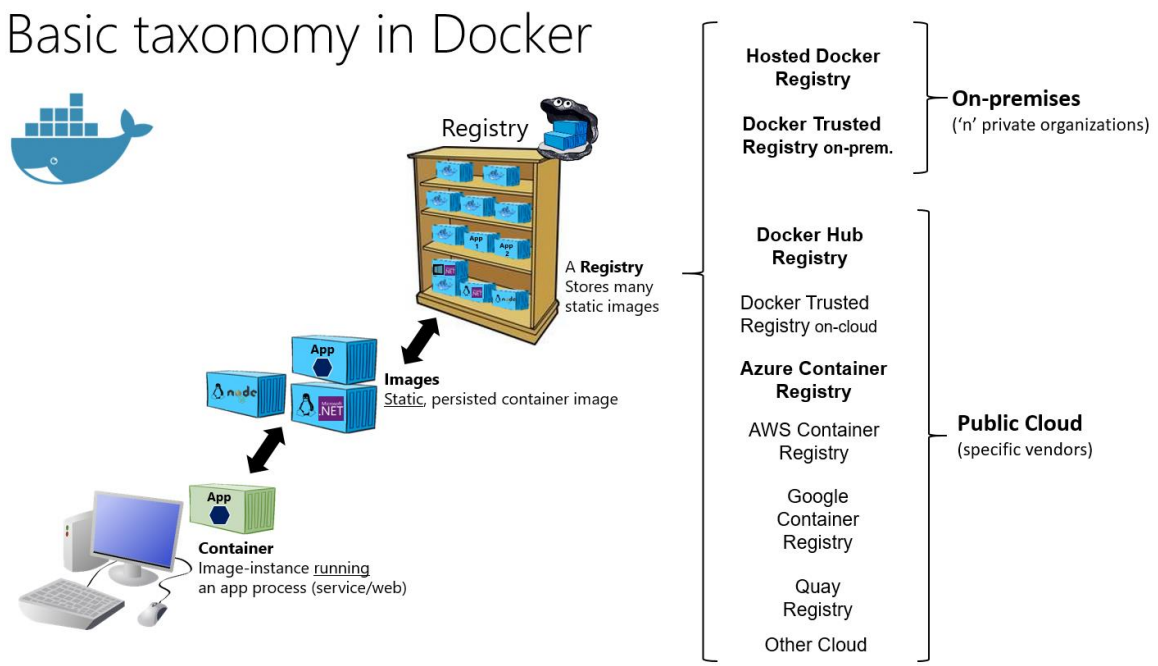


Figure 2-4. Taxonomy of Docker terms and concepts

The registry is like a bookshelf where images are stored and available to be pulled for building containers to run services or web apps. There are private Docker registries on-premises and on the public cloud. Docker Hub is a public registry maintained by Docker, along the Docker Trusted Registry an enterprise-grade solution, Azure offers the Azure Container Registry. AWS, Google, and others also have container registries.

Putting images in a registry lets you store static and immutable application bits, including all their dependencies at a framework level. Those images can then be versioned and deployed in multiple environments and therefore provide a consistent deployment unit.

Private image registries, either hosted on-premises or in the cloud, are recommended when:

- Your images must not be shared publicly due to confidentiality.
- You want to have minimum network latency between your images and your chosen deployment environment. For example, if your production environment is Azure cloud, you probably want to store your images in Azure Container Registry so that network latency will be minimal. In a similar way, if your production environment is on-premises, you might want to have an on-premises Docker Trusted Registry available within the same local network.

Choosing Between .NET and .NET Framework for Docker Containers

There are two supported frameworks for building server-side containerized Docker applications with .NET: .NET Framework and .NET 7. They share many .NET platform components, and you can share code across the two. However, there are fundamental differences between them, and which framework you use will depend on what you want to accomplish. This section provides guidance on when to choose each framework.

General guidance

This section provides a summary of when to choose .NET 7 or .NET Framework. We provide more details about these choices in the sections that follow.

Use .NET 7, with Linux or Windows Containers, for your containerized Docker server application when:

- You have cross-platform needs. For example, you want to use both Linux and Windows Containers.
- Your application architecture is based on microservices.
- You need to start containers fast and want a small footprint per container to achieve better density or more containers per hardware unit in order to lower your costs.

In short, when you create new containerized .NET applications, you should consider .NET 7 as the default choice. It has many benefits and fits best with the containers philosophy and style of working.

An extra benefit of using .NET 7 is that you can run side-by-side .NET versions for applications within the same machine. This benefit is more important for servers or VMs that do not use containers, because containers isolate the versions of .NET that the app needs. (As long as they are compatible with the underlying OS.)

Use .NET Framework for your containerized Docker server application when:

- Your application currently uses .NET Framework and has strong dependencies on Windows.

- You need to use Windows APIs that are not supported by .NET 7.
- You need to use third-party .NET libraries or NuGet packages that are not available for .NET 7.

Using .NET Framework on Docker can improve your deployment experiences by minimizing deployment issues. This "lift and shift" scenario is important for containerizing legacy applications that were originally developed with the traditional .NET Framework, like ASP.NET WebForms, MVC web apps, or WCF (Windows Communication Foundation) services.

Additional resources

- **E-book: Modernize existing .NET Framework applications with Azure and Windows Containers**
  https://aka.ms/liftandshiftwithcontainersebook
- **Sample apps: Modernization of legacy ASP.NET web apps by using Windows Containers**
  https://aka.ms/eshopmodernizing

When to choose .NET for Docker containers

The modularity and lightweight nature of .NET 7 makes it perfect for containers. When you deploy and start a container, its image is far smaller with .NET 7 than with .NET Framework. In contrast, to use .NET Framework for a container, you must base your image on the Windows Server Core image, which is a lot heavier than the Windows Nano Server or Linux images that you use for .NET 7.

Additionally, .NET 7 is cross-platform, so you can deploy server apps with Linux or Windows container images. However, if you are using the traditional .NET Framework, you can only deploy images based on Windows Server Core.

The following is a more detailed explanation of why to choose .NET 7.

Developing and deploying cross platform

Clearly, if your goal is to have an application (web app or service) that can run on multiple platforms supported by Docker (Linux and Windows), the right choice is .NET 7, because .NET Framework only supports Windows.

.NET 7 also supports macOS as a development platform. However, when you deploy containers to a Docker host, that host must (currently) be based on Linux or Windows. For example, in a development environment, you could use a Linux VM running on a Mac.

Visual Studio provides an integrated development environment (IDE) for Windows and supports Docker development.

Visual Studio for Mac is an IDE, evolution of Xamarin Studio, that runs on macOS and supports Docker-based application development. This tool should be the preferred choice for developers working in Mac machines who also want to use a powerful IDE.

You can also use Visual Studio Code on macOS, Linux, and Windows. Visual Studio Code fully supports .NET 7, including IntelliSense and debugging. Because VS Code is a lightweight editor, you

can use it to develop containerized apps on the machine in conjunction with the Docker CLI and the .NET CLI. You can also target .NET 7 with most third-party editors like Sublime, Emacs, vi, and the open-source OmniSharp project, which also provides IntelliSense support.

In addition to the IDEs and editors, you can use the .NET CLI for all supported platforms.

Using containers for new ("green-field") projects

Containers are commonly used in conjunction with a microservices architecture, although they can also be used to containerize web apps or services that follow any architectural pattern. You can use .NET Framework on Windows Containers, but the modularity and lightweight nature of .NET 7 makes it perfect for containers and microservices architectures. When you create and deploy a container, its image is far smaller with .NET 7 than with .NET Framework.

Create and deploy microservices on containers

You could use the traditional .NET Framework for building microservices-based applications (without containers) by using plain processes. That way, because the .NET Framework is already installed and shared across processes, processes are light and fast to start. However, if you are using containers, the image for the traditional .NET Framework is also based on Windows Server Core and that makes it too heavy for a microservices-on-containers approach. However, teams have been looking for opportunities to improve the experience for .NET Framework users as well. Recently, size of the Windows Server Core container images have been reduced to >40% smaller.

On the other hand, .NET 7 is the best candidate if you're embracing a microservices-oriented system that is based on containers because .NET 7 is lightweight. In addition, its related container images, for either Linux or Windows Nano Server, are lean and small, making containers light and fast to start.

A microservice is meant to be as small as possible: to be light when spinning up, to have a small footprint, to have a small Bounded Context (check DDD, Domain-Driven Design), to represent a small area of concerns, and to be able to start and stop fast. For those requirements, you will want to use small and fast-to-instantiate container images like the .NET 7 container image.

A microservices architecture also allows you to mix technologies across a service boundary. This approach enables a gradual migration to .NET 7 for new microservices that work in conjunction with other microservices or with services developed with Node.js, Python, Java, GoLang, or other technologies.

Deploying high density in scalable systems

When your container-based system needs the best possible density, granularity, and performance, .NET and ASP.NET Core are your best options. ASP.NET Core is up to 10 times faster than ASP.NET in the traditional .NET Framework, and it leads to other popular industry technologies for microservices, such as Java servlets, Go, and Node.js.

This approach is especially relevant for microservices architectures, where you could have hundreds of microservices (containers) running. With ASP.NET Core images (based on the .NET runtime) on Linux or Windows Nano, you can run your system with a much lower number of servers or VMs, ultimately saving costs in infrastructure and hosting.

When to choose .NET Framework for Docker containers

While .NET 7 offers significant benefits for new applications and application patterns, .NET Framework will continue to be a good choice for many existing scenarios.

Migrating existing applications directly to a Windows Server container

You might want to use Docker containers just to simplify deployment, even if you are not creating microservices. For example, perhaps you want to improve your DevOps workflow with Docker—containers can give you better isolated test environments and can also eliminate deployment issues caused by missing dependencies when you move to a production environment. In cases like these, even if you are deploying a monolithic application, it makes sense to use Docker and Windows Containers for your current .NET Framework applications.

In most cases for this scenario, you will not need to migrate your existing applications to .NET 7; you can use Docker containers that include the traditional .NET Framework. However, a recommended approach is to use .NET 7 as you extend an existing application, such as writing a new service in ASP.NET Core.

Using third-party .NET libraries or NuGet packages not available for .NET 7

Third-party libraries are quickly embracing .NET Standard, which enables code sharing across all .NET flavors, including .NET 7. With .NET Standard 2.0 and later, the API surface compatibility across different frameworks has become significantly larger. Even more, .NET Core 2.x and newer applications can also directly reference existing .NET Framework libraries (see .NET Framework 4.6.1 supporting .NET Standard 2.0).

In addition, the Windows Compatibility Pack extends the API surface available for .NET Standard 2.0 on Windows. This pack allows recompiling most existing code to .NET Standard 2.x with little or no modification, to run on Windows.

However, even with that exceptional progression since .NET Standard 2.0 and .NET Core 2.1 or later, there might be cases where certain NuGet packages need Windows to run and might not support .NET Core or later. If those packages are critical for your application, then you will need to use .NET Framework on Windows Containers.

Using .NET technologies not available for .NET 7

Some .NET Framework technologies aren't available in .NET 7. Some of them might become available in later releases, but others don't fit the new application patterns targeted by .NET Core and might never be available.

The following list shows most of the technologies that aren't available in .NET 7:

- ASP.NET Web Forms. This technology is only available on .NET Framework. Currently there are no plans to bring ASP.NET Web Forms to .NET or later.
- Workflow-related services. Windows Workflow Foundation (WF), Workflow Services (WCF + WF in a single service), and WCF Data Services (formerly known as ADO.NET Data Services) are only available on .NET Framework. There are currently no plans to bring them to .NET 7.

In addition to the technologies listed in the official .NET roadmap, other features might be ported to the new unified .NET platform. You might consider participating in the discussions on GitHub so that your voice can be heard. And if you think something is missing, file a new issue in the dotnet/runtime GitHub repository.

Using a platform or API that doesn't support .NET 7

Some Microsoft and third-party platforms don't support .NET 7. For example, some Azure services provide an SDK that isn't yet available for consumption on .NET 7 yet. Most Azure SDK should eventually be ported to .NET 7/.NET Standard, but some might not for several reasons. You can see the available Azure SDKs in the Azure SDK Latest Releases page.

In the meantime, if any platform or service in Azure still doesn't support .NET 7 with its client API, you can use the equivalent REST API from the Azure service or the client SDK on .NET Framework.

Porting existing ASP.NET application to .NET 7

.NET Core is a revolutionary step forward from .NET Framework. It offers a host of advantages over .NET Framework across the board from productivity to performance, and from cross-platform support to developer satisfaction. If you are using .NET Framework and planning to migrate your application to .NET Core or .NET 5+, see Porting Existing ASP.NET Apps to .NET Core.

Additional resources

- **.NET fundamentals**
  https://learn.microsoft.com/dotnet/fundamentals
- **Porting Projects to .NET 5**
  https://learn.microsoft.com/events/dotnetconf-2020/porting-projects-to-net-5
- **.NET on Docker Guide**
  https://learn.microsoft.com/dotnet/core/docker/introduction

Decision table: .NET implementations to use for Docker

The following decision table summarizes whether to use .NET Framework or .NET 7. Remember that for Linux containers, you need Linux-based Docker hosts (VMs or servers), and that for Windows Containers, you need Windows Server-based Docker hosts (VMs or servers).

Important

Your development machines will run one Docker host, either Linux or Windows. Related microservices that you want to run and test together in one solution will all need to run on the same container platform.

Architecture / App Type	Linux containers	Windows Containers
Microservices on containers	.NET 7	.NET 7
Monolithic app	.NET 7	.NET Framework .NET 7
Best-in-class performance and scalability	.NET 7	.NET 7
Windows Server legacy app ("brown-field") migration to containers	–	.NET Framework
New container-based development ("green-field")	.NET 7	.NET 7
ASP.NET Core	.NET 7	.NET 7 (recommended) .NET Framework
ASP.NET 4 (MVC 5, Web API 2, and Web Forms)	–	.NET Framework
SignalR services	.NET Core 2.1 or higher version	.NET Framework .NET Core 2.1 or higher version
WCF, WF, and other legacy frameworks	WCF in .NET Core (client library only) or CoreWCF	.NET Framework WCF in .NET 7 (client library only) or CoreWCF
Consumption of Azure services	.NET 7 (eventually most Azure services will provide client SDKs for .NET 7)	.NET Framework .NET 7 (eventually most Azure services will provide client SDKs for .NET 7)

What OS to target with .NET containers

Given the diversity of operating systems supported by Docker and the differences between .NET Framework and .NET 7, you should target a specific OS and specific versions depending on the framework you are using.

For Windows, you can use Windows Server Core or Windows Nano Server. These Windows versions provide different characteristics (IIS in Windows Server Core versus a self-hosted web server like Kestrel in Nano Server) that might be needed by .NET Framework or .NET 7, respectively.

For Linux, multiple distros are available and supported in official .NET Docker images (like Debian).

In Figure 3-1, you can see the possible OS version depending on the .NET framework used.

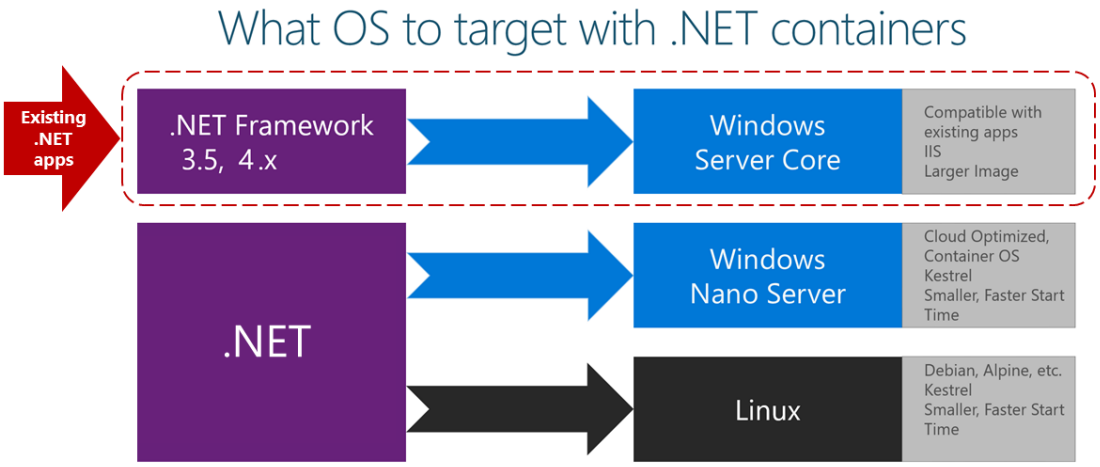


Figure 3-1. Operating systems to target depending on versions of the .NET framework

When deploying legacy .NET Framework applications you have to target Windows Server Core, compatible with legacy apps and IIS, but it has a larger image. When deploying .NET 7 applications, you can target Windows Nano Server, which is cloud optimized, uses Kestrel and is smaller and starts faster. You can also target Linux, supporting Debian, Alpine, and others.

You can also create your own Docker image in cases where you want to use a different Linux distro or where you want an image with versions not provided by Microsoft. For example, you might create an image with ASP.NET Core running on the traditional .NET Framework and Windows Server Core, which is a not-so-common scenario for Docker.

When you add the image name to your Dockerfile file, you can select the operating system and version depending on the tag you use, as in the following examples:

Image	Comments
mcr.microsoft.com/dotnet/runtime:7.0	.NET 7 multi-architecture: Supports Linux and Windows Nano Server depending on the Docker host.
mcr.microsoft.com/dotnet/aspnet:7.0	ASP.NET Core 7.0 multi-architecture: Supports Linux and Windows Nano Server depending on the Docker host. The aspnetcore image has a few optimizations for ASP.NET Core.
mcr.microsoft.com/dotnet/aspnet:7.0-bullseye-slim	.NET 7 runtime-only on Linux Debian distro
mcr.microsoft.com/dotnet/aspnet:7.0-nanoserver-1809	.NET 7 runtime-only on Windows Nano Server (Windows Server version 1809)

Official .NET Docker images

The Official .NET Docker images are Docker images created and optimized by Microsoft. They're publicly available on Microsoft Artifact Registry. You can search over the catalog to find all .NET image repositories, for example .NET SDK repository.

Each repository can contain multiple images, depending on .NET versions, and depending on the OS and versions (Linux Debian, Linux Alpine, Windows Nano Server, Windows Server Core, and so on). Image repositories provide extensive tagging to help you select not just a specific framework version, but also to choose an OS (Linux distribution or Windows version).

.NET and Docker image optimizations for development versus production

When building Docker images for developers, Microsoft focused on the following main scenarios:

- Images used to *develop* and build .NET apps.
- Images used to *run* .NET apps.

Why multiple images? When developing, building, and running containerized applications, you usually have different priorities. By providing different images for these separate tasks, Microsoft helps optimize the separate processes of developing, building, and deploying apps.

During development and build

During development, what is important is how fast you can iterate changes, and the ability to debug the changes. The size of the image isn't as important as the ability to make changes to your code and see the changes quickly. Some tools and "build-agent containers", use the development .NET image (*mcr.microsoft.com/dotnet/sdk:7.0*) during development and build process. When building inside a Docker container, the important aspects are the elements that are needed to compile your app. This includes the compiler and any other .NET dependencies.

Why is this type of build image important? You don't deploy this image to production. Instead, it's an image that you use to build the content you place into a production image. This image would be used in your continuous integration (CI) environment or build environment when using Docker multi-stage builds.

In production

What is important in production is how fast you can deploy and start your containers based on a production .NET image. Therefore, the runtime-only image based on *mcr.microsoft.com/dotnet/aspnet:7.0* is small so that it can travel quickly across the network from your Docker registry to your Docker hosts. The contents are ready to run, enabling the fastest time from starting the container to processing results. In the Docker model, there is no need for compilation from C# code, as there's when you run dotnet build or dotnet publish when using the build container.

In this optimized image, you put only the binaries and other content needed to run the application. For example, the content created by dotnet publish contains only the compiled .NET binaries, images, .js, and .css files. Over time, you'll see images that contain pre-jitted (the compilation from IL to native that occurs at run time) packages.

Although there are multiple versions of the .NET and ASP.NET Core images, they all share one or more layers, including the base layer. Therefore, the amount of disk space needed to store an image is small; it consists only of the delta between your custom image and its base image. The result is that it's quick to pull the image from your registry.

When you explore the .NET image repositories at Microsoft Artifact Registry, you'll find multiple image versions classified or marked with tags. These tags help to decide which one to use, depending on the version you need, like those in the following table:

Image	Comments
mcr.microsoft.com/dotnet/aspnet:**7.0**	ASP.NET Core, with runtime only and ASP.NET Core optimizations, on Linux and Windows (multi-arch)
mcr.microsoft.com/dotnet/sdk:**7.0**	.NET 7, with SDKs included, on Linux and Windows (multi-arch)

You can find all the available docker images in dotnet-docker and also refer to the latest preview releases by using nightly build mcr.microsoft.com/dotnet/nightly/*

Architecting container and microservice-based applications

Microservices offer great benefits but also raise huge new challenges. Microservice architecture patterns are fundamental pillars when creating a microservice-based application.

Earlier in this guide, you learned basic concepts about containers and Docker. That information was the minimum you needed to get started with containers. Even though containers are enablers of, and a great fit for microservices, they aren't mandatory for a microservice architecture. Many architectural concepts in this architecture section could be applied without containers. However, this guide focuses on the intersection of both due to the already introduced importance of containers.

Enterprise applications can be complex and are often composed of multiple services instead of a single service-based application. For those cases, you need to understand other architectural approaches, such as the microservices and certain Domain-Driven Design (DDD) patterns plus container orchestration concepts. Note that this chapter describes not just microservices on containers, but any containerized application, as well.

Container design principles

In the container model, a container image instance represents a single process. By defining a container image as a process boundary, you can create primitives that can be used to scale or batch the process.

When you design a container image, you'll see an ENTRYPOINT definition in the Dockerfile. This definition defines the process whose lifetime controls the lifetime of the container. When the process completes, the container lifecycle ends. Containers might represent long-running processes like web servers, but can also represent short-lived processes like batch jobs, which formerly might have been implemented as Azure WebJobs.

If the process fails, the container ends, and the orchestrator takes over. If the orchestrator was configured to keep five instances running and one fails, the orchestrator will create another container instance to replace the failed process. In a batch job, the process is started with parameters. When the process completes, the work is complete. This guidance drills-down on orchestrators, later on.

You might find a scenario where you want multiple processes running in a single container. For that scenario, since there can be only one entry point per container, you could run a script within the container that launches as many programs as needed. For example, you can use Supervisor or a similar tool to take care of launching multiple processes inside a single container. However, even though you can find architectures that hold multiple processes per container, that approach isn't very common.

Containerizing monolithic applications

You might want to build a single, monolithically deployed web application or service and deploy it as a container. The application itself might not be internally monolithic, but structured as several libraries, components, or even layers (application layer, domain layer, data-access layer, etc.). Externally, however, it's a single container—a single process, a single web application, or a single service.

To manage this model, you deploy a single container to represent the application. To increase capacity, you scale out, that is, just add more copies with a load balancer in front. The simplicity comes from managing a single deployment in a single container or VM.

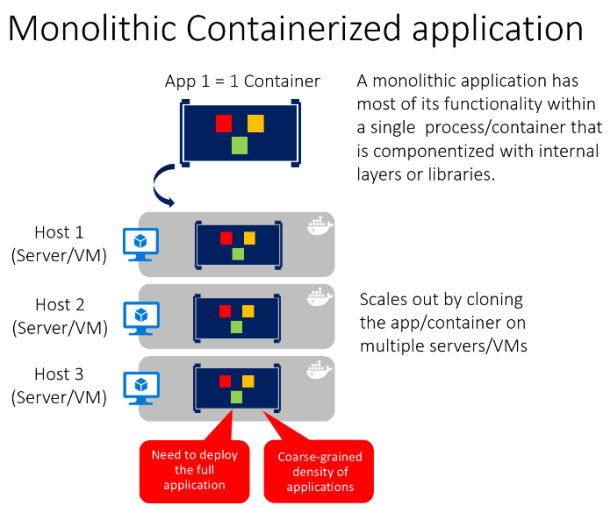


Figure 4-1. Example of the architecture of a containerized monolithic application

You can include multiple components, libraries, or internal layers in each container, as illustrated in Figure 4-1. A monolithic containerized application has most of its functionality within a single container, with internal layers or libraries, and scales out by cloning the container on multiple servers/VMs. However, this monolithic pattern might conflict with the container principle "a container does one thing, and does it in one process", but might be ok for some cases.

The downside of this approach becomes evident if the application grows, requiring it to scale. If the entire application can scale, it isn't really a problem. However, in most cases, just a few parts of the application are the choke points that require scaling, while other components are used less.

For example, in a typical e-commerce application, you likely need to scale the product information subsystem, because many more customers browse products than purchase them. More customers use

their basket than use the payment pipeline. Fewer customers add comments or view their purchase history. And you might have only a handful of employees that need to manage the content and marketing campaigns. If you scale the monolithic design, all the code for these different tasks is deployed multiple times and scaled at the same grade.

There are multiple ways to scale an application-horizontal duplication, splitting different areas of the application, and partitioning similar business concepts or data. But, in addition to the problem of scaling all components, changes to a single component require complete retesting of the entire application, and a complete redeployment of all the instances.

However, the monolithic approach is common, because the development of the application is initially easier than for microservices approaches. Thus, many organizations develop using this architectural approach. While some organizations have had good enough results, others are hitting limits. Many organizations designed their applications using this model because tools and infrastructure made it too difficult to build service-oriented architectures (SOA) years ago, and they did not see the need-until the application grew.

From an infrastructure perspective, each server can run many applications within the same host and have an acceptable ratio of efficiency in resources usage, as shown in Figure 4-2.

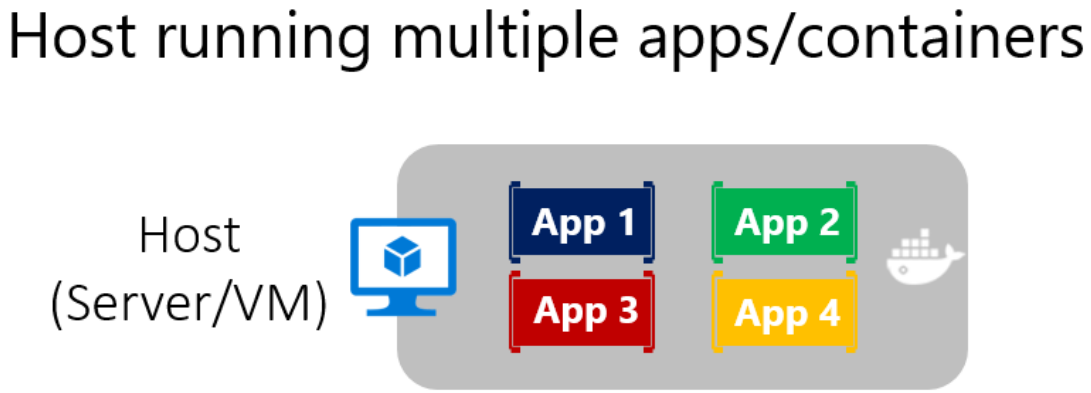


Figure 4-2. Monolithic approach: Host running multiple apps, each app running as a container

Monolithic applications in Microsoft Azure can be deployed using dedicated VMs for each instance. Additionally, using Azure virtual machine scale sets, you can easily scale the VMs. Azure App Service can also run monolithic applications and easily scale instances without requiring you to manage the VMs. Since 2016, Azure App Services can run single instances of Docker containers as well, simplifying deployment.

As a QA environment or a limited production environment, you can deploy multiple Docker host VMs and balance them using the Azure balancer, as shown in Figure 4-3. This lets you manage scaling with a coarse-grain approach, because the whole application lives within a single container.

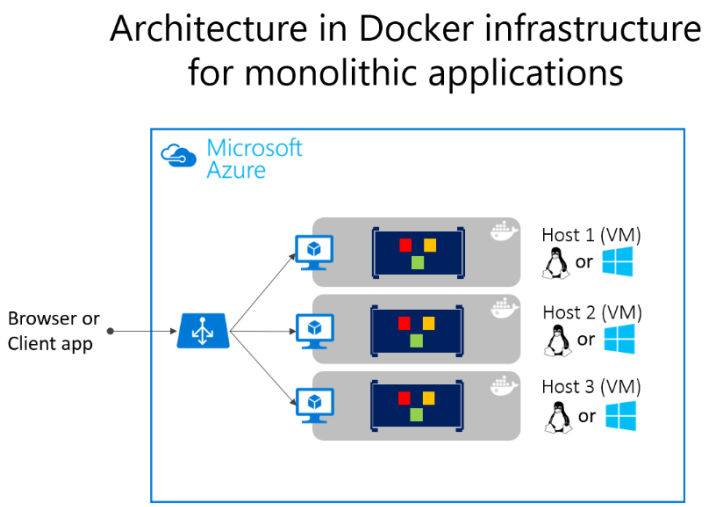


Figure 4-3. Example of multiple hosts scaling up a single container application

Deployment to the various hosts can be managed with traditional deployment techniques. Docker hosts can be managed with commands like docker run or docker-compose performed manually, or through automation such as continuous delivery (CD) pipelines.

Deploying a monolithic application as a container

There are benefits to using containers to manage monolithic application deployments. Scaling container instances is far faster and easier than deploying additional VMs. Even if you use virtual machine scale sets, VMs take time to start. When deployed as traditional application instances instead of containers, the configuration of the application is managed as part of the VM, which isn't ideal.

Deploying updates as Docker images is far faster and network efficient. Docker images typically start in seconds, which speeds rollouts. Tearing down a Docker image instance is as easy as issuing a docker stop command, and typically completes in less than a second.

Because containers are immutable by design, you never need to worry about corrupted VMs. In contrast, update scripts for a VM might forget to account for some specific configuration or file left on disk.

While monolithic applications can benefit from Docker, we're touching only on the benefits. Additional benefits of managing containers come from deploying with container orchestrators, which manage the various instances and lifecycle of each container instance. Breaking up the monolithic application into subsystems that can be scaled, developed, and deployed individually is your entry point into the realm of microservices.

Publishing a single-container-based application to Azure App Service

Whether you want to get validation of a container deployed to Azure or when an application is simply a single-container application, Azure App Service provides a great way to provide scalable single-container-based services. Using Azure App Service is simple. It provides great integration with Git to make it easy to take your code, build it in Visual Studio, and deploy it directly to Azure.

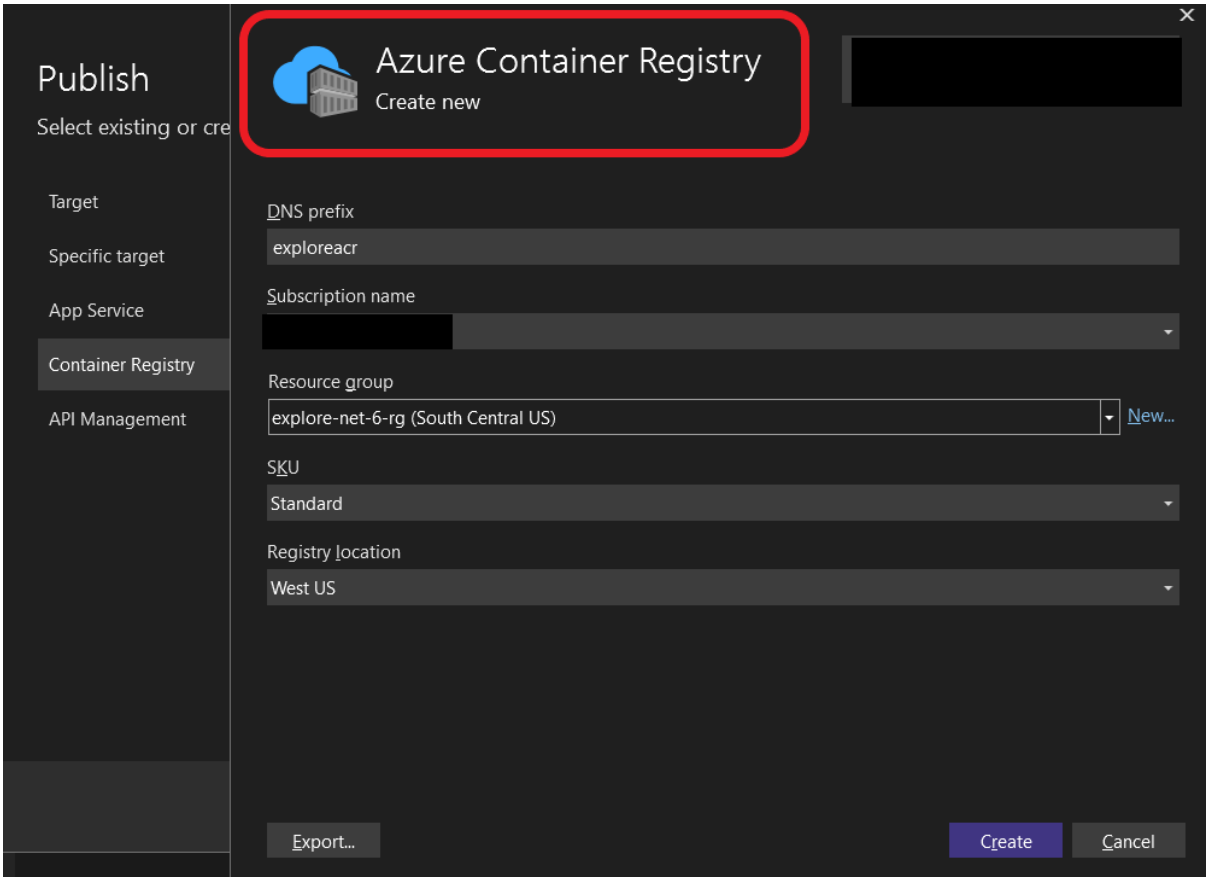


Figure 4-4. Publishing a single-container application to Azure App Service from Visual Studio 2022

Without Docker, if you needed other capabilities, frameworks, or dependencies that aren't supported in Azure App Service, you had to wait until the Azure team updated those dependencies in App Service. Or you had to switch to other services like Azure Cloud Services or VMs, where you had further control and you could install a required component or framework for your application.

Container support in Visual Studio 2017 and later gives you the ability to include whatever you want in your application environment, as shown in Figure 4-4. Since you're running it in a container, if you add a dependency to your application, you can include the dependency in your Dockerfile or Docker image.

As also shown in Figure 4-4, the publish flow pushes an image through a container registry. This can be the Azure Container Registry (a registry close to your deployments in Azure and secured by Azure Active Directory groups and accounts), or any other Docker registry, like Docker Hub or an on-premises registry.

Manage state and data in Docker applications

In most cases, you can think of a container as an instance of a process. A process doesn't maintain persistent state. While a container can write to its local storage, assuming that an instance will be around indefinitely would be like assuming that a single location in memory will be durable. You

should assume that container images, like processes, have multiple instances or will eventually be killed. If they're managed with a container orchestrator, you should assume that they might get moved from one node or VM to another.

The following solutions are used to manage data in Docker applications:

From the Docker host, as Docker Volumes:

- **Volumes** are stored in an area of the host filesystem that's managed by Docker.
- **Bind mounts** can map to any folder in the host filesystem, so access can't be controlled from Docker process and can pose a security risk as a container could access sensitive OS folders.
- **tmpfs mounts** are like virtual folders that only exist in the host's memory and are never written to the filesystem.

From remote storage:

- Azure Storage, which provides geo-distributable storage, providing a good long-term persistence solution for containers.
- Remote relational databases like Azure SQL Database or NoSQL databases like Azure Cosmos DB, or cache services like Redis.

From the Docker container:

- **Overlay File System**. This Docker feature implements a copy-on-write task that stores updated information to the root file system of the container. That information is "on top" of the original image on which the container is based. If the container is deleted from the system, those changes are lost. Therefore, while it's possible to save the state of a container within its local storage, designing a system around this would conflict with the premise of container design, which by default is stateless.

However, using Docker Volumes is now the preferred way to handle local data in Docker. If you need more information about storage in containers check on Docker storage drivers and About storage drivers.

The following provides more detail about these options:

Volumes are directories mapped from the host OS to directories in containers. When code in the container has access to the directory, that access is actually to a directory on the host OS. This directory is not tied to the lifetime of the container itself, and the directory is managed by Docker and isolated from the core functionality of the host machine. Thus, data volumes are designed to persist data independently of the life of the container. If you delete a container or an image from the Docker host, the data persisted in the data volume isn't deleted.

Volumes can be named or anonymous (the default). Named volumes are the evolution of **Data Volume Containers** and make it easy to share data between containers. Volumes also support volume drivers that allow you to store data on remote hosts, among other options.

Bind mounts are available since a long time ago and allow the mapping of any folder to a mount point in a container. Bind mounts have more limitations than volumes and some important security issues, so volumes are the recommended option.

tmpfs mounts are basically virtual folders that live only in the host's memory and are never written to the filesystem. They are fast and secure but use memory and are only meant for temporary, non-persistent data.

As shown in Figure 4-5, regular Docker volumes can be stored outside of the containers themselves but within the physical boundaries of the host server or VM. However, Docker containers can't access a volume from one host server or VM to another. In other words, with these volumes, it isn't possible to manage data shared between containers that run on different Docker hosts, although it could be achieved with a volume driver that supports remote hosts.

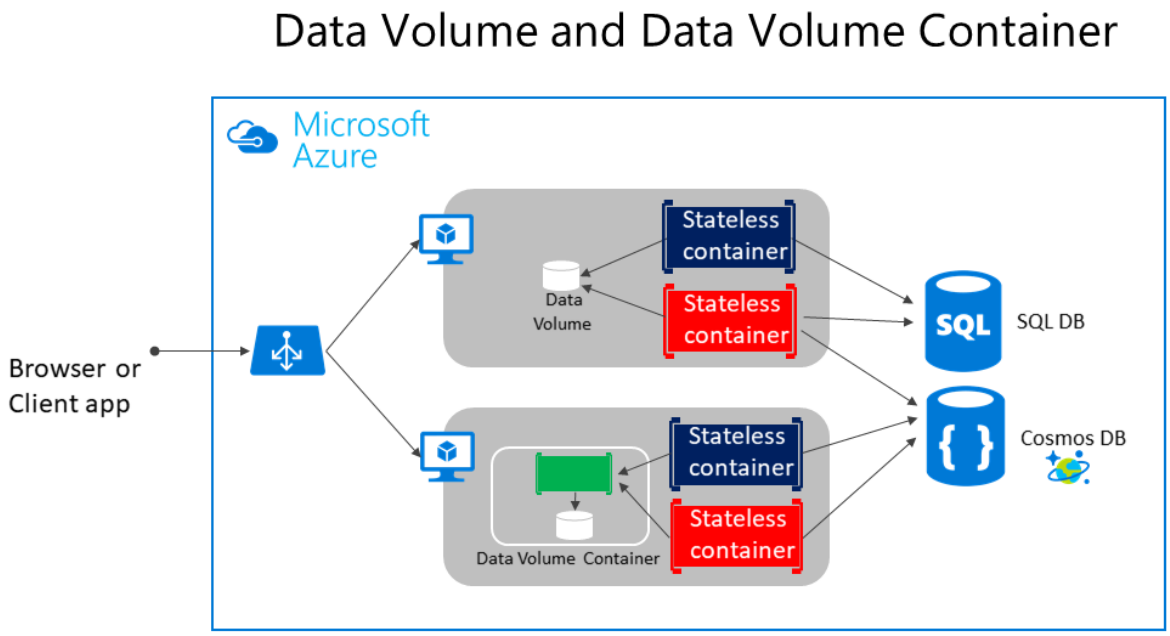


Figure 4-5. Volumes and external data sources for container-based applications

Volumes can be shared between containers, but only in the same host, unless you use a remote driver that supports remote hosts. In addition, when Docker containers are managed by an orchestrator, containers might "move" between hosts, depending on the optimizations performed by the cluster. Therefore, it isn't recommended that you use data volumes for business data. But they're a good mechanism to work with trace files, temporal files, or similar that will not impact business data consistency.

Remote data sources and cache tools like Azure SQL Database, Azure Cosmos DB, or a remote cache like Redis can be used in containerized applications the same way they are used when developing without containers. This is a proven way to store business application data.

Azure Storage. Business data usually will need to be placed in external resources or databases, like Azure Storage. Azure Storage, in concrete, provides the following services in the cloud:

- Blob storage stores unstructured object data. A blob can be any type of text or binary data, such as document or media files (images, audio, and video files). Blob storage is also referred to as Object storage.

- File storage offers shared storage for legacy applications using standard SMB protocol. Azure virtual machines and cloud services can share file data across application components via mounted shares. On-premises applications can access file data in a share via the File service REST API.
- Table storage stores structured datasets. Table storage is a NoSQL key-attribute data store, which allows rapid development and fast access to large quantities of data.

Relational databases and NoSQL databases. There are many choices for external databases, from relational databases like SQL Server, PostgreSQL, Oracle, or NoSQL databases like Azure Cosmos DB, MongoDB, etc. These databases are not going to be explained as part of this guide since they are in a completely different subject.

Service-oriented architecture

Service-oriented architecture (SOA) was an overused term and has meant different things to different people. But as a common denominator, SOA means that you structure your application by decomposing it into multiple services (most commonly as HTTP services) that can be classified as different types like subsystems or tiers.

Those services can now be deployed as Docker containers, which solves deployment issues, because all the dependencies are included in the container image. However, when you need to scale up SOA applications, you might have scalability and availability challenges if you're deploying based on single Docker hosts. This is where Docker clustering software or an orchestrator can help you, as explained in later sections where deployment approaches for microservices are described.

Docker containers are useful (but not required) for both traditional service-oriented architectures and the more advanced microservices architectures.

Microservices derive from SOA, but SOA is different from microservices architecture. Features like large central brokers, central orchestrators at the organization level, and the Enterprise Service Bus (ESB) are typical in SOA. But in most cases, these are anti-patterns in the microservice community. In fact, some people argue that "The microservice architecture is SOA done right."

This guide focuses on microservices, because a SOA approach is less prescriptive than the requirements and techniques used in a microservice architecture. If you know how to build a microservice-based application, you also know how to build a simpler service-oriented application.

Microservices architecture

As the name implies, a microservices architecture is an approach to building a server application as a set of small services. That means a microservices architecture is mainly oriented to the back-end, although the approach is also being used for the front end. Each service runs in its own process and communicates with other processes using protocols such as HTTP/HTTPS, WebSockets, or AMQP. Each microservice implements a specific end-to-end domain or business capability within a certain context boundary, and each must be developed autonomously and be deployable independently. Finally, each microservice should own its related domain data model and domain logic (sovereignty

and decentralized data management) and could be based on different data storage technologies (SQL, NoSQL) and different programming languages.

What size should a microservice be? When developing a microservice, size shouldn't be the important point. Instead, the important point should be to create loosely coupled services so you have autonomy of development, deployment, and scale, for each service. Of course, when identifying and designing microservices, you should try to make them as small as possible as long as you don't have too many direct dependencies with other microservices. More important than the size of the microservice is the internal cohesion it must have and its independence from other services.

Why a microservices architecture? In short, it provides long-term agility. Microservices enable better maintainability in complex, large, and highly-scalable systems by letting you create applications based on many independently deployable services that each have granular and autonomous lifecycles.

As an additional benefit, microservices can scale out independently. Instead of having a single monolithic application that you must scale out as a unit, you can instead scale out specific microservices. That way, you can scale just the functional area that needs more processing power or network bandwidth to support demand, rather than scaling out other areas of the application that don't need to be scaled. That means cost savings because you need less hardware.

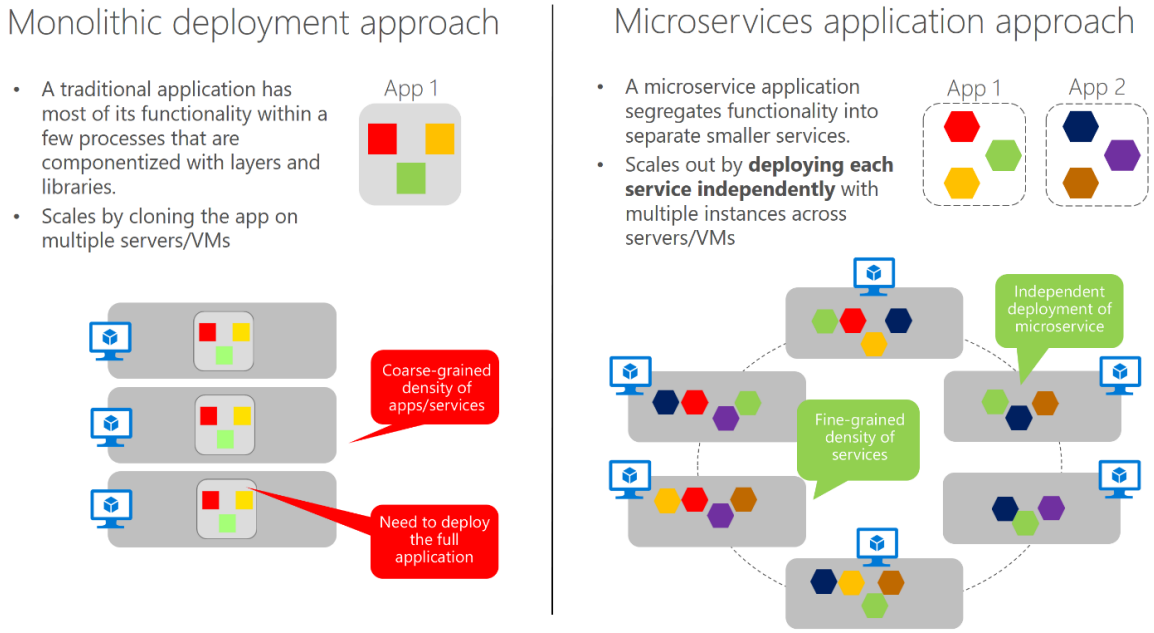


Figure 4-6. Monolithic deployment versus the microservices approach

As Figure 4-6 shows, in the traditional monolithic approach, the application scales by cloning the whole app in several servers/VM. In the microservices approach, functionality is segregated in smaller services, so each service can scale independently. The microservices approach allows agile changes and rapid iteration of each microservice, because you can change specific, small areas of complex, large, and scalable applications.

Architecting fine-grained microservices-based applications enables continuous integration and continuous delivery practices. It also accelerates delivery of new functions into the application. Fine-grained composition of applications also allows you to run and test microservices in isolation, and to

evolve them autonomously while maintaining clear contracts between them. As long as you don't change the interfaces or contracts, you can change the internal implementation of any microservice or add new functionality without breaking other microservices.

The following are important aspects to enable success in going into production with a microservices-based system:

- Monitoring and health checks of the services and infrastructure.
- Scalable infrastructure for the services (that is, cloud and orchestrators).
- Security design and implementation at multiple levels: authentication, authorization, secrets management, secure communication, etc.
- Rapid application delivery, usually with different teams focusing on different microservices.
- DevOps and CI/CD practices and infrastructure.

Of these, only the first three are covered or introduced in this guide. The last two points, which are related to application lifecycle, are covered in the additional Containerized Docker Application Lifecycle with Microsoft Platform and Tools e-book.

Additional resources

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

Data sovereignty per microservice

An important rule for microservices architecture is that each microservice must own its domain data and logic. Just as a full application owns its logic and data, so must each microservice own its logic and data under an autonomous lifecycle, with independent deployment per microservice.

This means that the conceptual model of the domain will differ between subsystems or microservices. Consider enterprise applications, where customer relationship management (CRM) applications,

transactional purchase subsystems, and customer support subsystems each call on unique customer entity attributes and data, and where each employs a different Bounded Context (BC).

This principle is similar in Domain-driven design (DDD), where each Bounded Context or autonomous subsystem or service must own its domain model (data plus logic and behavior). Each DDD Bounded Context correlates to one business microservice (one or several services). This point about the Bounded Context pattern is expanded in the next section.

On the other hand, the traditional (monolithic data) approach used in many applications is to have a single centralized database or just a few databases. This is often a normalized SQL database that's used for the whole application and all its internal subsystems, as shown in Figure 4-7.

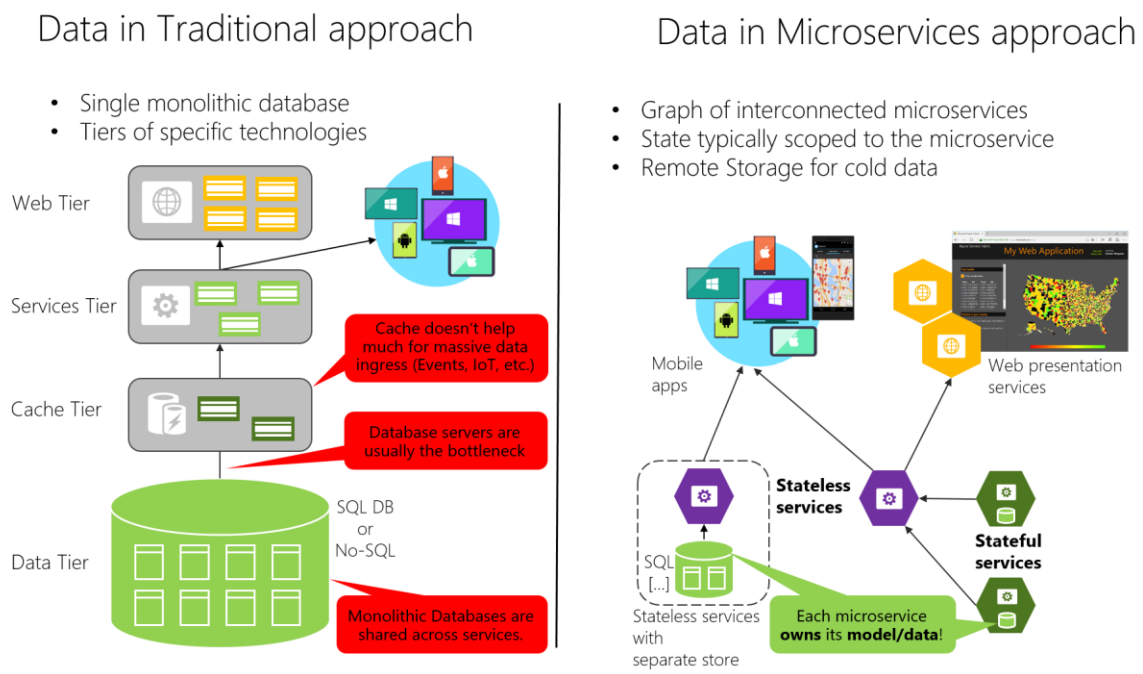


Figure 4-7. Data sovereignty comparison: monolithic database versus microservices

In the traditional approach, there's a single database shared across all services, typically in a tiered architecture. In the microservices approach, each microservice owns its model/data. The centralized database approach initially looks simpler and seems to enable reuse of entities in different subsystems to make everything consistent. But the reality is you end up with huge tables that serve many different subsystems, and that include attributes and columns that aren't needed in most cases. It's like trying to use the same physical map for hiking a short trail, taking a day-long car trip, and learning geography.

A monolithic application with typically a single relational database has two important benefits: ACID transactions and the SQL language, both working across all the tables and data related to your application. This approach provides a way to easily write a query that combines data from multiple tables.

However, data access becomes much more complicated when you move to a microservices architecture. Even when using ACID transactions within a microservice or Bounded Context, it is crucial to consider that the data owned by each microservice is private to that microservice and should only

be accessed either synchronously through its API endpoints(REST, gRPC, SOAP, etc) or asynchronously via messaging(AMQP or similar).

Encapsulating the data ensures that the microservices are loosely coupled and can evolve independently of one another. If multiple services were accessing the same data, schema updates would require coordinated updates to all the services. This would break the microservice lifecycle autonomy. But distributed data structures mean that you can't make a single ACID transaction across microservices. This in turn means you must use eventual consistency when a business process spans multiple microservices. This is much harder to implement than simple SQL joins, because you can't create integrity constraints or use distributed transactions between separate databases, as we'll explain later on. Similarly, many other relational database features aren't available across multiple microservices.

Going even further, different microservices often use different *kinds* of databases. Modern applications store and process diverse kinds of data, and a relational database isn't always the best choice. For some use cases, a NoSQL database such as Azure CosmosDB or MongoDB might have a more convenient data model and offer better performance and scalability than a SQL database like SQL Server or Azure SQL Database. In other cases, a relational database is still the best approach. Therefore, microservices-based applications often use a mixture of SQL and NoSQL databases, which is sometimes called the polyglot persistence approach.

A partitioned, polyglot-persistent architecture for data storage has many benefits. These include loosely coupled services and better performance, scalability, costs, and manageability. However, it can introduce some distributed data management challenges, as explained in "Identifying domain-model boundaries" later in this chapter.

The relationship between microservices and the Bounded Context pattern

The concept of microservice derives from the Bounded Context (BC) pattern in domain-driven design (DDD). DDD deals with large models by dividing them into multiple BCs and being explicit about their boundaries. Each BC must have its own model and database; likewise, each microservice owns its related data. In addition, each BC usually has its own ubiquitous language to help communication between software developers and domain experts.

Those terms (mainly domain entities) in the ubiquitous language can have different names in different Bounded Contexts, even when different domain entities share the same identity (that is, the unique ID that's used to read the entity from storage). For instance, in a user-profile Bounded Context, the User domain entity might share identity with the Buyer domain entity in the ordering Bounded Context.

A microservice is therefore like a Bounded Context, but it also specifies that it's a distributed service. It's built as a separate process for each Bounded Context, and it must use the distributed protocols noted earlier, like HTTP/HTTPS, WebSockets, or AMQP. The Bounded Context pattern, however, doesn't specify whether the Bounded Context is a distributed service or if it's simply a logical boundary (such as a generic subsystem) within a monolithic-deployment application.

It's important to highlight that defining a service for each Bounded Context is a good place to start. But you don't have to constrain your design to it. Sometimes you must design a Bounded Context or

business microservice composed of several physical services. But ultimately, both patterns -Bounded Context and microservice- are closely related.

DDD benefits from microservices by getting real boundaries in the form of distributed microservices. But ideas like not sharing the model between microservices are what you also want in a Bounded Context.

Additional resources

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

Logical architecture versus physical architecture

It's useful at this point to stop and discuss the distinction between logical architecture and physical architecture, and how this applies to the design of microservice-based applications.

To begin, building microservices doesn't require the use of any specific technology. For instance, Docker containers aren't mandatory to create a microservice-based architecture. Those microservices could also be run as plain processes. Microservices is a logical architecture.

Moreover, even when a microservice could be physically implemented as a single service, process, or container (for simplicity's sake, that's the approach taken in the initial version of eShopOnContainers), this parity between business microservice and physical service or container isn't necessarily required in all cases when you build a large and complex application composed of many dozens or even hundreds of services.

This is where there's a difference between an application's logical architecture and physical architecture. The logical architecture and logical boundaries of a system do not necessarily map one-to-one to the physical or deployment architecture. It can happen, but it often doesn't.

Although you might have identified certain business microservices or Bounded Contexts, it doesn't mean that the best way to implement them is always by creating a single service (such as an ASP.NET Web API) or single Docker container for each business microservice. Having a rule saying each business microservice has to be implemented using a single service or container is too rigid.

Therefore, a business microservice or Bounded Context is a logical architecture that might coincide (or not) with physical architecture. The important point is that a business microservice or Bounded Context must be autonomous by allowing code and state to be independently versioned, deployed, and scaled.

As Figure 4-8 shows, the catalog business microservice could be composed of several services or processes. These could be multiple ASP.NET Web API services or any other kind of services using HTTP or any other protocol. More importantly, the services could share the same data, as long as these services are cohesive with respect to the same business domain.

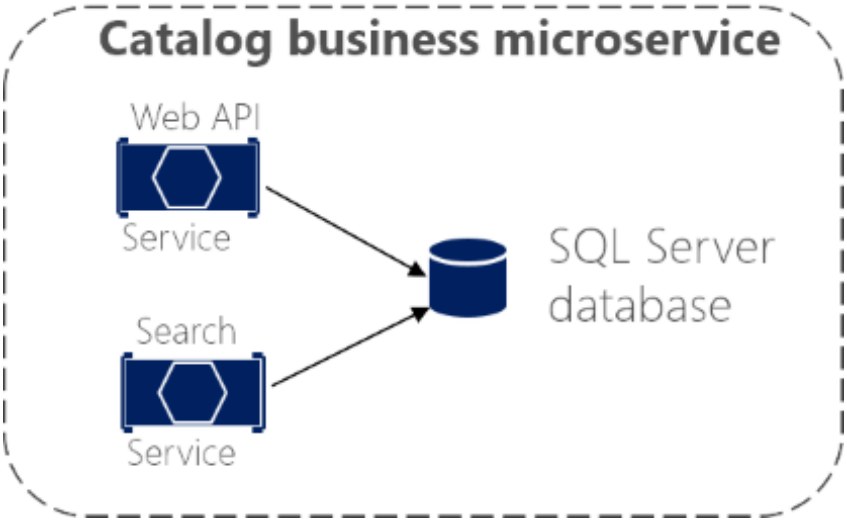


Figure 4-8. Business microservice with several physical services

The services in the example share the same data model because the Web API service targets the same data as the Search service. So, in the physical implementation of the business microservice, you're splitting that functionality so you can scale each of those internal services up or down as needed. Maybe the Web API service usually needs more instances than the Search service, or vice versa.

In short, the logical architecture of microservices doesn't always have to coincide with the physical deployment architecture. In this guide, whenever we mention a microservice, we mean a business or logical microservice that could map to one or more (physical) services. In most cases, this will be a single service, but it might be more.

Challenges and solutions for distributed data management

Challenge #1: How to define the boundaries of each microservice

Defining microservice boundaries is probably the first challenge anyone encounters. Each microservice has to be a piece of your application and each microservice should be autonomous with all the benefits and challenges that it conveys. But how do you identify those boundaries?

First, you need to focus on the application's logical domain models and related data. Try to identify decoupled islands of data and different contexts within the same application. Each context could have a different business language (different business terms). The contexts should be defined and managed independently. The terms and entities that are used in those different contexts might sound similar, but you might discover that in a particular context, a business concept with one is used for a different

purpose in another context, and might even have a different name. For instance, a user can be referred as a user in the identity or membership context, as a customer in a CRM context, as a buyer in an ordering context, and so forth.

The way you identify boundaries between multiple application contexts with a different domain for each context is exactly how you can identify the boundaries for each business microservice and its related domain model and data. You always attempt to minimize the coupling between those microservices. This guide goes into more detail about this identification and domain model design in the section Identifying domain-model boundaries for each microservice later.

Challenge #2: How to create queries that retrieve data from several microservices

A second challenge is how to implement queries that retrieve data from several microservices, while avoiding chatty communication to the microservices from remote client apps. An example could be a single screen from a mobile app that needs to show user information that's owned by the basket, catalog, and user identity microservices. Another example would be a complex report involving many tables located in multiple microservices. The right solution depends on the complexity of the queries. But in any case, you'll need a way to aggregate information if you want to improve the efficiency in the communications of your system. The most popular solutions are the following.

API Gateway. For simple data aggregation from multiple microservices that own different databases, the recommended approach is an aggregation microservice referred to as an API Gateway. However, you need to be careful about implementing this pattern, because it can be a choke point in your system, and it can violate the principle of microservice autonomy. To mitigate this possibility, you can have multiple fined-grained API Gateways each one focusing on a vertical "slice" or business area of the system. The API Gateway pattern is explained in more detail in the API Gateway section later.

GraphQL Federation One option to consider if your microservices are already using GraphQL is GraphQL Federation. Federation allows you to define "subgraphs" from other services and compose them into an aggregate "supergraph" that acts as a standalone schema.

CQRS with query/reads tables. Another solution for aggregating data from multiple microservices is the Materialized View pattern. In this approach, you generate, in advance (prepare denormalized data before the actual queries happen), a read-only table with the data that's owned by multiple microservices. The table has a format suited to the client app's needs.

Consider something like the screen for a mobile app. If you have a single database, you might pull together the data for that screen using a SQL query that performs a complex join involving multiple tables. However, when you have multiple databases, and each database is owned by a different microservice, you cannot query those databases and create a SQL join. Your complex query becomes a challenge. You can address the requirement using a CQRS approach—you create a denormalized table in a different database that's used just for queries. The table can be designed specifically for the data you need for the complex query, with a one-to-one relationship between fields needed by your application's screen and the columns in the query table. It could also serve for reporting purposes.

This approach not only solves the original problem (how to query and join across microservices), but it also improves performance considerably when compared with a complex join, because you already

have the data that the application needs in the query table. Of course, using Command and Query Responsibility Segregation (CQRS) with query/reads tables means additional development work, and you'll need to embrace eventual consistency. Nonetheless, requirements on performance and high scalability in collaborative scenarios (or competitive scenarios, depending on the point of view) are where you should apply CQRS with multiple databases.

"Cold data" in central databases. For complex reports and queries that might not require real-time data, a common approach is to export your "hot data" (transactional data from the microservices) as "cold data" into large databases that are used only for reporting. That central database system can be a Big Data-based system, like Hadoop; a data warehouse like one based on Azure SQL Data Warehouse; or even a single SQL database that's used just for reports (if size won't be an issue).

Keep in mind that this centralized database would be used only for queries and reports that do not need real-time data. The original updates and transactions, as your source of truth, have to be in your microservices data. The way you would synchronize data would be either by using event-driven communication (covered in the next sections) or by using other database infrastructure import/export tools. If you use event-driven communication, that integration process would be similar to the way you propagate data as described earlier for CQRS query tables.

However, if your application design involves constantly aggregating information from multiple microservices for complex queries, it might be a symptom of a bad design -a microservice should be as isolated as possible from other microservices. (This excludes reports/analytics that always should use cold-data central databases.) Having this problem often might be a reason to merge microservices. You need to balance the autonomy of evolution and deployment of each microservice with strong dependencies, cohesion, and data aggregation.

Challenge #3: How to achieve consistency across multiple microservices

As stated previously, the data owned by each microservice is private to that microservice and can only be accessed using its microservice API. Therefore, a challenge presented is how to implement end-to-end business processes while keeping consistency across multiple microservices.

To analyze this problem, let's look at an example from the eShopOnContainers reference application. The Catalog microservice maintains information about all the products, including the product price. The Basket microservice manages temporal data about product items that users are adding to their shopping baskets, which includes the price of the items at the time they were added to the basket. When a product's price is updated in the catalog, that price should also be updated in the active baskets that hold that same product, plus the system should probably warn the user saying that a particular item's price has changed since they added it to their basket.

In a hypothetical monolithic version of this application, when the price changes in the products table, the catalog subsystem could simply use an ACID transaction to update the current price in the Basket table.

However, in a microservices-based application, the Product and Basket tables are owned by their respective microservices. No microservice should ever include tables/storage owned by another microservice in its own transactions, not even in direct queries, as shown in Figure 4-9.

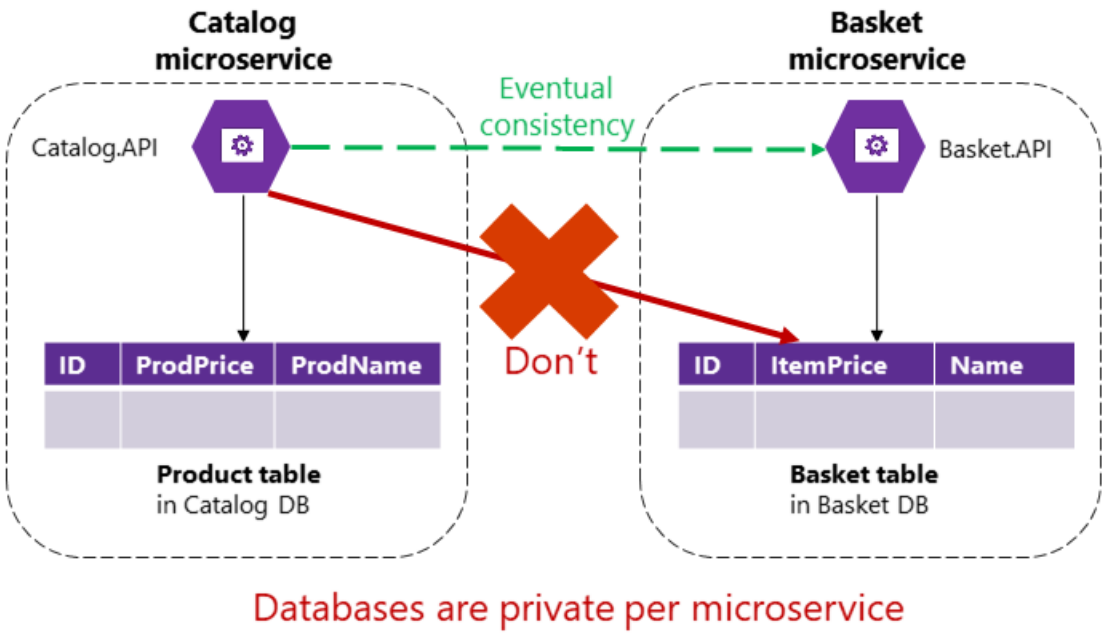


Figure 4-9. A microservice can't directly access a table in another microservice

The Catalog microservice shouldn't update the Basket table directly, because the Basket table is owned by the Basket microservice. To make an update to the Basket microservice, the Catalog microservice should use eventual consistency probably based on asynchronous communication such as integration events (message and event-based communication). This is how the eShopOnContainers reference application performs this type of consistency across microservices.

As stated by the CAP theorem, you need to choose between availability and ACID strong consistency. Most microservice-based scenarios demand availability and high scalability as opposed to strong consistency. Mission-critical applications must remain up and running, and developers can work around strong consistency by using techniques for working with weak or eventual consistency. This is the approach taken by most microservice-based architectures.

Moreover, ACID-style or two-phase commit transactions are not just against microservices principles; most NoSQL databases (like Azure Cosmos DB, MongoDB, etc.) do not support two-phase commit transactions, typical in distributed databases scenarios. However, maintaining data consistency across services and databases is essential. This challenge is also related to the question of how to propagate changes across multiple microservices when certain data needs to be redundant—for example, when you need to have the product's name or description in the Catalog microservice and the Basket microservice.

A good solution for this problem is to use eventual consistency between microservices articulated through event-driven communication and a publish-and-subscribe system. These topics are covered in the section Asynchronous event-driven communication later in this guide.

Challenge #4: How to design communication across microservice boundaries

Communicating across microservice boundaries is a real challenge. In this context, communication doesn't refer to what protocol you should use (HTTP and REST, AMQP, messaging, and so on). Instead, it addresses what communication style you should use, and especially how coupled your microservices should be. Depending on the level of coupling, when failure occurs, the impact of that failure on your system will vary significantly.

In a distributed system like a microservices-based application, with so many artifacts moving around and with distributed services across many servers or hosts, components will eventually fail. Partial failure and even larger outages will occur, so you need to design your microservices and the communication across them considering the common risks in this type of distributed system.

A popular approach is to implement HTTP (REST)-based microservices, due to their simplicity. An HTTP-based approach is perfectly acceptable; the issue here is related to how you use it. If you use HTTP requests and responses just to interact with your microservices from client applications or from API Gateways, that's fine. But if you create long chains of synchronous HTTP calls across microservices, communicating across their boundaries as if the microservices were objects in a monolithic application, your application will eventually run into problems.

For instance, imagine that your client application makes an HTTP API call to an individual microservice like the Ordering microservice. If the Ordering microservice in turn calls additional microservices using HTTP within the same request/response cycle, you're creating a chain of HTTP calls. It might sound reasonable initially. However, there are important points to consider when going down this path:

- Blocking and low performance. Due to the synchronous nature of HTTP, the original request doesn't get a response until all the internal HTTP calls are finished. Imagine if the number of these calls increases significantly and at the same time one of the intermediate HTTP calls to a microservice is blocked. The result is that performance is impacted, and the overall scalability will be exponentially affected as additional HTTP requests increase.
- Coupling microservices with HTTP. Business microservices shouldn't be coupled with other business microservices. Ideally, they shouldn't "know" about the existence of other microservices. If your application relies on coupling microservices as in the example, achieving autonomy per microservice will be almost impossible.
- Failure in any one microservice. If you implemented a chain of microservices linked by HTTP calls, when any of the microservices fails (and eventually they will fail) the whole chain of microservices will fail. A microservice-based system should be designed to continue to work as well as possible during partial failures. Even if you implement client logic that uses retries with exponential backoff or circuit breaker mechanisms, the more complex the HTTP call chains are, the more complex it is to implement a failure strategy based on HTTP.

In fact, if your internal microservices are communicating by creating chains of HTTP requests as described, it could be argued that you have a monolithic application, but one based on HTTP between processes instead of intra-process communication mechanisms.

Therefore, in order to enforce microservice autonomy and have better resiliency, you should minimize the use of chains of request/response communication across microservices. It's recommended that you use only asynchronous interaction for inter-microservice communication, either by using asynchronous message- and event-based communication, or by using (asynchronous) HTTP polling independently of the original HTTP request/response cycle.

The use of asynchronous communication is explained with additional details later in this guide in the sections Asynchronous microservice integration enforces microservice's autonomy and Asynchronous message-based communication.

Additional resources

- **CAP theorem**
  https://en.wikipedia.org/wiki/CAP_theorem
- **Eventual consistency**
  https://en.wikipedia.org/wiki/Eventual_consistency
- **Data Consistency Primer**
  https://learn.microsoft.com/previous-versions/msp-n-p/dn589800(v=pandp.10)
- **Martin Fowler. CQRS (Command and Query Responsibility Segregation)**
  https://martinfowler.com/bliki/CQRS.html
- **Materialized View**
  https://learn.microsoft.com/azure/architecture/patterns/materialized-view
- **Charles Row. ACID vs. BASE: The Shifting pH of Database Transaction Processing**
  https://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/
- **Compensating Transaction**
  https://learn.microsoft.com/azure/architecture/patterns/compensating-transaction
- **Udi Dahan. Service Oriented Composition**
  https://udidahan.com/2014/07/30/service-oriented-composition-with-video/

Identify domain-model boundaries for each microservice

The goal when identifying model boundaries and size for each microservice isn't to get to the most granular separation possible, although you should tend toward small microservices if possible. Instead, your goal should be to get to the most meaningful separation guided by your domain knowledge. The emphasis isn't on the size, but instead on business capabilities. In addition, if there's clear cohesion needed for a certain area of the application based on a high number of dependencies, that indicates the need for a single microservice, too. Cohesion is a way to identify how to break apart

or group together microservices. Ultimately, while you gain more knowledge about the domain, you should adapt the size of your microservice, iteratively. Finding the right size isn't a one-shot process.

Sam Newman, a recognized promoter of microservices and author of the book Building Microservices, highlights that you should design your microservices based on the Bounded Context (BC) pattern (part of domain-driven design), as introduced earlier. Sometimes, a BC could be composed of several physical services, but not vice versa.

A domain model with specific domain entities applies within a concrete BC or microservice. A BC delimits the applicability of a domain model and gives developer team members a clear and shared understanding of what must be cohesive and what can be developed independently. These are the same goals for microservices.

Another tool that informs your design choice is Conway's law, which states that an application will reflect the social boundaries of the organization that produced it. But sometimes the opposite is true - the company's organization is formed by the software. You might need to reverse Conway's law and build the boundaries the way you want the company to be organized, leaning toward business process consulting.

To identify bounded contexts, you can use a DDD pattern called the Context Mapping pattern. With Context Mapping, you identify the various contexts in the application and their boundaries. It's common to have a different context and boundary for each small subsystem, for instance. The Context Map is a way to define and make explicit those boundaries between domains. A BC is autonomous and includes the details of a single domain -details like the domain entities- and defines integration contracts with other BCs. This is similar to the definition of a microservice: it's autonomous, it implements certain domain capability, and it must provide interfaces. This is why Context Mapping and the Bounded Context pattern are good approaches for identifying the domain model boundaries of your microservices.

When designing a large application, you'll see how its domain model can be fragmented - a domain expert from the catalog domain will name entities differently in the catalog and inventory domains than a shipping domain expert, for instance. Or the user domain entity might be different in size and number of attributes when dealing with a CRM expert who wants to store every detail about the customer than for an ordering domain expert who just needs partial data about the customer. It's very hard to disambiguate all domain terms across all the domains related to a large application. But the most important thing is that you shouldn't try to unify the terms. Instead, accept the differences and richness provided by each domain. If you try to have a unified database for the whole application, attempts at a unified vocabulary will be awkward and won't sound right to any of the multiple domain experts. Therefore, BCs (implemented as microservices) will help you to clarify where you can use certain domain terms and where you'll need to split the system and create additional BCs with different domains.

You'll know that you got the right boundaries and sizes of each BC and domain model if you have few strong relationships between domain models, and you do not usually need to merge information from multiple domain models when performing typical application operations.

Perhaps the best answer to the question of how large a domain model for each microservice should be is the following: it should have an autonomous BC, as isolated as possible, that enables you to work without having to constantly switch to other contexts (other microservice's models). In Figure 4-

10, you can see how multiple microservices (multiple BCs) each has their own model and how their entities can be defined, depending on the specific requirements for each of the identified domains in your application.

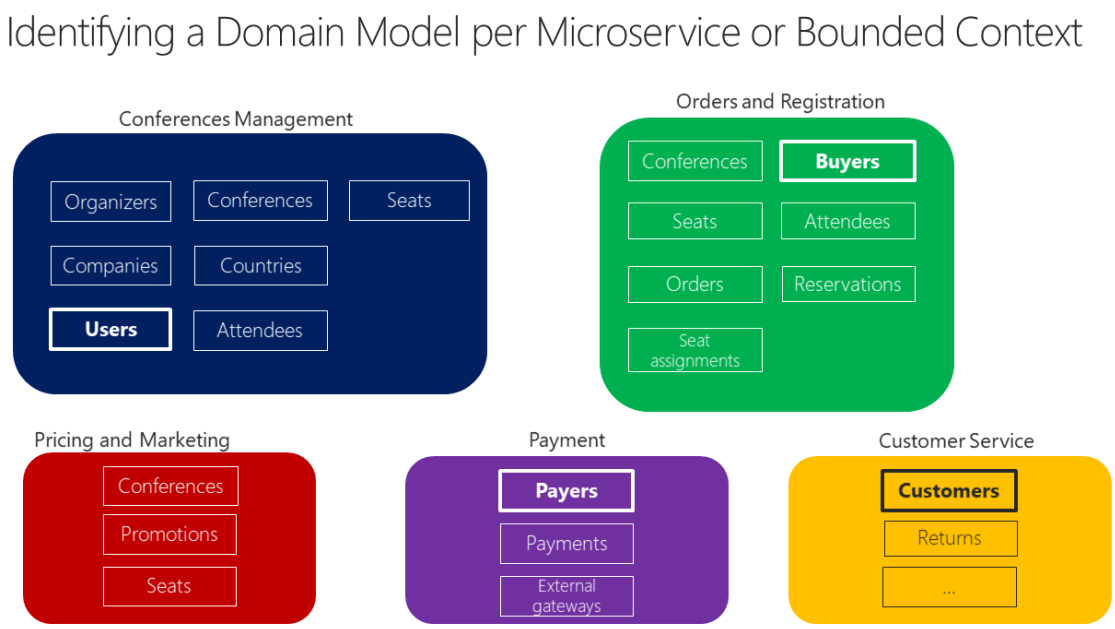


Figure 4-10. Identifying entities and microservice model boundaries

Figure 4-10 illustrates a sample scenario related to an online conference management system. The same entity appears as "Users", "Buyers", "Payers", and "Customers" depending on the bounded context. You've identified several BCs that could be implemented as microservices, based on domains that domain experts defined for you. As you can see, there are entities that are present just in a single microservice model, like Payments in the Payment microservice. Those will be easy to implement.

However, you might also have entities that have a different shape but share the same identity across the multiple domain models from the multiple microservices. For example, the User entity is identified in the Conferences Management microservice. That same user, with the same identity, is the one named Buyers in the Ordering microservice, or the one named Payer in the Payment microservice, and even the one named Customer in the Customer Service microservice. This is because, depending on the ubiquitous language that each domain expert is using, a user might have a different perspective even with different attributes. The user entity in the microservice model named Conferences Management might have most of its personal data attributes. However, that same user in the shape of Payer in the microservice Payment or in the shape of Customer in the microservice Customer Service might not need the same list of attributes.

A similar approach is illustrated in Figure 4-11.

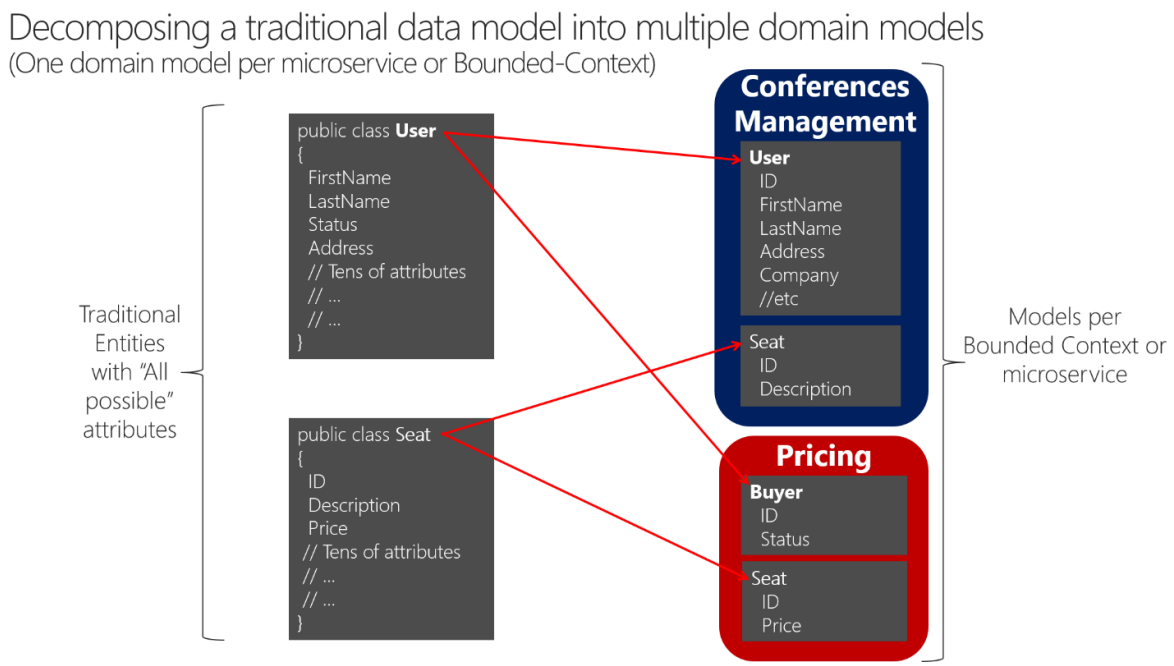


Figure 4-11. Decomposing traditional data models into multiple domain models

When decomposing a traditional data model between bounded contexts, you can have different entities that share the same identity (a buyer is also a user) with different attributes in each bounded context. You can see how the user is present in the Conferences Management microservice model as the User entity and is also present in the form of the Buyer entity in the Pricing microservice, with alternate attributes or details about the user when it's actually a buyer. Each microservice or BC might not need all the data related to a User entity, just part of it, depending on the problem to solve or the context. For instance, in the Pricing microservice model, you do not need the address or the name of the user, just the ID (as identity) and Status, which will have an impact on discounts when pricing the seats per buyer.

The Seat entity has the same name but different attributes in each domain model. However, Seat shares identity based on the same ID, as happens with User and Buyer.

Basically, there's a shared concept of a user that exists in multiple services (domains), which all share the identity of that user. But in each domain model there might be additional or different details about the user entity. Therefore, there needs to be a way to map a user entity from one domain (microservice) to another.

There are several benefits to not sharing the same user entity with the same number of attributes across domains. One benefit is to reduce duplication, so that microservice models do not have any data that they do not need. Another benefit is having a primary microservice that owns a certain type of data per entity so that updates and queries for that type of data are driven only by that microservice.

The API gateway pattern versus the Direct client-to-microservice communication

In a microservices architecture, each microservice exposes a set of (typically) fine-grained endpoints. This fact can impact the client-to-microservice communication, as explained in this section.

Direct client-to-microservice communication

A possible approach is to use a direct client-to-microservice communication architecture. In this approach, a client app can make requests directly to some of the microservices, as shown in Figure 4-12.

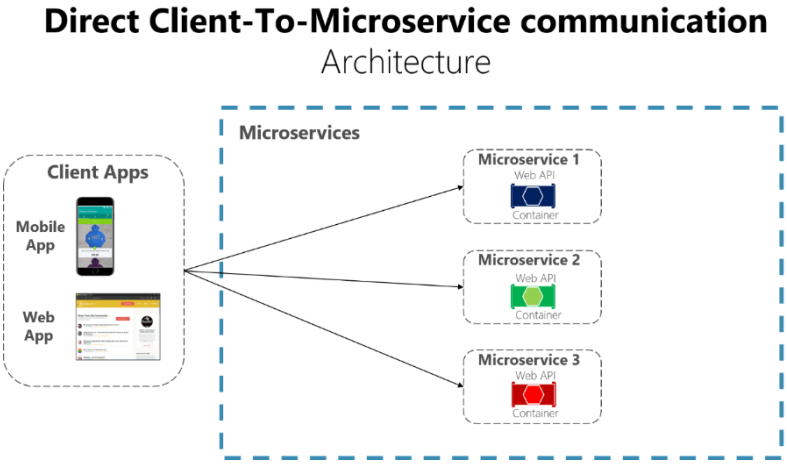


Figure 4-12. Using a direct client-to-microservice communication architecture

In this approach, each microservice has a public endpoint, sometimes with a different TCP port for each microservice. An example of a URL for a particular service could be the following URL in Azure:

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

In a production environment based on a cluster, that URL would map to the load balancer used in the cluster, which in turn distributes the requests across the microservices. In production environments, you could have an Application Delivery Controller (ADC) like Azure Application Gateway between your microservices and the Internet. This layer acts as a transparent tier that not only performs load balancing, but secures your services by offering SSL termination. This approach improves the load of your hosts by offloading CPU-intensive SSL termination and other routing duties to the Azure Application Gateway. In any case, a load balancer and ADC are transparent from a logical application architecture point of view.

A direct client-to-microservice communication architecture could be good enough for a small microservice-based application, especially if the client app is a server-side web application like an ASP.NET MVC app. However, when you build large and complex microservice-based applications (for example, when handling dozens of microservice types), and especially when the client apps are remote mobile apps or SPA web applications, that approach faces a few issues.

Consider the following questions when developing a large application based on microservices:

- *How can client apps minimize the number of requests to the back end and reduce chatty communication to multiple microservices?*

Interacting with multiple microservices to build a single UI screen increases the number of round trips across the Internet. This approach increases latency and complexity on the UI side. Ideally, responses should be efficiently aggregated in the server side. This approach reduces latency, since multiple pieces of data come back in parallel and some UI can show data as soon as it's ready.

- *How can you handle cross-cutting concerns such as authorization, data transformations, and dynamic request dispatching?*

Implementing security and cross-cutting concerns like security and authorization on every microservice can require significant development effort. A possible approach is to have those services within the Docker host or internal cluster to restrict direct access to them from the outside, and to implement those cross-cutting concerns in a centralized place, like an API Gateway.

- *How can client apps communicate with services that use non-Internet-friendly protocols?*

Protocols used on the server side (like AMQP or binary protocols) are not supported in client apps. Therefore, requests must be performed through protocols like HTTP/HTTPS and translated to the other protocols afterwards. A *man-in-the-middle* approach can help in this situation.

- *How can you shape a facade especially made for mobile apps?*

The API of multiple microservices might not be well designed for the needs of different client applications. For instance, the needs of a mobile app might be different than the needs of a web app. For mobile apps, you might need to optimize even further so that data responses can be more efficient. You might do this functionality by aggregating data from multiple microservices and returning a single set of data, and sometimes eliminating any data in the response that isn't needed by the mobile app. And, of course, you might compress that data. Again, a facade or API in between the mobile app and the microservices can be convenient for this scenario.

Why consider API Gateways instead of direct client-to-microservice communication

In a microservices architecture, the client apps usually need to consume functionality from more than one microservice. If that consumption is performed directly, the client needs to handle multiple calls to microservice endpoints. What happens when the application evolves and new microservices are introduced or existing microservices are updated? If your application has many microservices, handling so many endpoints from the client apps can be a nightmare. Since the client app would be coupled to those internal endpoints, evolving the microservices in the future can cause high impact for the client apps.

Therefore, having an intermediate level or tier of indirection (Gateway) can be convenient for microservice-based applications. If you don't have API Gateways, the client apps must send requests directly to the microservices and that raises problems, such as the following issues:

- **Coupling**: Without the API Gateway pattern, the client apps are coupled to the internal microservices. The client apps need to know how the multiple areas of the application are decomposed in microservices. When evolving and refactoring the internal microservices, those actions impact maintenance because they cause breaking changes to the client apps due to the direct reference to the internal microservices from the client apps. Client apps need to be updated frequently, making the solution harder to evolve.
- **Too many round trips**: A single page/screen in the client app might require several calls to multiple services. That approach can result in multiple network round trips between the client and the server, adding significant latency. Aggregation handled in an intermediate level could improve the performance and user experience for the client app.
- **Security issues**: Without a gateway, all the microservices must be exposed to the "external world", making the attack surface larger than if you hide internal microservices that aren't directly used by the client apps. The smaller the attack surface is, the more secure your application can be.
- **Cross-cutting concerns**: Each publicly published microservice must handle concerns such as authorization and SSL. In many situations, those concerns could be handled in a single tier so the internal microservices are simplified.

What is the API Gateway pattern?

When you design and build large or complex microservice-based applications with multiple client apps, a good approach to consider can be an API Gateway. This pattern is a service that provides a single-entry point for certain groups of microservices. It's similar to the Facade pattern from object-oriented design, but in this case, it's part of a distributed system. The API Gateway pattern is also sometimes known as the "backend for frontend" (BFF) because you build it while thinking about the needs of the client app.

Therefore, the API gateway sits between the client apps and the microservices. It acts as a reverse proxy, routing requests from clients to services. It can also provide other cross-cutting features such as authentication, SSL termination, and cache.

Figure 4-13 shows how a custom API Gateway can fit into a simplified microservice-based architecture with just a few microservices.

Using a single custom **API Gateway service**

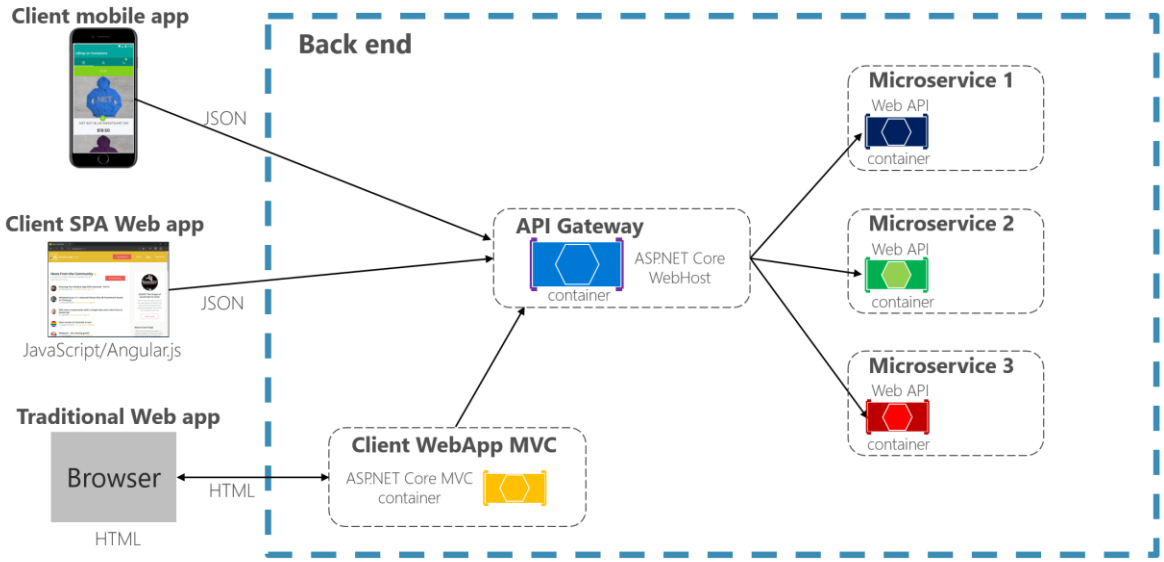


Figure 4-13. Using an API Gateway implemented as a custom service

Apps connect to a single endpoint, the API Gateway, that's configured to forward requests to individual microservices. In this example, the API Gateway would be implemented as a custom ASP.NET Core WebHost service running as a container.

It's important to highlight that in that diagram, you would be using a single custom API Gateway service facing multiple and different client apps. That fact can be an important risk because your API Gateway service will be growing and evolving based on many different requirements from the client apps. Eventually, it will be bloated because of those different needs and effectively it could be similar to a monolithic application or monolithic service. That's why it's very much recommended to split the API Gateway in multiple services or multiple smaller API Gateways, one per client app form-factor type, for instance.

You need to be careful when implementing the API Gateway pattern. Usually it isn't a good idea to have a single API Gateway aggregating all the internal microservices of your application. If it does, it acts as a monolithic aggregator or orchestrator and violates microservice autonomy by coupling all the microservices.

Therefore, the API Gateways should be segregated based on business boundaries and the client apps and not act as a single aggregator for all the internal microservices.

When splitting the API Gateway tier into multiple API Gateways, if your application has multiple client apps, that can be a primary pivot when identifying the multiple API Gateways types, so that you can have a different facade for the needs of each client app. This case is a pattern named "Backend for Frontend" (BFF) where each API Gateway can provide a different API tailored for each client app type, possibly even based on the client form factor by implementing specific adapter code which underneath calls multiple internal microservices, as shown in the following image:

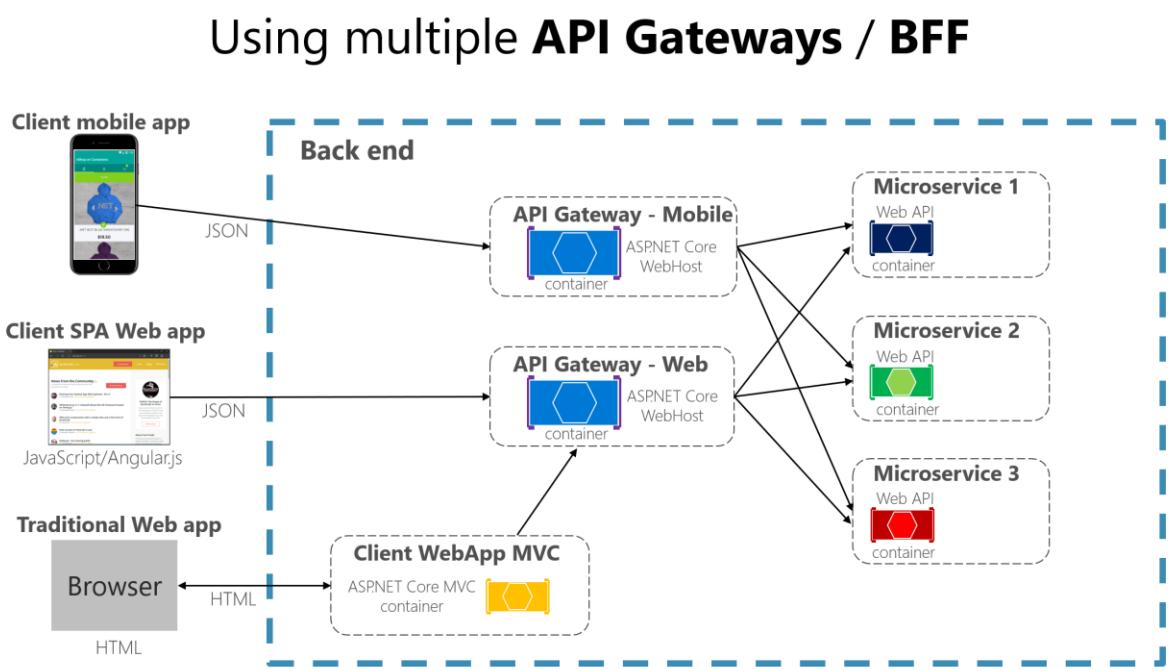


Figure 4-13.1. Using multiple custom API Gateways

Figure 4-13.1 shows API Gateways that are segregated by client type; one for mobile clients and one for web clients. A traditional web app connects to an MVC microservice that uses the web API Gateway. The example depicts a simplified architecture with multiple fine-grained API Gateways. In this case, the boundaries identified for each API Gateway are based purely on the "Backend for Frontend" (BFF) pattern, hence based just on the API needed per client app. But in larger applications you should also go further and create other API Gateways based on business boundaries as a second design pivot.

Main features in the API Gateway pattern

An API Gateway can offer multiple features. Depending on the product it might offer richer or simpler features, however, the most important and foundational features for any API Gateway are the following design patterns:

Reverse proxy or gateway routing. The API Gateway offers a reverse proxy to redirect or route requests (layer 7 routing, usually HTTP requests) to the endpoints of the internal microservices. The gateway provides a single endpoint or URL for the client apps and then internally maps the requests to a group of internal microservices. This routing feature helps to decouple the client apps from the microservices but it's also convenient when modernizing a monolithic API by sitting the API Gateway in between the monolithic API and the client apps, then you can add new APIs as new microservices while still using the legacy monolithic API until it's split into many microservices in the future. Because of the API Gateway, the client apps won't notice if the APIs being used are implemented as internal microservices or a monolithic API and more importantly, when evolving and refactoring the monolithic API into microservices, thanks to the API Gateway routing, client apps won't be impacted with any URI change.

For more information, see Gateway routing pattern.

Requests aggregation. As part of the gateway pattern you can aggregate multiple client requests (usually HTTP requests) targeting multiple internal microservices into a single client request. This pattern is especially convenient when a client page/screen needs information from several microservices. With this approach, the client app sends a single request to the API Gateway that dispatches several requests to the internal microservices and then aggregates the results and sends everything back to the client app. The main benefit and goal of this design pattern is to reduce chattiness between the client apps and the backend API, which is especially important for remote apps out of the datacenter where the microservices live, like mobile apps or requests coming from SPA apps that come from JavaScript in client remote browsers. For regular web apps performing the requests in the server environment (like an ASP.NET Core MVC web app), this pattern is not so important as the latency is very much smaller than for remote client apps.

Depending on the API Gateway product you use, it might be able to perform this aggregation. However, in many cases it's more flexible to create aggregation microservices under the scope of the API Gateway, so you define the aggregation in code (that is, C# code):

For more information, see Gateway aggregation pattern.

Cross-cutting concerns or gateway offloading. Depending on the features offered by each API Gateway product, you can offload functionality from individual microservices to the gateway, which simplifies the implementation of each microservice by consolidating cross-cutting concerns into one tier. This approach is especially convenient for specialized features that can be complex to implement properly in every internal microservice, such as the following functionality:

- Authentication and authorization
- Service discovery integration
- Response caching
- Retry policies, circuit breaker, and QoS
- Rate limiting and throttling
- Load balancing
- Logging, tracing, correlation
- Headers, query strings, and claims transformation
- IP allowlisting

For more information, see Gateway offloading pattern.

Using products with API Gateway features

There can be many more cross-cutting concerns offered by the API Gateways products depending on each implementation. We'll explore here:

- Azure API Management
- Ocelot

Azure API Management

Azure API Management (as shown in Figure 4-14) not only solves your API Gateway needs but provides features like gathering insights from your APIs. If you're using an API management solution, an API Gateway is only a component within that full API management solution.

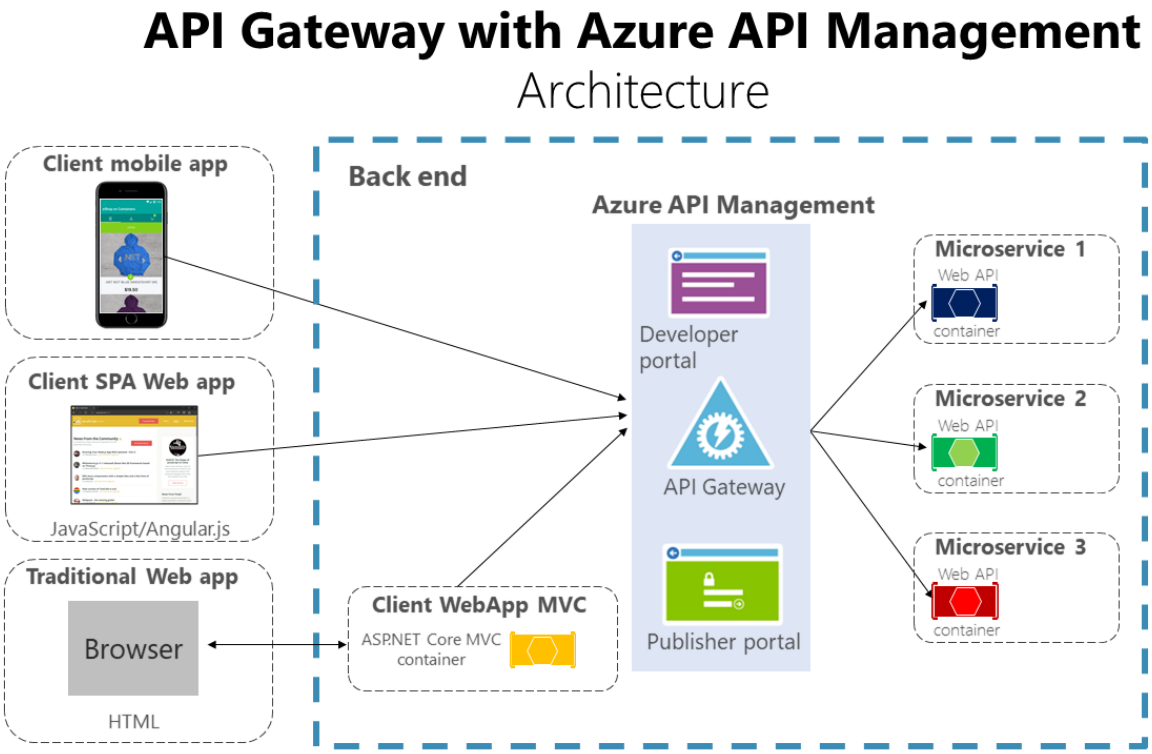


Figure 4-14. Using Azure API Management for your API Gateway

Azure API Management solves both your API Gateway and Management needs like logging, security, metering, etc. In this case, when using a product like Azure API Management, the fact that you might have a single API Gateway is not so risky because these kinds of API Gateways are "thinner", meaning that you don't implement custom C# code that could evolve towards a monolithic component.

The API Gateway products usually act like a reverse proxy for ingress communication, where you can also filter the APIs from the internal microservices plus apply authorization to the published APIs in this single tier.

The insights available from an API Management system help you get an understanding of how your APIs are being used and how they are performing. They do this activity by letting you view near real-time analytics reports and identifying trends that might impact your business. Plus, you can have logs about request and response activity for further online and offline analysis.

With Azure API Management, you can secure your APIs using a key, a token, and IP filtering. These features let you enforce flexible and fine-grained quotas and rate limits, modify the shape and behavior of your APIs using policies, and improve performance with response caching.

In this guide and the reference sample application (eShopOnContainers), the architecture is limited to a simpler and custom-made containerized architecture in order to focus on plain containers without

using PaaS products like Azure API Management. But for large microservice-based applications that are deployed into Microsoft Azure, we encourage you to evaluate Azure API Management as the base for your API Gateways in production.

Ocelot

Ocelot is a lightweight API Gateway, recommended for simpler approaches. Ocelot is an Open Source .NET Core-based API Gateway especially made for microservices architectures that need unified points of entry into their systems. It's lightweight, fast, and scalable and provides routing and authentication among many other features.

The main reason to choose Ocelot for the eShopOnContainers reference application 2.0 is because Ocelot is a .NET Core lightweight API Gateway that you can deploy into the same application deployment environment where you're deploying your microservices/containers, such as a Docker Host, Kubernetes, etc. And since it's based on .NET Core, it's cross-platform allowing you to deploy on Linux or Windows.

The previous diagrams showing custom API Gateways running in containers are precisely how you can also run Ocelot in a container and microservice-based application.

In addition, there are many other products in the market offering API Gateways features, such as Apigee, Kong, MuleSoft, WSO2, and other products like Linkerd and Istio for service mesh ingress controller features.

After the initial architecture and patterns explanation sections, the next sections explain how to implement API Gateways with Ocelot.

Drawbacks of the API Gateway pattern

- The most important drawback is that when you implement an API Gateway, you're coupling that tier with the internal microservices. Coupling like this might introduce serious difficulties for your application. Clemens Vaster, architect at the Azure Service Bus team, refers to this potential difficulty as "the new ESB" in the "Messaging and Microservices" session at GOTO 2016.
- Using a microservices API Gateway creates an additional possible single point of failure.
- An API Gateway can introduce increased response time due to the additional network call. However, this extra call usually has less impact than having a client interface that's too chatty directly calling the internal microservices.
- If not scaled out properly, the API Gateway can become a bottleneck.
- An API Gateway requires additional development cost and future maintenance if it includes custom logic and data aggregation. Developers must update the API Gateway in order to expose each microservice's endpoints. Moreover, implementation changes in the internal microservices might cause code changes at the API Gateway level. However, if the API Gateway is just applying security, logging, and versioning (as when using Azure API Management), this additional development cost might not apply.

- If the API Gateway is developed by a single team, there can be a development bottleneck. This aspect is another reason why a better approach is to have several fined-grained API Gateways that respond to different client needs. You could also segregate the API Gateway internally into multiple areas or layers that are owned by the different teams working on the internal microservices.

Additional resources

- **Chris Richardson. Pattern: API Gateway / Backend for Front-End**
  https://microservices.io/patterns/apigateway.html
- **API Gateway pattern**
  https://learn.microsoft.com/azure/architecture/microservices/gateway
- **Aggregation and composition pattern**
  https://microservices.io/patterns/data/api-composition.html
- **Azure API Management**
  https://azure.microsoft.com/services/api-management/
- **Udi Dahan. Service Oriented Composition**
  https://udidahan.com/2014/07/30/service-oriented-composition-with-video/
- **Clemens Vasters. Messaging and Microservices at GOTO 2016 (video)**
  https://www.youtube.com/watch?v=rXi5CLjIQ9k
- **API Gateway in a Nutshell** (ASP.NET Core API Gateway Tutorial Series)
  https://www.pogsdotnet.com/2018/08/api-gateway-in-nutshell.html

Communication in a microservice architecture

In a monolithic application running on a single process, components invoke one another using language-level method or function calls. These can be strongly coupled if you're creating objects with code (for example, new ClassName()), or can be invoked in a decoupled way if you're using Dependency Injection by referencing abstractions rather than concrete object instances. Either way, the objects are running within the same process. The biggest challenge when changing from a monolithic application to a microservices-based application lies in changing the communication mechanism. A direct conversion from in-process method calls into RPC calls to services will cause a chatty and not efficient communication that won't perform well in distributed environments. The challenges of designing distributed system properly are well enough known that there's even a canon known as the Fallacies of distributed computing that lists assumptions that developers often make when moving from monolithic to distributed designs.

There isn't one solution, but several. One solution involves isolating the business microservices as much as possible. You then use asynchronous communication between the internal microservices and replace fine-grained communication that's typical in intra-process communication between objects with coarser-grained communication. You can do this by grouping calls, and by returning data that aggregates the results of multiple internal calls, to the client.

A microservices-based application is a distributed system running on multiple processes or services, usually even across multiple servers or hosts. Each service instance is typically a process. Therefore, services must interact using an inter-process communication protocol such as HTTP, AMQP, or a binary protocol like TCP, depending on the nature of each service.

The microservice community promotes the philosophy of "smart endpoints and dumb pipes". This slogan encourages a design that's as decoupled as possible between microservices, and as cohesive as possible within a single microservice. As explained earlier, each microservice owns its own data and its own domain logic. But the microservices composing an end-to-end application are usually simply choreographed by using REST communications rather than complex protocols such as WS-* and flexible event-driven communications instead of centralized business-process-orchestrators.

The two commonly used protocols are HTTP request/response with resource APIs (when querying most of all), and lightweight asynchronous messaging when communicating updates across multiple microservices. These are explained in more detail in the following sections.

Communication types

Client and services can communicate through many different types of communication, each one targeting a different scenario and goals. Initially, those types of communications can be classified in two axes.

The first axis defines if the protocol is synchronous or asynchronous:

- Synchronous protocol. HTTP is a synchronous protocol. The client sends a request and waits for a response from the service. That's independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread isn't blocked, and the response will reach a callback eventually). The important point here is that the protocol (HTTP/HTTPS) is synchronous and the client code can only continue its task when it receives the HTTP server response.
- Asynchronous protocol. Other protocols like AMQP (a protocol supported by many operating systems and cloud environments) use asynchronous messages. The client code or message sender usually doesn't wait for a response. It just sends the message as when sending a message to a RabbitMQ queue or any other message broker.

The second axis defines if the communication has a single receiver or multiple receivers:

- Single receiver. Each request must be processed by exactly one receiver or service. An example of this communication is the Command pattern.
- Multiple receivers. Each request can be processed by zero to multiple receivers. This type of communication must be asynchronous. An example is the publish/subscribe mechanism used in patterns like Event-driven architecture. This is based on an event-bus interface or message broker when propagating data updates between multiple microservices through events; it's usually implemented through a service bus or similar artifact like Azure Service Bus by using topics and subscriptions.

A microservice-based application will often use a combination of these communication styles. The most common type is single-receiver communication with a synchronous protocol like HTTP/HTTPS when invoking a regular Web API HTTP service. Microservices also typically use messaging protocols for asynchronous communication between microservices.

These axes are good to know so you have clarity on the possible communication mechanisms, but they're not the important concerns when building microservices. Neither the asynchronous nature of client thread execution nor the asynchronous nature of the selected protocol are the important points when integrating microservices. What *is* important is being able to integrate your microservices asynchronously while maintaining the independence of microservices, as explained in the following section.

Asynchronous microservice integration enforces microservice's autonomy

As mentioned, the important point when building a microservices-based application is the way you integrate your microservices. Ideally, you should try to minimize the communication between the internal microservices. The fewer communications between microservices, the better. But in many cases, you'll have to somehow integrate the microservices. When you need to do that, the critical rule here is that the communication between the microservices should be asynchronous. That doesn't mean that you have to use a specific protocol (for example, asynchronous messaging versus synchronous HTTP). It just means that the communication between microservices should be done only by propagating data asynchronously, but try not to depend on other internal microservices as part of the initial service's HTTP request/response operation.

If possible, never depend on synchronous communication (request/response) between multiple microservices, not even for queries. The goal of each microservice is to be autonomous and available to the client consumer, even if the other services that are part of the end-to-end application are down or unhealthy. If you think you need to make a call from one microservice to other microservices (like performing an HTTP request for a data query) to be able to provide a response to a client application, you have an architecture that won't be resilient when some microservices fail.

Moreover, having HTTP dependencies between microservices, like when creating long request/response cycles with HTTP request chains, as shown in the first part of the Figure 4-15, not only makes your microservices not autonomous but also their performance is impacted as soon as one of the services in that chain isn't performing well.

The more you add synchronous dependencies between microservices, such as query requests, the worse the overall response time gets for the client apps.

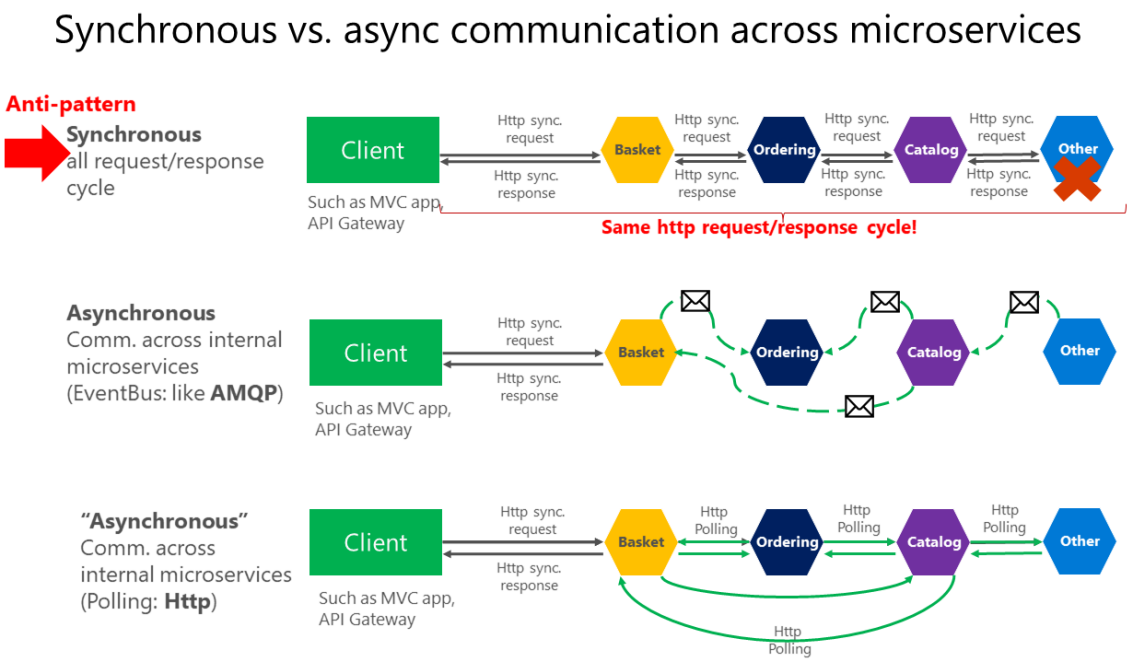


Figure 4-15. Anti-patterns and patterns in communication between microservices

As shown in the above diagram, in synchronous communication a "chain" of requests is created between microservices while serving the client request. This is an anti-pattern. In asynchronous communication microservices use asynchronous messages or http polling to communicate with other microservices, but the client request is served right away.

If your microservice needs to raise an additional action in another microservice, if possible, do not perform that action synchronously and as part of the original microservice request and reply operation. Instead, do it asynchronously (using asynchronous messaging or integration events, queues, etc.). But, as much as possible, do not invoke the action synchronously as part of the original synchronous request and reply operation.

And finally (and this is where most of the issues arise when building microservices), if your initial microservice needs data that's originally owned by other microservices, do not rely on making synchronous requests for that data. Instead, replicate or propagate that data (only the attributes you need) into the initial service's database by using eventual consistency (typically by using integration events, as explained in upcoming sections).

As noted earlier in the Identifying domain-model boundaries for each microservice section, duplicating some data across several microservices isn't an incorrect design—on the contrary, when doing that you can translate the data into the specific language or terms of that additional domain or Bounded Context. For instance, in the eShopOnContainers application you have a microservice named identity-api that's in charge of most of the user's data with an entity named User. However, when you need to store data about the user within the Ordering microservice, you store it as a different entity named Buyer. The Buyer entity shares the same identity with the original User entity, but it might have only the few attributes needed by the Ordering domain, and not the whole user profile.

You might use any protocol to communicate and propagate data asynchronously across microservices in order to have eventual consistency. As mentioned, you could use integration events using an event bus or message broker or you could even use HTTP by polling the other services instead. It doesn't matter. The important rule is to not create synchronous dependencies between your microservices.

The following sections explain the multiple communication styles you can consider using in a microservice-based application.

Communication styles

There are many protocols and choices you can use for communication, depending on the communication type you want to use. If you're using a synchronous request/response-based communication mechanism, protocols such as HTTP and REST approaches are the most common, especially if you're publishing your services outside the Docker host or microservice cluster. If you're communicating between services internally (within your Docker host or microservices cluster), you might also want to use binary format communication mechanisms (like WCF using TCP and binary format). Alternatively, you can use asynchronous, message-based communication mechanisms such as AMQP.

There are also multiple message formats like JSON or XML, or even binary formats, which can be more efficient. If your chosen binary format isn't a standard, it's probably not a good idea to publicly publish your services using that format. You could use a non-standard format for internal communication between your microservices. You might do this when communicating between microservices within your Docker host or microservice cluster (for example, Docker orchestrators), or for proprietary client applications that talk to the microservices.

Request/response communication with HTTP and REST

When a client uses request/response communication, it sends a request to a service, then the service processes the request and sends back a response. Request/response communication is especially well suited for querying data for a real-time UI (a live user interface) from client apps. Therefore, in a microservice architecture you'll probably use this communication mechanism for most queries, as shown in Figure 4-16.

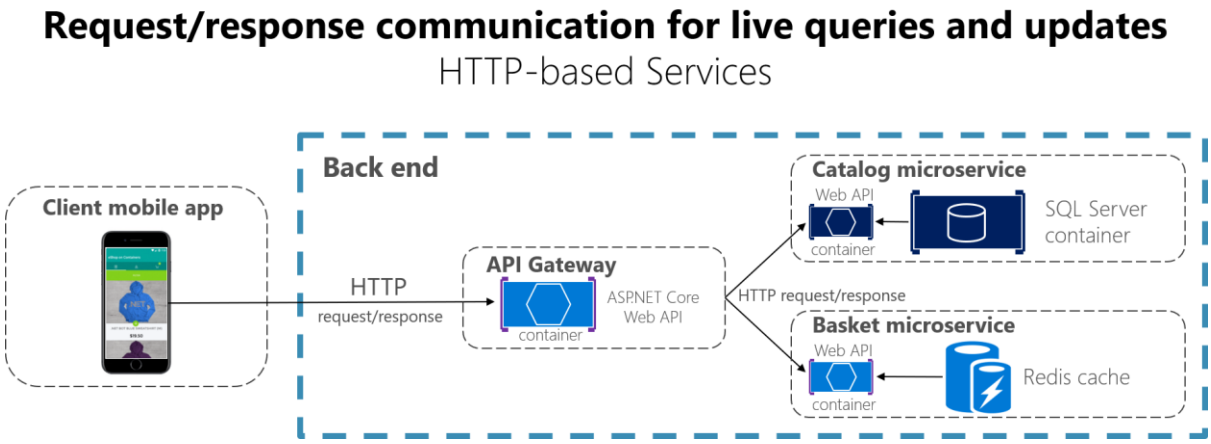


Figure 4-16. Using HTTP request/response communication (synchronous or asynchronous)

When a client uses request/response communication, it assumes that the response will arrive in a short time, typically less than a second, or a few seconds at most. For delayed responses, you need to implement asynchronous communication based on messaging patterns and messaging technologies, which is a different approach that we explain in the next section.

A popular architectural style for request/response communication is REST. This approach is based on, and tightly coupled to, the HTTP protocol, embracing HTTP verbs like GET, POST, and PUT. REST is the most commonly used architectural communication approach when creating services. You can implement REST services when you develop ASP.NET Core Web API services.

There's additional value when using HTTP REST services as your interface definition language. For instance, if you use Swagger metadata to describe your service API, you can use tools that generate client stubs that can directly discover and consume your services.

Additional resources

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

Push and real-time communication based on HTTP

Another possibility (usually for different purposes than REST) is a real-time and one-to-many communication with higher-level frameworks such as ASP.NET SignalR and protocols such as WebSockets.

As Figure 4-17 shows, real-time HTTP communication means that you can have server code pushing content to connected clients as the data becomes available, rather than having the server wait for a client to request new data.

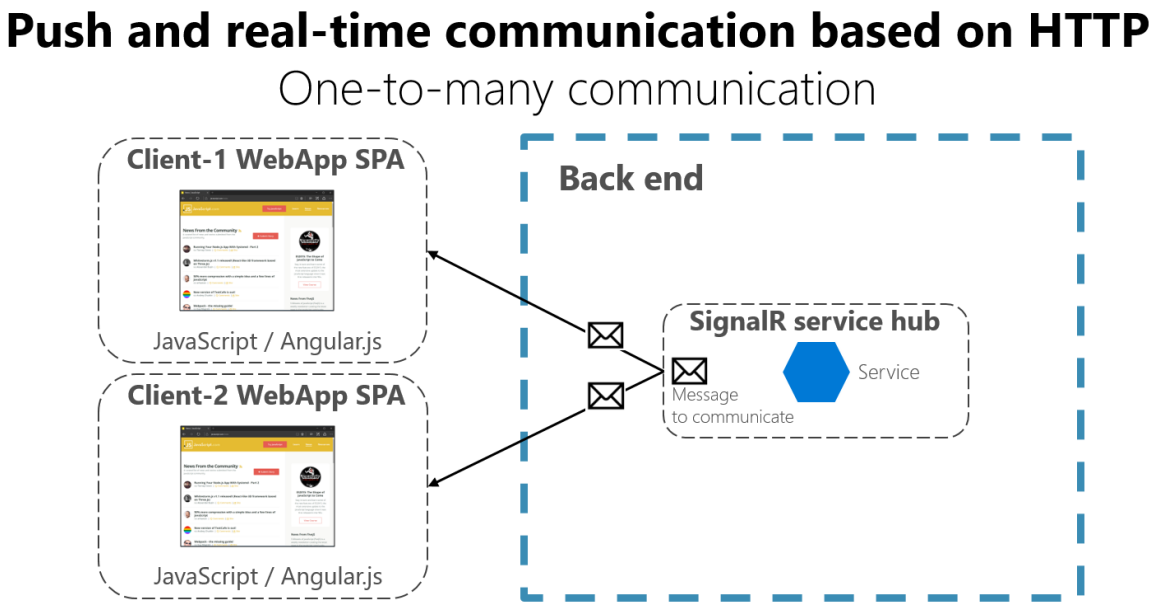


Figure 4-17. One-to-many real-time asynchronous message communication

SignalR is a good way to achieve real-time communication for pushing content to the clients from a back-end server. Since communication is in real time, client apps show the changes almost instantly. This is usually handled by a protocol such as WebSockets, using many WebSockets connections (one per client). A typical example is when a service communicates a change in the score of a sports game to many client web apps simultaneously.

Asynchronous message-based communication

Asynchronous messaging and event-driven communication are critical when propagating changes across multiple microservices and their related domain models. As mentioned earlier in the discussion microservices and Bounded Contexts (BCs), models (User, Customer, Product, Account, etc.) can mean different things to different microservices or BCs. That means that when changes occur, you need some way to reconcile changes across the different models. A solution is eventual consistency and event-driven communication based on asynchronous messaging.

When using messaging, processes communicate by exchanging messages asynchronously. A client makes a command or a request to a service by sending it a message. If the service needs to reply, it sends a different message back to the client. Since it's a message-based communication, the client assumes that the reply won't be received immediately, and that there might be no response at all.

A message is composed by a header (metadata such as identification or security information) and a body. Messages are usually sent through asynchronous protocols like AMQP.

The preferred infrastructure for this type of communication in the microservices community is a lightweight message broker, which is different than the large brokers and orchestrators used in SOA. In a lightweight message broker, the infrastructure is typically "dumb," acting only as a message broker, with simple implementations such as RabbitMQ or a scalable service bus in the cloud like

Azure Service Bus. In this scenario, most of the "smart" thinking still lives in the endpoints that are producing and consuming messages-that is, in the microservices.

Another rule you should try to follow, as much as possible, is to use only asynchronous messaging between the internal services, and to use synchronous communication (such as HTTP) only from the client apps to the front-end services (API Gateways plus the first level of microservices).

There are two kinds of asynchronous messaging communication: single receiver message-based communication, and multiple receivers message-based communication. The following sections provide details about them.

Single-receiver message-based communication

Message-based asynchronous communication with a single receiver means there's point-to-point communication that delivers a message to exactly one of the consumers that's reading from the channel, and that the message is processed just once. However, there are special situations. For instance, in a cloud system that tries to automatically recover from failures, the same message could be sent multiple times. Due to network or other failures, the client has to be able to retry sending messages, and the server has to implement an operation to be idempotent in order to process a particular message just once.

Single-receiver message-based communication is especially well suited for sending asynchronous commands from one microservice to another as shown in Figure 4-18 that illustrates this approach.

Once you start sending message-based communication (either with commands or events), you should avoid mixing message-based communication with synchronous HTTP communication.

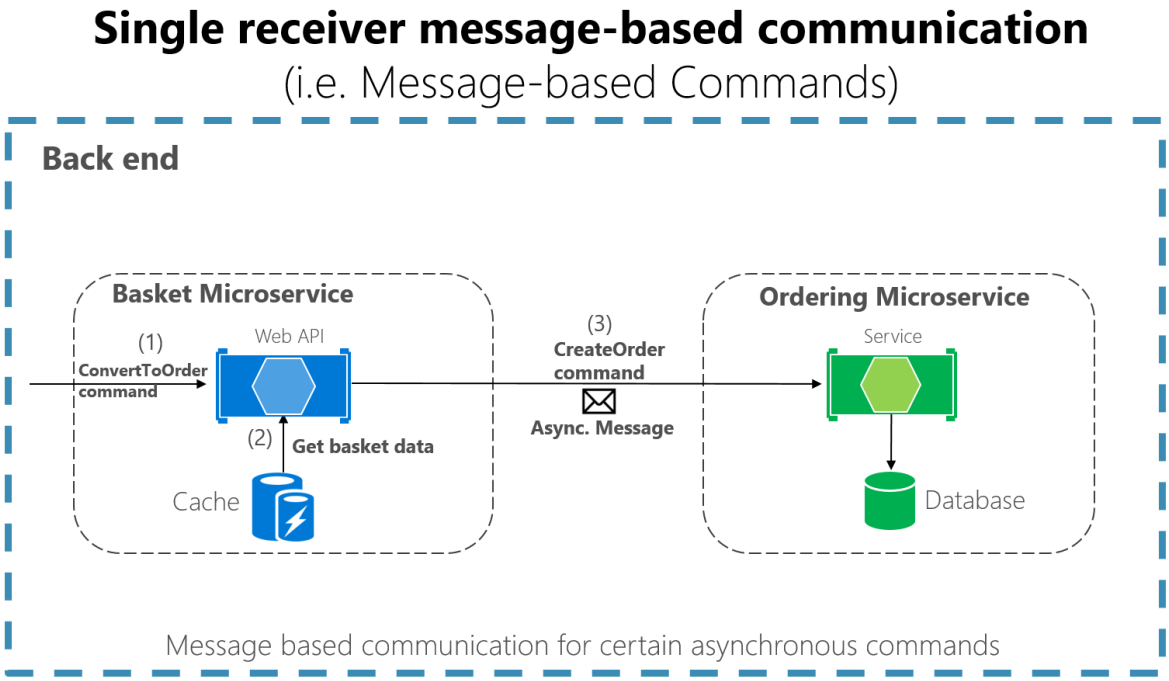


Figure 4-18. A single microservice receiving an asynchronous message

When the commands come from client applications, they can be implemented as HTTP synchronous commands. Use message-based commands when you need higher scalability or when you're already in a message-based business process.

Multiple-receivers message-based communication

As a more flexible approach, you might also want to use a publish/subscribe mechanism so that your communication from the sender will be available to additional subscriber microservices or to external applications. Thus, it helps you to follow the open/closed principle in the sending service. That way, additional subscribers can be added in the future without the need to modify the sender service.

When you use a publish/subscribe communication, you might be using an event bus interface to publish events to any subscriber.

Asynchronous event-driven communication

When using asynchronous event-driven communication, a microservice publishes an integration event when something happens within its domain and another microservice needs to be aware of it, like a price change in a product catalog microservice. Additional microservices subscribe to the events so they can receive them asynchronously. When that happens, the receivers might update their own domain entities, which can cause more integration events to be published. This publish/subscribe system is performed by using an implementation of an event bus. The event bus can be designed as an abstraction or interface, with the API that's needed to subscribe or unsubscribe to events and to publish events. The event bus can also have one or more implementations based on any inter-process and messaging broker, like a messaging queue or service bus that supports asynchronous communication and a publish/subscribe model.

If a system uses eventual consistency driven by integration events, it's recommended that this approach is made clear to the end user. The system shouldn't use an approach that mimics integration events, like SignalR or polling systems from the client. The end user and the business owner have to explicitly embrace eventual consistency in the system and realize that in many cases the business doesn't have any problem with this approach, as long as it's explicit. This approach is important because users might expect to see some results immediately and this aspect might not happen with eventual consistency.

As noted earlier in the Challenges and solutions for distributed data management section, you can use integration events to implement business tasks that span multiple microservices. Thus, you'll have eventual consistency between those services. An eventually consistent transaction is made up of a collection of distributed actions. At each action, the related microservice updates a domain entity and publishes another integration event that raises the next action within the same end-to-end business task.

An important point is that you might want to communicate to multiple microservices that are subscribed to the same event. To do so, you can use publish/subscribe messaging based on event-driven communication, as shown in Figure 4-19. This publish/subscribe mechanism isn't exclusive to the microservice architecture. It's similar to the way Bounded Contexts in DDD should communicate, or to the way you propagate updates from the write database to the read database in the Command

and Query Responsibility Segregation (CQRS) architecture pattern. The goal is to have eventual consistency between multiple data sources across your distributed system.

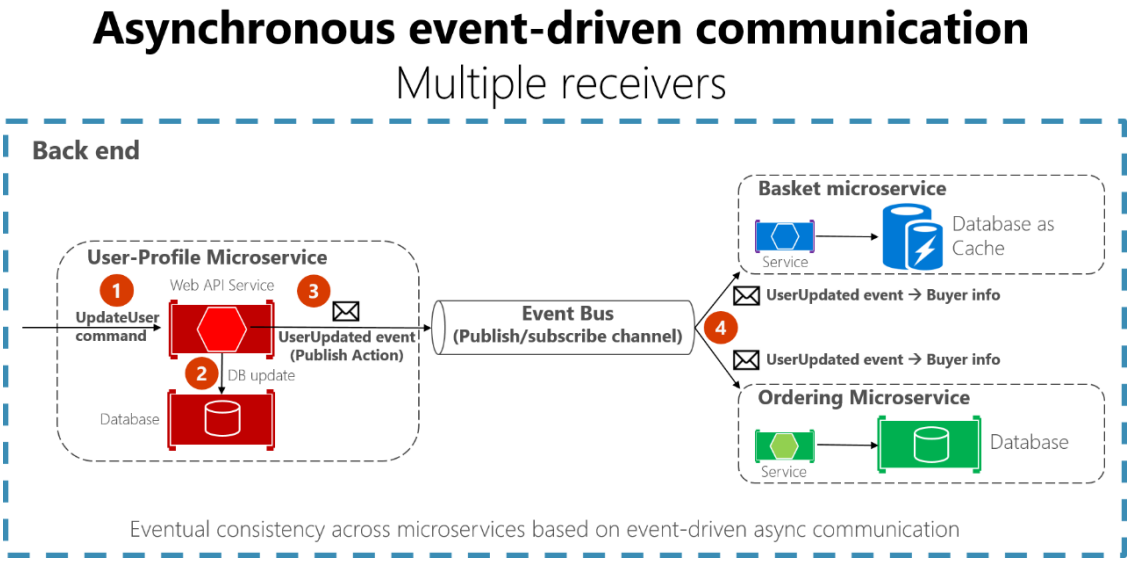


Figure 4-19. Asynchronous event-driven message communication

In asynchronous event-driven communication, one microservice publishes events to an event bus and many microservices can subscribe to it, to get notified and act on it. Your implementation will determine what protocol to use for event-driven, message-based communications. AMQP can help achieve reliable queued communication.

When you use an event bus, you might want to use an abstraction level (like an event bus interface) based on a related implementation in classes with code using the API from a message broker like RabbitMQ or a service bus like Azure Service Bus with Topics. Alternatively, you might want to use a higher-level service bus like NServiceBus, MassTransit, or Brighter to articulate your event bus and publish/subscribe system.

A note about messaging technologies for production systems

The messaging technologies available for implementing your abstract event bus are at different levels. For instance, products like RabbitMQ (a messaging broker transport) and Azure Service Bus sit at a lower level than other products like NServiceBus, MassTransit, or Brighter, which can work on top of RabbitMQ and Azure Service Bus. Your choice depends on how many rich features at the application level and out-of-the-box scalability you need for your application. For implementing just a proof-of-concept event bus for your development environment, as it was done in the eShopOnContainers sample, a simple implementation on top of RabbitMQ running on a Docker container might be enough.

However, for mission-critical and production systems that need hyper-scalability, you might want to evaluate Azure Service Bus. For high-level abstractions and features that make the development of distributed applications easier, we recommend that you evaluate other commercial and open-source service buses, such as NServiceBus, MassTransit, and Brighter. Of course, you can build your own

service-bus features on top of lower-level technologies like RabbitMQ and Docker. But that plumbing work might cost too much for a custom enterprise application.

Resiliently publishing to the event bus

A challenge when implementing an event-driven architecture across multiple microservices is how to atomically update state in the original microservice while resiliently publishing its related integration event into the event bus, somehow based on transactions. The following are a few ways to accomplish this functionality, although there could be additional approaches as well.

- Using a transactional (DTC-based) queue like MSMQ. (However, this is a legacy approach.)
- Using transaction log mining.
- Using full Event Sourcing pattern.
- Using the Outbox pattern: a transactional database table as a message queue that will be the base for an event-creator component that would create the event and publish it.

For a more complete description of the challenges in this space, including how messages with potentially incorrect data can end up being published, see Data platform for mission-critical workloads on Azure: Every message must be processed.

Additional topics to consider when using asynchronous communication are message idempotence and message deduplication. These topics are covered in the section Implementing event-based communication between microservices (integration events) later in this guide.

Additional resources

- **Event Driven Messaging**
  https://patterns.arcitura.com/soa-patterns/design_patterns/event_driven_messaging
- **Publish/Subscribe Channel**
  https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
- **Udi Dahan. Clarified CQRS**
  https://udidahan.com/2009/12/09/clarified-cqrs/
- **Command and Query Responsibility Segregation (CQRS)**
  https://learn.microsoft.com/azure/architecture/patterns/cqrs
- **Communicating Between Bounded Contexts**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591572(v=pandp.10)
- **Eventual consistency**
  https://en.wikipedia.org/wiki/Eventual_consistency
- **Jimmy Bogard. Refactoring Towards Resilience: Evaluating Coupling**
  https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/

Creating, evolving, and versioning microservice APIs and contracts

A microservice API is a contract between the service and its clients. You'll be able to evolve a microservice independently only if you do not break its API contract, which is why the contract is so important. If you change the contract, it will impact your client applications or your API Gateway.

The nature of the API definition depends on which protocol you're using. For instance, if you're using messaging, like AMQP, the API consists of the message types. If you're using HTTP and RESTful services, the API consists of the URLs and the request and response JSON formats.

However, even if you're thoughtful about your initial contract, a service API will need to change over time. When that happens—and especially if your API is a public API consumed by multiple client applications — you typically can't force all clients to upgrade to your new API contract. You usually need to incrementally deploy new versions of a service in a way that both old and new versions of a service contract are running simultaneously. Therefore, it's important to have a strategy for your service versioning.

When the API changes are small, like if you add attributes or parameters to your API, clients that use an older API should switch and work with the new version of the service. You might be able to provide default values for any missing attributes that are required, and the clients might be able to ignore any extra response attributes.

However, sometimes you need to make major and incompatible changes to a service API. Because you might not be able to force client applications or services to upgrade immediately to the new version, a service must support older versions of the API for some period. If you're using an HTTP-based mechanism such as REST, one approach is to embed the API version number in the URL or into an HTTP header. Then you can decide between implementing both versions of the service simultaneously within the same service instance, or deploying different instances that each handle a version of the API. A good approach for this functionality is the Mediator pattern (for example, MediatR library) to decouple the different implementation versions into independent handlers.

Finally, if you're using a REST architecture, Hypermedia is the best solution for versioning your services and allowing evolvable APIs.

Additional resources

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

Microservices addressability and the service registry

Each microservice has a unique name (URL) that's used to resolve its location. Your microservice needs to be addressable wherever it's running. If you have to think about which computer is running a particular microservice, things can go bad quickly. In the same way that DNS resolves a URL to a particular computer, your microservice needs to have a unique name so that its current location is discoverable. Microservices need addressable names that make them independent from the infrastructure that they're running on. This approach implies that there's an interaction between how your service is deployed and how it's discovered, because there needs to be a service registry. In the same vein, when a computer fails, the registry service must be able to indicate where the service is now running.

The service registry pattern is a key part of service discovery. The registry is a database containing the network locations of service instances. A service registry needs to be highly available and up-to-date. Clients could cache network locations obtained from the service registry. However, that information eventually goes out of date and clients can no longer discover service instances. So, a service registry consists of a cluster of servers that use a replication protocol to maintain consistency.

In some microservice deployment environments (called clusters, to be covered in a later section), service discovery is built in. For example, an Azure Kubernetes Service (AKS) environment can handle service instance registration and deregistration. It also runs a proxy on each cluster host that plays the role of server-side discovery router.

Additional resources

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

Creating composite UI based on microservices

Microservices architecture often starts with the server-side handling data and logic, but, in many cases, the UI is still handled as a monolith. However, a more advanced approach, called micro frontends, is to design your application UI based on microservices as well. That means having a composite UI produced by the microservices, instead of having microservices on the server and just a monolithic client app consuming the microservices. With this approach, the microservices you build can be complete with both logic and visual representation.

Figure 4-20 shows the simpler approach of just consuming microservices from a monolithic client application. Of course, you could have an ASP.NET MVC service in between producing the HTML and JavaScript. The figure is a simplification that highlights that you have a single (monolithic) client UI

consuming the microservices, which just focus on logic and data and not on the UI shape (HTML and JavaScript).

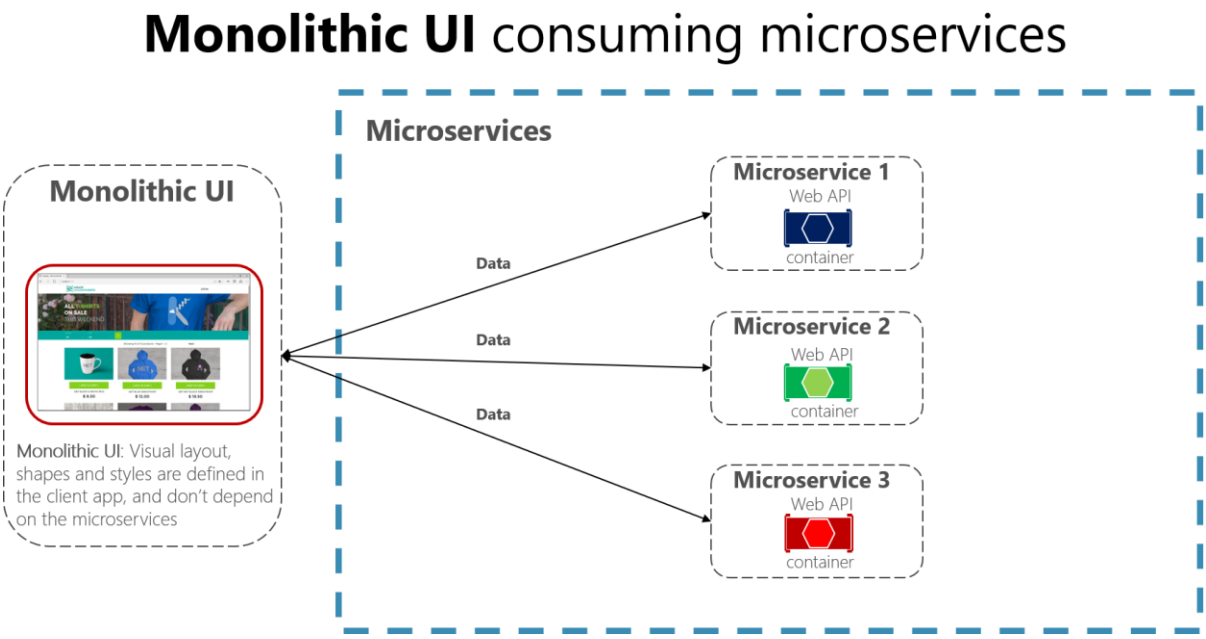


Figure 4-20. A monolithic UI application consuming back-end microservices

In contrast, a composite UI is precisely generated and composed by the microservices themselves. Some of the microservices drive the visual shape of specific areas of the UI. The key difference is that you have client UI components (TypeScript classes, for example) based on templates, and the data-shaping-UI ViewModel for those templates comes from each microservice.

At client application start-up time, each of the client UI components (TypeScript classes, for example) registers itself with an infrastructure microservice capable of providing ViewModels for a given scenario. If the microservice changes the shape, the UI changes also.

Figure 4-21 shows a version of this composite UI approach. This approach is simplified because you might have other microservices that are aggregating granular parts that are based on different techniques. It depends on whether you're building a traditional web approach (ASP.NET MVC) or an SPA (Single Page Application).

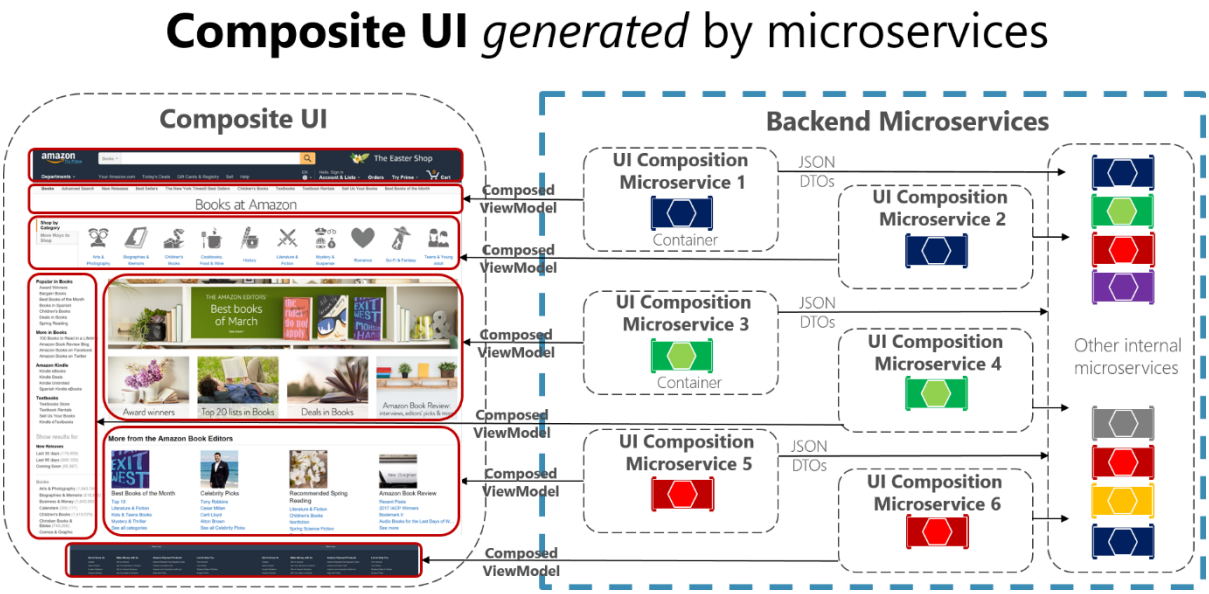


Figure 4-21. Example of a composite UI application shaped by back-end microservices

Each of those UI composition microservices would be similar to a small API Gateway. But in this case, each one is responsible for a small UI area.

A composite UI approach that's driven by microservices can be more challenging or less so, depending on what UI technologies you're using. For instance, you won't use the same techniques for building a traditional web application that you use for building an SPA or for native mobile app (as when developing Xamarin apps, which can be more challenging for this approach).

The eShopOnContainers sample application uses the monolithic UI approach for multiple reasons. First, it's an introduction to microservices and containers. A composite UI is more advanced but also requires further complexity when designing and developing the UI. Second, eShopOnContainers also provides a native mobile app based on Xamarin, which would make it more complex on the client C# side.

However, we encourage you to use the following references to learn more about composite UI based on microservices.

Additional resources

- **Micro Frontends (Martin Fowler's blog)**
  https://martinfowler.com/articles/micro-frontends.html
- **Micro Frontends (Michael Geers site)**
  https://micro-frontends.org/
- **Composite UI using ASP.NET (Particular's Workshop)**
  https://github.com/Particular/Workshop/tree/master/demos/asp-net-core
- **Ruben Oostinga. The Monolithic Frontend in the Microservices Architecture**
  https://xebia.com/blog/the-monolithic-frontend-in-the-microservices-architecture/

- **Mauro Servienti. The secret of better UI composition**
  https://particular.net/blog/secret-of-better-ui-composition
- **Viktor Farcic. Including Front-End Web Components Into Microservices**
  https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/
- **Managing Frontend in the Microservices Architecture**
  https://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

Resiliency and high availability in microservices

Dealing with unexpected failures is one of the hardest problems to solve, especially in a distributed system. Much of the code that developers write involves handling exceptions, and this is also where the most time is spent in testing. The problem is more involved than writing code to handle failures. What happens when the machine where the microservice is running fails? Not only do you need to detect this microservice failure (a hard problem on its own), but you also need something to restart your microservice.

A microservice needs to be resilient to failures and to be able to restart often on another machine for availability. This resiliency also comes down to the state that was saved on behalf of the microservice, where the microservice can recover this state from, and whether the microservice can restart successfully. In other words, there needs to be resiliency in the compute capability (the process can restart at any time) as well as resilience in the state or data (no data loss, and the data remains consistent).

The problems of resiliency are compounded during other scenarios, such as when failures occur during an application upgrade. The microservice, working with the deployment system, needs to determine whether it can continue to move forward to the newer version or instead roll back to a previous version to maintain a consistent state. Questions such as whether enough machines are available to keep moving forward and how to recover previous versions of the microservice need to be considered. This approach requires the microservice to emit health information so that the overall application and orchestrator can make these decisions.

In addition, resiliency is related to how cloud-based systems must behave. As mentioned, a cloud-based system must embrace failures and must try to automatically recover from them. For instance, in case of network or container failures, client apps or client services must have a strategy to retry sending messages or to retry requests, since in many cases failures in the cloud are partial. The Implementing Resilient Applications section in this guide addresses how to handle partial failure. It describes techniques like retries with exponential backoff or the Circuit Breaker pattern in .NET by using libraries like Polly, which offers a large variety of policies to handle this subject.

Health management and diagnostics in microservices

It may seem obvious, and it's often overlooked, but a microservice must report its health and diagnostics. Otherwise, there's little insight from an operations perspective. Correlating diagnostic events across a set of independent services and dealing with machine clock skews to make sense of

the event order is challenging. In the same way that you interact with a microservice over agreed-upon protocols and data formats, there's a need for standardization in how to log health and diagnostic events that ultimately end up in an event store for querying and viewing. In a microservices approach, it's key that different teams agree on a single logging format. There needs to be a consistent approach to viewing diagnostic events in the application.

Health checks

Health is different from diagnostics. Health is about the microservice reporting its current state to take appropriate actions. A good example is working with upgrade and deployment mechanisms to maintain availability. Although a service might currently be unhealthy due to a process crash or machine reboot, the service might still be operational. The last thing you need is to make this worse by performing an upgrade. The best approach is to do an investigation first or allow time for the microservice to recover. Health events from a microservice help us make informed decisions and, in effect, help create self-healing services.

In the Implementing health checks in ASP.NET Core services section of this guide, we explain how to use a new ASP.NET HealthChecks library in your microservices so they can report their state to a monitoring service to take appropriate actions.

You also have the option of using an excellent open-source library called AspNetCore.Diagnostics.HealthChecks, available on GitHub and as a NuGet package. This library also does health checks, with a twist, it handles two types of checks:

- **Liveness**: Checks if the microservice is alive, that is, if it's able to accept requests and respond.
- **Readiness**: Checks if the microservice's dependencies (Database, queue services, etc.) are themselves ready, so the microservice can do what it's supposed to do.

Using diagnostics and logs event streams

Logs provide information about how an application or service is running, including exceptions, warnings, and simple informational messages. Usually, each log is in a text format with one line per event, although exceptions also often show the stack trace across multiple lines.

In monolithic server-based applications, you can write logs to a file on disk (a logfile) and then analyze it with any tool. Since application execution is limited to a fixed server or VM, it generally isn't too complex to analyze the flow of events. However, in a distributed application where multiple services are executed across many nodes in an orchestrator cluster, being able to correlate distributed events is a challenge.

A microservice-based application should not try to store the output stream of events or logfiles by itself, and not even try to manage the routing of the events to a central place. It should be transparent, meaning that each process should just write its event stream to a standard output that underneath will be collected by the execution environment infrastructure where it's running. An example of these event stream routers is Microsoft.Diagnostic.EventFlow, which collects event streams from multiple sources and publishes it to output systems. These can include simple standard output for a development environment or cloud systems like Azure Monitor and Azure Diagnostics. There are also good third-party log analysis platforms and tools that can search, alert, report, and monitor logs, even in real time, like Splunk.

Orchestrators managing health and diagnostics information

When you create a microservice-based application, you need to deal with complexity. Of course, a single microservice is simple to deal with, but dozens or hundreds of types and thousands of instances of microservices is a complex problem. It isn't just about building your microservice architecture—you also need high availability, addressability, resiliency, health, and diagnostics if you intend to have a stable and cohesive system.

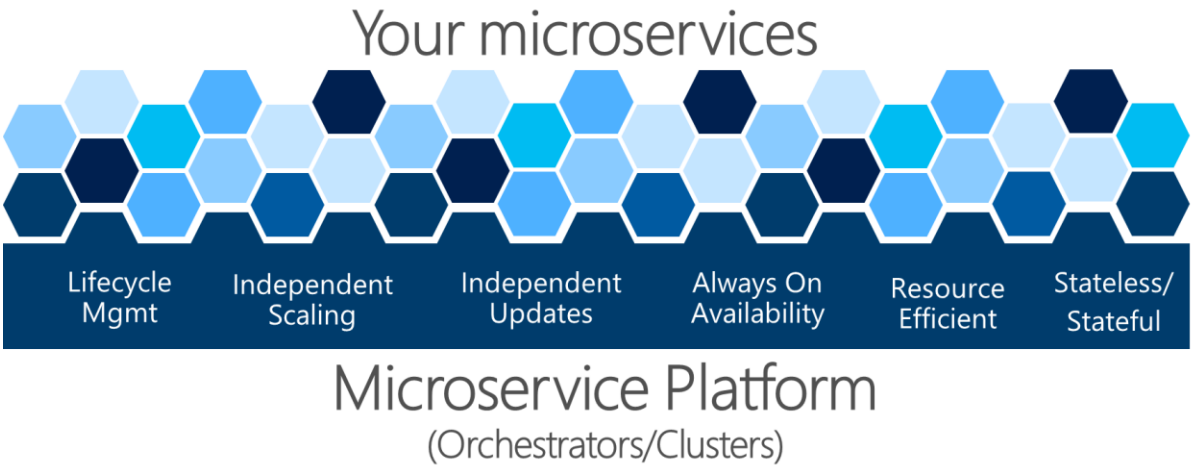


Figure 4-22. A Microservice Platform is fundamental for an application's health management

The complex problems shown in Figure 4-22 are hard to solve by yourself. Development teams should focus on solving business problems and building custom applications with microservice-based approaches. They should not focus on solving complex infrastructure problems; if they did, the cost of any microservice-based application would be huge. Therefore, there are microservice-oriented platforms, referred to as orchestrators or microservice clusters, that try to solve the hard problems of building and running a service and using infrastructure resources efficiently. This approach reduces the complexities of building applications that use a microservices approach.

Different orchestrators might sound similar, but the diagnostics and health checks offered by each of them differ in features and state of maturity, sometimes depending on the OS platform, as explained in the next section.

Additional resources

- **The Twelve-Factor App. XI. Logs: Treat logs as event streams**
  https://12factor.net/logs
- **Microsoft Diagnostic EventFlow Library** GitHub repo.
  https://github.com/Azure/diagnostics-eventflow
- **What is Azure Diagnostics**
  https://learn.microsoft.com/azure/azure-diagnostics

- **Connect Windows computers to the Azure Monitor service**
  https://learn.microsoft.com/azure/azure-monitor/platform/agent-windows
- **Logging What You Mean: Using the Semantic Logging Application Block**
  https://learn.microsoft.com/previous-versions/msp-n-p/dn440729(v=pandp.60)
- **Splunk** Official site.
  https://www.splunk.com/
- **EventSource Class** API for events tracing for Windows (ETW)
  https://learn.microsoft.com/dotnet/api/system.diagnostics.tracing.eventsource

Orchestrate microservices and multi-container applications for high scalability and availability

Using orchestrators for production-ready applications is essential if your application is based on microservices or simply split across multiple containers. As introduced previously, in a microservice-based approach, each microservice owns its model and data so that it will be autonomous from a development and deployment point of view. But even if you have a more traditional application that's composed of multiple services (like SOA), you'll also have multiple containers or services comprising a single business application that need to be deployed as a distributed system. These kinds of systems are complex to scale out and manage; therefore, you absolutely need an orchestrator if you want to have a production-ready and scalable multi-container application.

Figure 4-23 illustrates deployment into a cluster of an application composed of multiple microservices (containers).

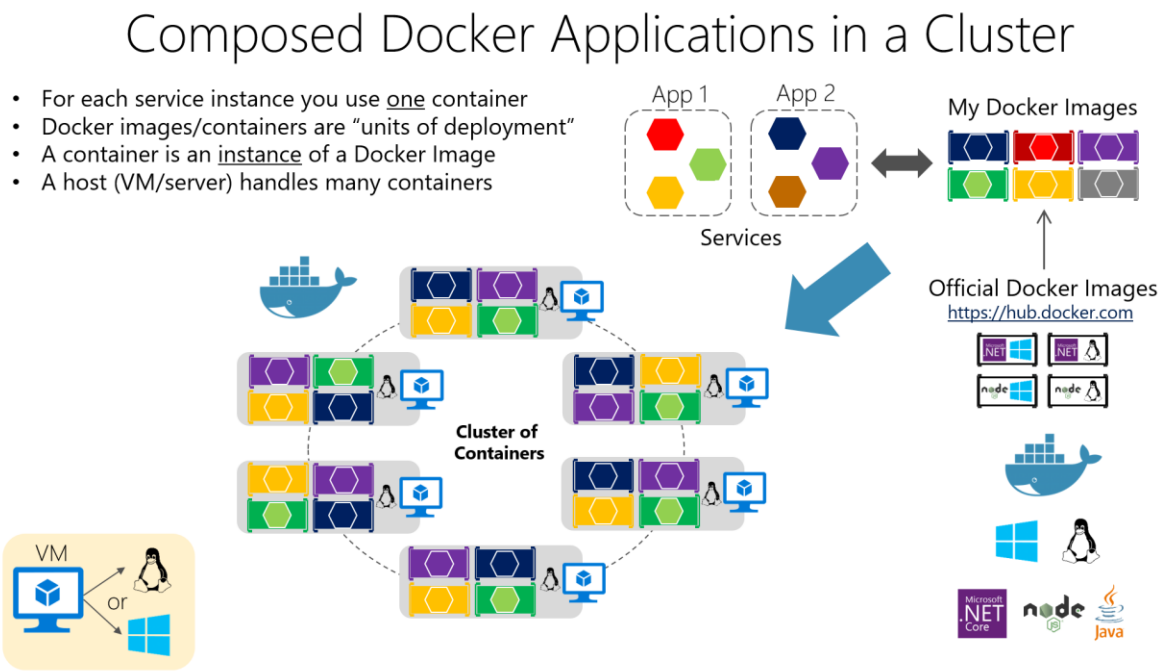


Figure 4-23. A cluster of containers

You use one container for each service instance. Docker containers are "units of deployment" and a container is an instance of a Docker. A host handles many containers. It looks like a logical approach. But how are you handling load-balancing, routing, and orchestrating these composed applications?

The plain Docker Engine in single Docker hosts meets the needs of managing single image instances on one host, but it falls short when it comes to managing multiple containers deployed on multiple hosts for more complex distributed applications. In most cases, you need a management platform that will automatically start containers, scale out containers with multiple instances per image, suspend them or shut them down when needed, and ideally also control how they access resources like the network and data storage.

To go beyond the management of individual containers or simple composed apps and move toward larger enterprise applications with microservices, you must turn to orchestration and clustering platforms.

From an architecture and development point of view, if you're building large enterprise composed of microservices-based applications, it's important to understand the following platforms and products that support advanced scenarios:

Clusters and orchestrators. When you need to scale out applications across many Docker hosts, as when a large microservice-based application, it's critical to be able to manage all those hosts as a single cluster by abstracting the complexity of the underlying platform. That's what the container clusters and orchestrators provide. Kubernetes is an example of an orchestrator, and is available in Azure through Azure Kubernetes Service.

Schedulers. *Scheduling* means to have the capability for an administrator to launch containers in a cluster so they also provide a UI. A cluster scheduler has several responsibilities: to use the cluster's

resources efficiently, to set the constraints provided by the user, to efficiently load-balance containers across nodes or hosts, and to be robust against errors while providing high availability.

The concepts of a cluster and a scheduler are closely related, so the products provided by different vendors often provide both sets of capabilities. The following list shows the most important platform and software choices you have for clusters and schedulers. These orchestrators are generally offered in public clouds like Azure.

Software platforms for container clustering, orchestration, and scheduling

Platform	Description
Kubernetes	*Kubernetes* is an open-source product that provides functionality that ranges from cluster infrastructure and container scheduling to orchestrating capabilities. It lets you automate deployment, scaling, and operations of application containers across clusters of hosts. *Kubernetes* provides a container-centric infrastructure that groups application containers into logical units for easy management and discovery. *Kubernetes* is mature in Linux, less mature in Windows.
Azure Kubernetes Service (AKS)	AKS is a managed Kubernetes container orchestration service in Azure that simplifies Kubernetes cluster's management, deployment, and operations.
Azure Container Apps	Azure Container Apps is a managed serverless container service for building and deploying modern apps at scale.

Using container-based orchestrators in Microsoft Azure

Several cloud vendors offer Docker containers support plus Docker clusters and orchestration support, including Microsoft Azure, Amazon EC2 Container Service, and Google Container Engine. Microsoft Azure provides Docker cluster and orchestrator support through Azure Kubernetes Service (AKS).

Using Azure Kubernetes Service

A Kubernetes cluster pools multiple Docker hosts and exposes them as a single virtual Docker host, so you can deploy multiple containers into the cluster and scale-out with any number of container instances. The cluster will handle all the complex management plumbing, like scalability, health, and so forth.

AKS provides a way to simplify the creation, configuration, and management of a cluster of virtual machines in Azure that are preconfigured to run containerized applications. Using an optimized configuration of popular open-source scheduling and orchestration tools, AKS enables you to use your existing skills or draw on a large and growing body of community expertise to deploy and manage container-based applications on Microsoft Azure.

Azure Kubernetes Service optimizes the configuration of popular Docker clustering open-source tools and technologies specifically for Azure. You get an open solution that offers portability for both your containers and your application configuration. You select the size, the number of hosts, and the orchestrator tools, and AKS handles everything else.

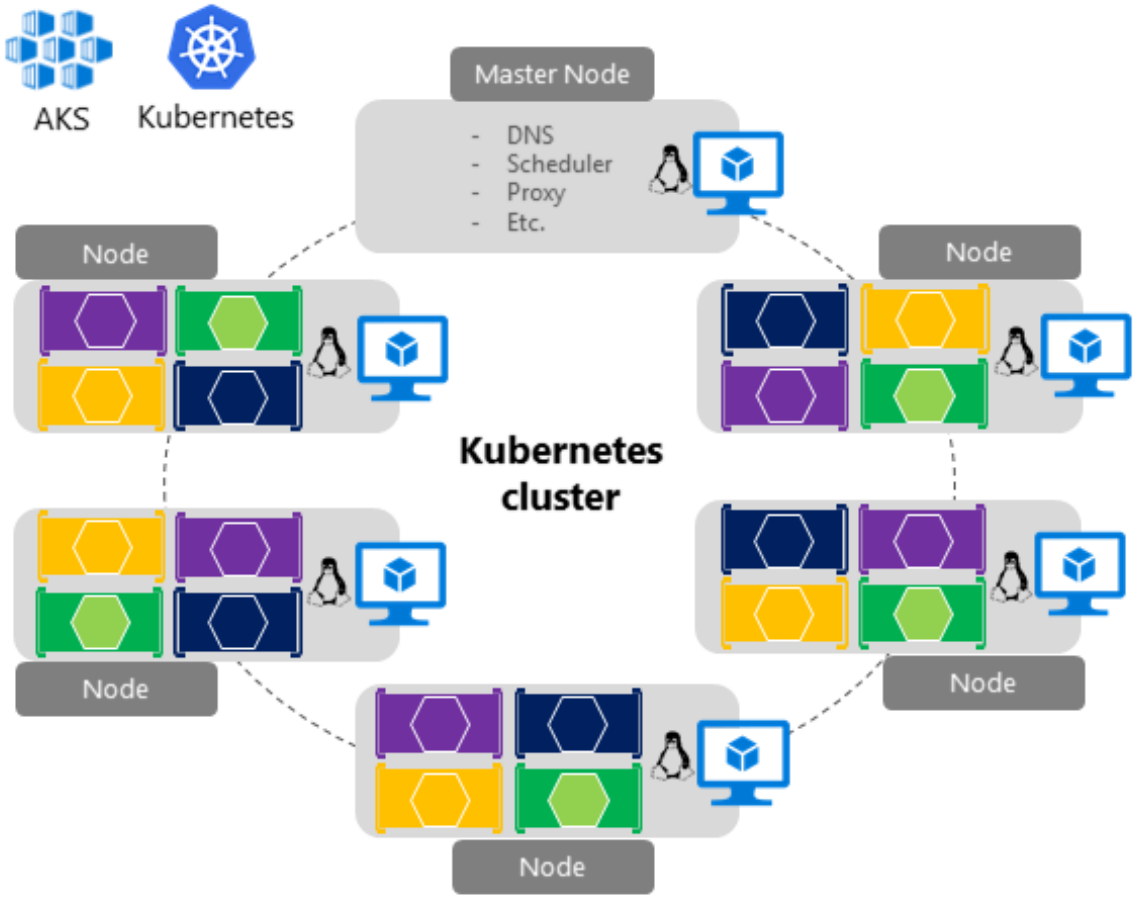


Figure 4-24. Kubernetes cluster's simplified structure and topology

In figure 4-24, you can see the structure of a Kubernetes cluster where a master node (VM) controls most of the coordination of the cluster and you can deploy containers to the rest of the nodes, which are managed as a single pool from an application point of view and allows you to scale to thousands or even tens of thousands of containers.

Development environment for Kubernetes

In the development environment, Docker announced in July 2018 that Kubernetes can also run in a single development machine (Windows 10 or macOS) by installing Docker Desktop. You can later deploy to the cloud (AKS) for further integration tests, as shown in figure 4-25.

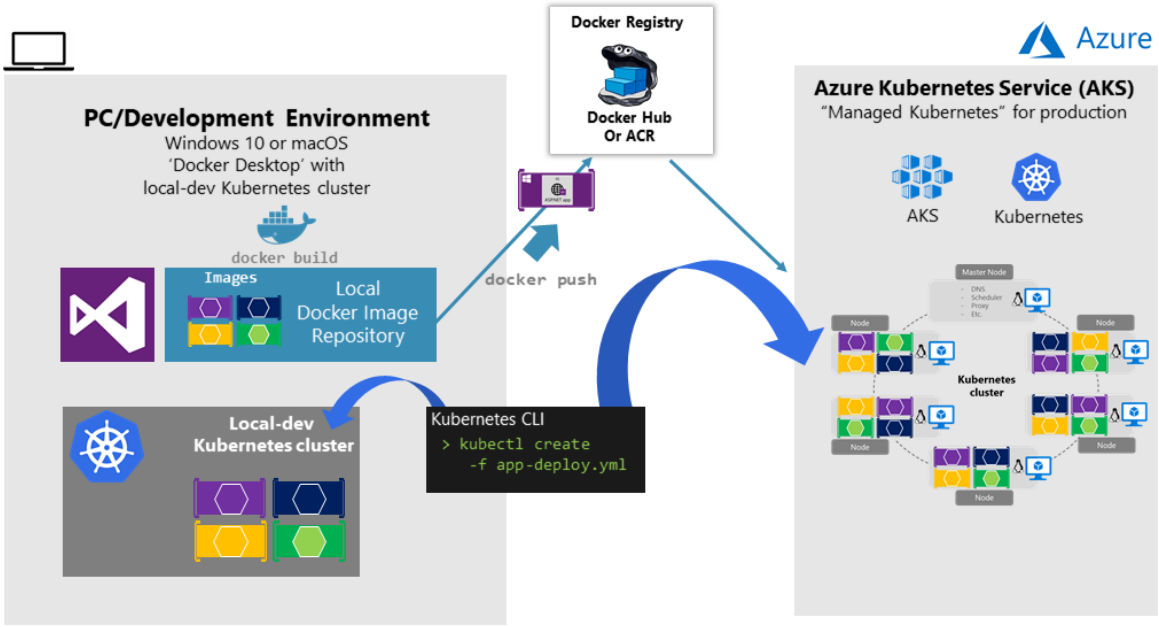


Figure 4-25. Running Kubernetes in dev machine and the cloud

Getting started with Azure Kubernetes Service (AKS)

To begin using AKS, you deploy an AKS cluster from the Azure portal or by using the CLI. For more information on deploying a Kubernetes cluster in Azure, see Deploy an Azure Kubernetes Service (AKS) cluster.

There are no fees for any of the software installed by default as part of AKS. All default options are implemented with open-source software. AKS is available for multiple virtual machines in Azure. You're charged only for the compute instances you choose, and the other underlying infrastructure resources consumed, such as storage and networking. There are no incremental charges for AKS itself.

The default production deployment option for Kubernetes is to use Helm charts, which are introduced in the next section.

Deploy with Helm charts into Kubernetes clusters

When deploying an application to a Kubernetes cluster, you can use the original kubectl.exe CLI tool using deployment files based on the native format (.yaml files), as already mentioned in the previous section. However, for more complex Kubernetes applications such as when deploying complex microservice-based applications, it's recommended to use Helm.

Helm Charts helps you define, version, install, share, upgrade, or rollback even the most complex Kubernetes application.

Going further, Helm usage is also recommended because other Kubernetes environments in Azure, such as Azure Dev Spaces are also based on Helm charts.

Helm is maintained by the Cloud Native Computing Foundation (CNCF) - in collaboration with Microsoft, Google, Bitnami, and the Helm contributor community.

For more implementation information on Helm charts and Kubernetes, see the Using Helm Charts to deploy eShopOnContainers to AKS post.

Additional resources

- **Getting started with Azure Kubernetes Service (AKS)**
  https://learn.microsoft.com/azure/aks/kubernetes-walkthrough-portal
- **Azure Dev Spaces**
  https://learn.microsoft.com/azure/dev-spaces/azure-dev-spaces
- **Kubernetes** The official site.
  https://kubernetes.io/

Development process for Docker-based applications

Develop containerized .NET applications the way you like, either Integrated Development Environment (IDE) focused with Visual Studio and Visual Studio tools for Docker or CLI/Editor focused with Docker CLI and Visual Studio Code.

Development environment for Docker apps

Development tool choices: IDE or editor

Whether you prefer a full and powerful IDE or a lightweight and agile editor, Microsoft has tools that you can use for developing Docker applications.

Visual Studio (for Windows). Docker-based .NET 7 application development with Visual Studio requires Visual Studio 2022 version 17.0 or later. Visual Studio 2022 comes with tools for Docker already built in. The tools for Docker let you develop, run, and validate your applications directly in the target Docker environment. You can press F5 to run and debug your application (single container or multiple containers) directly into a Docker host, or press CTRL + F5 to edit and refresh your application without having to rebuild the container. This IDE is the most powerful development choice for Docker-based apps.

Visual Studio for Mac. It's an IDE, evolution of Xamarin Studio, running in macOS. This tool should be the preferred choice for developers working in macOS machines who also want to use a powerful IDE.

Visual Studio Code and Docker CLI. If you prefer a lightweight and cross-platform editor that supports any development language, you can use Visual Studio Code and the Docker CLI. This IDE is a cross-platform development approach for macOS, Linux, and Windows. Additionally, Visual Studio Code supports extensions for Docker such as IntelliSense for Dockerfiles and shortcut tasks to run Docker commands from the editor.

By installing Docker Desktop, you can use a single Docker CLI to build apps for both Windows and Linux.

Additional resources

- **Visual Studio**. Official site.
  https://visualstudio.microsoft.com/vs/
- **Visual Studio Code**. Official site.
  https://code.visualstudio.com/download
- **Docker Desktop for Windows**
  https://hub.docker.com/editions/community/docker-ce-desktop-windows
- **Docker Desktop for Mac**
  https://hub.docker.com/editions/community/docker-ce-desktop-mac

.NET languages and frameworks for Docker containers

As mentioned in earlier sections of this guide, you can use .NET Framework, .NET 7, or the open-source Mono project when developing Docker containerized .NET applications. You can develop in C#, F#, or Visual Basic when targeting Linux or Windows Containers, depending on which .NET framework is in use. For more details about.NET languages, see the blog post The .NET Language Strategy.

Development workflow for Docker apps

The application development life cycle starts at your computer, as a developer, where you code the application using your preferred language and test it locally. With this workflow, no matter which language, framework, and platform you choose, you're always developing and testing Docker containers, but doing so locally.

Each container (an instance of a Docker image) includes the following components:

- An operating system selection, for example, a Linux distribution, Windows Nano Server, or Windows Server Core.
- Files added during development, for example, source code and application binaries.
- Configuration information, such as environment settings and dependencies.

Workflow for developing Docker container-based applications

This section describes the *inner-loop* development workflow for Docker container-based applications. The inner-loop workflow means it's not considering the broader DevOps workflow, which can include up to production deployment, and just focuses on the development work done on the developer's computer. The initial steps to set up the environment aren't included, since those steps are done only once.

An application is composed of your own services plus additional libraries (dependencies). The following are the basic steps you usually take when building a Docker application, as illustrated in Figure 5-1.

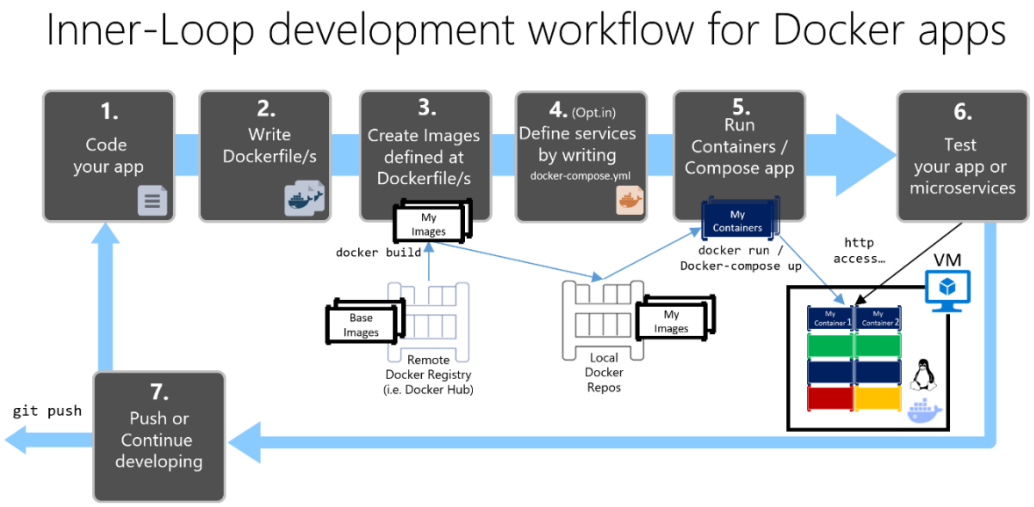


Figure 5-1. Step-by-step workflow for developing Docker containerized apps

In this section, this whole process is detailed and every major step is explained by focusing on a Visual Studio environment.

When you're using an editor/CLI development approach (for example, Visual Studio Code plus Docker CLI on macOS or Windows), you need to know every step, generally in more detail than if you're using Visual Studio. For more information about working in a CLI environment, see the e-book Containerized Docker Application lifecycle with Microsoft Platforms and Tools.

When you're using Visual Studio 2022, many of those steps are handled for you, which dramatically improves your productivity. This is especially true when you're using Visual Studio 2022 and targeting multi-container applications. For instance, with just one mouse click, Visual Studio adds the Dockerfile and docker-compose.yml file to your projects with the configuration for your application. When you run the application in Visual Studio, it builds the Docker image and runs the multi-container application directly in Docker; it even allows you to debug several containers at once. These features will boost your development speed.

However, just because Visual Studio makes those steps automatic doesn't mean that you don't need to know what's going on underneath with Docker. Therefore, the following guidance details every step.

Step 1. Start coding and create your initial application or service baseline

Developing a Docker application is similar to the way you develop an application without Docker. The difference is that while developing for Docker, you're deploying and testing your application or services running within Docker containers in your local environment (either a Linux VM setup by Docker or directly Windows if using Windows Containers).

Set up your local environment with Visual Studio

To begin, make sure you have Docker Desktop for Windows for Windows installed, as explained in the following instructions:

Get started with Docker Desktop for Windows

In addition, you need Visual Studio 2022 version 17.0, with the **.ASP.NET and web development** workload installed, as shown in Figure 5-2.

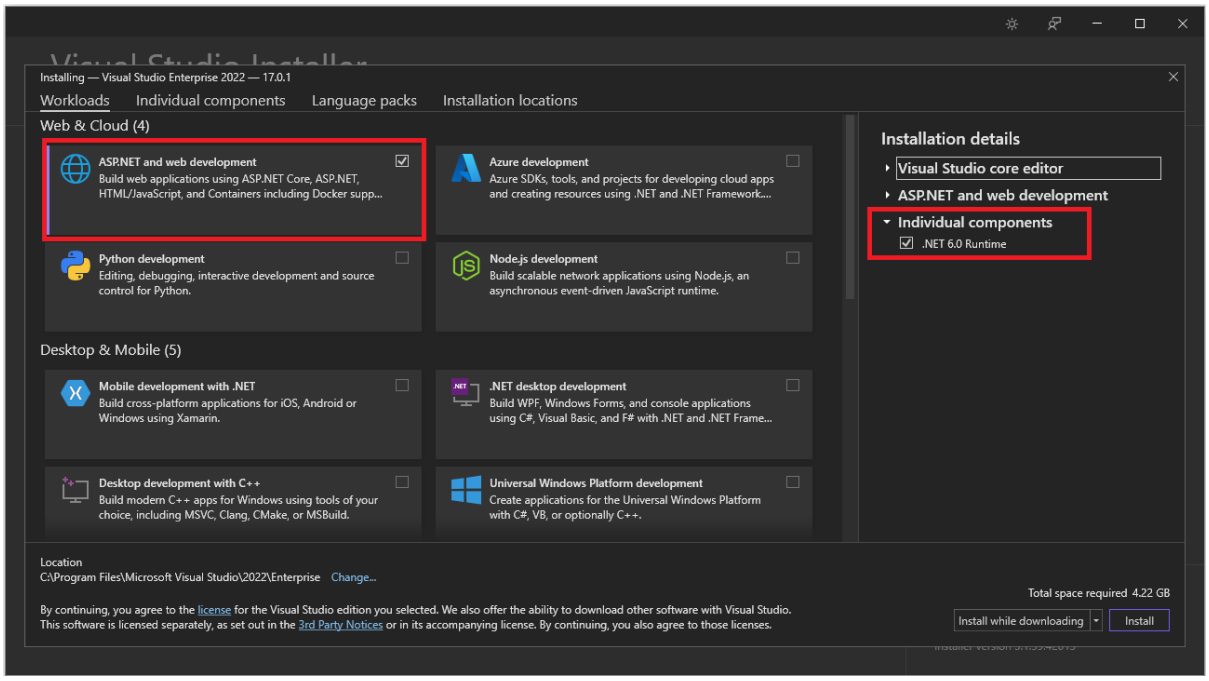


Figure 5-2. Selecting the ASP.NET and web development workload during Visual Studio 2022 setup

You can start coding your application in plain .NET (usually in .NET Core or later if you're planning to use containers) even before enabling Docker in your application and deploying and testing in Docker. However, it is recommended that you start working on Docker as soon as possible, because that will be the real environment and any issues can be discovered as soon as possible. This is encouraged

because Visual Studio makes it so easy to work with Docker that it almost feels transparent—the best example when debugging multi-container applications from Visual Studio.

Additional resources

- **Get started with Docker Desktop for Windows**
  https://docs.docker.com/docker-for-windows/
- **Visual Studio 2022**
  https://visualstudio.microsoft.com/downloads/

Step 2. Create a Dockerfile related to an existing .NET base image

You need a Dockerfile for each custom image you want to build; you also need a Dockerfile for each container to be deployed, whether you deploy automatically from Visual Studio or manually using the Docker CLI (docker run and docker-compose commands). If your application contains a single custom service, you need a single Dockerfile. If your application contains multiple services (as in a microservices architecture), you need one Dockerfile for each service.

The Dockerfile is placed in the root folder of your application or service. It contains the commands that tell Docker how to set up and run your application or service in a container. You can manually create a Dockerfile in code and add it to your project along with your .NET dependencies.

With Visual Studio and its tools for Docker, this task requires only a few mouse clicks. When you create a new project in Visual Studio 2022, there's an option named **Enable Docker Support**, as shown in Figure 5-3.

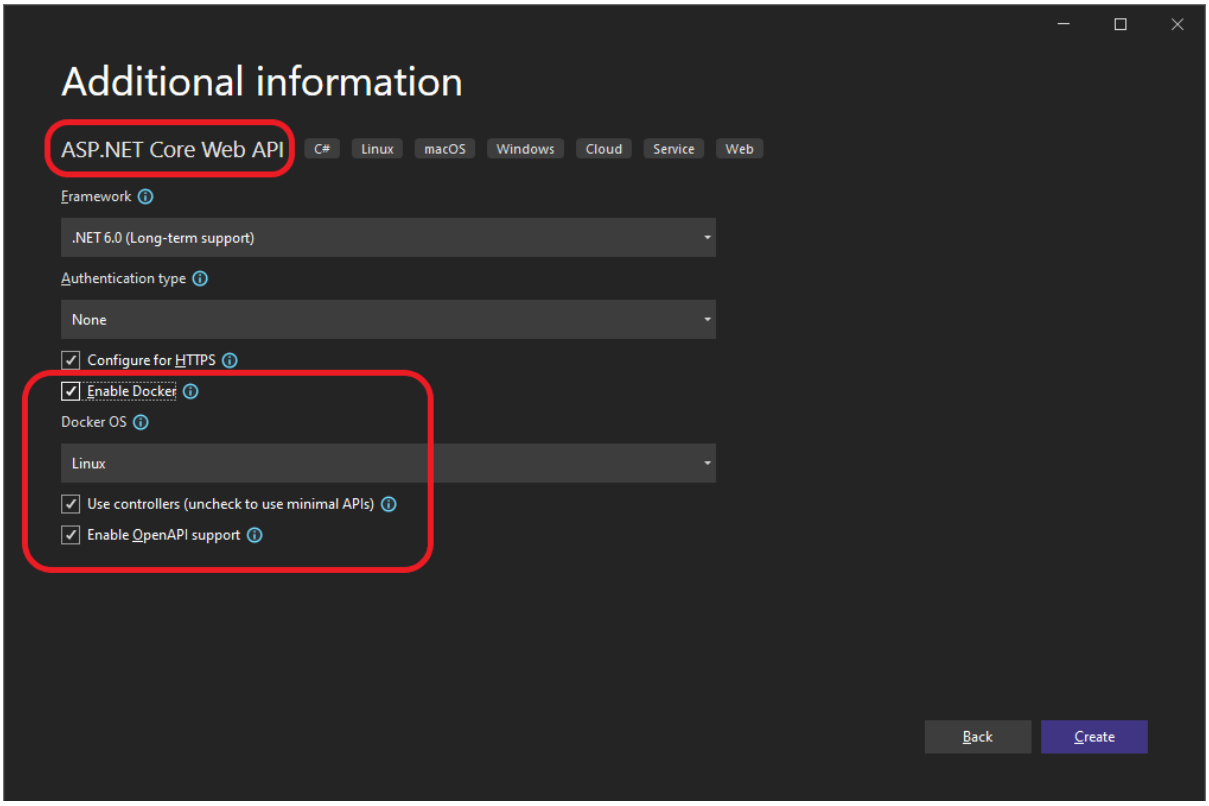


Figure 5-3. Enabling Docker Support when creating a new ASP.NET Core project in Visual Studio 2022

You can also enable Docker support on an existing ASP.NET Core web app project by right-clicking the project in **Solution Explorer** and selecting **Add** > **Docker Support...**, as shown in Figure 5-4.

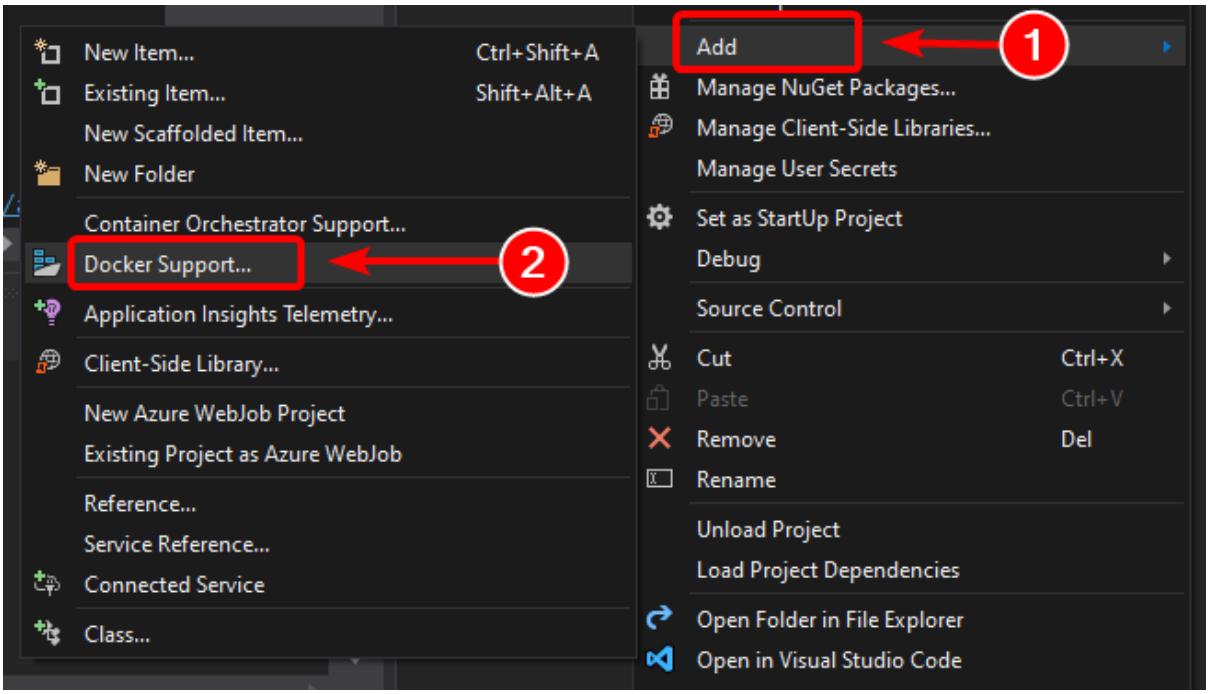


Figure 5-4. Enabling Docker support in an existing Visual Studio 2022 project

This action adds a *Dockerfile* to the project with the required configuration, and is only available on ASP.NET Core projects.

In a similar fashion, Visual Studio can also add a docker-compose.yml file for the whole solution with the option **Add > Container Orchestrator Support...**. In step 4, we'll explore this option in greater detail.

Using an existing official .NET Docker image

You usually build a custom image for your container on top of a base image you get from an official repository like the Docker Hub registry. That is precisely what happens under the covers when you enable Docker support in Visual Studio. Your Dockerfile will use an existing dotnet/core/aspnet image.

Earlier we explained which Docker images and repos you can use, depending on the framework and OS you have chosen. For instance, if you want to use ASP.NET Core (Linux or Windows), the image to use is mcr.microsoft.com/dotnet/aspnet:7.0. Therefore, you just need to specify what base Docker image you will use for your container. You do that by adding FROM mcr.microsoft.com/dotnet/aspnet:7.0 to your Dockerfile. This will be automatically performed by Visual Studio, but if you were to update the version, you update this value.

Using an official .NET image repository from Docker Hub with a version number ensures that the same language features are available on all machines (including development, testing, and production).

The following example shows a sample Dockerfile for an ASP.NET Core container.

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", " MySingleContainerWebApp.dll "]
```

In this case, the image is based on version 7.0 of the official ASP.NET Core Docker image (multi-arch for Linux and Windows). This is the setting FROM mcr.microsoft.com/dotnet/aspnet:7.0. (For more information about this base image, see the ASP.NET Core Docker Image page.) In the Dockerfile, you also need to instruct Docker to listen on the TCP port you will use at runtime (in this case, port 80, as configured with the EXPOSE setting).

You can specify additional configuration settings in the Dockerfile, depending on the language and framework you're using. For instance, the ENTRYPOINT line with ["dotnet", "MySingleContainerWebApp.dll"] tells Docker to run a .NET application. If you're using the SDK and the .NET CLI (dotnet CLI) to build and run the .NET application, this setting would be different. The bottom line is that the ENTRYPOINT line and other settings will be different depending on the language and platform you choose for your application.

Additional resources

- **Building Docker Images for ASP.NET Core Applications**
  https://learn.microsoft.com/dotnet/core/docker/building-net-docker-images

- **Build your own image**. In the official Docker documentation.
  https://docs.docker.com/engine/tutorials/dockerimages/
- **Staying up-to-date with .NET Container Images**
  https://devblogs.microsoft.com/dotnet/staying-up-to-date-with-net-container-images/
- **Using .NET and Docker Together - DockerCon 2018 Update**
  https://devblogs.microsoft.com/dotnet/using-net-and-docker-together-dockercon-2018-update/

Using multi-arch image repositories

A single repo can contain platform variants, such as a Linux image and a Windows image. This feature allows vendors like Microsoft (base image creators) to create a single repo to cover multiple platforms (that is Linux and Windows). For example, the .NET repository available in the Docker Hub registry provides support for Linux and Windows Nano Server by using the same repo name.

If you specify a tag, targeting a platform that is explicit like in the following cases:

- mcr.microsoft.com/dotnet/aspnet:7.0-bullseye-slim
  Targets: .NET 7 runtime-only on Linux
- mcr.microsoft.com/dotnet/aspnet:7.0-nanoserver-ltsc2022
  Targets: .NET 7 runtime-only on Windows Nano Server

But, if you specify the same image name, even with the same tag, the multi-arch images (like the aspnet image) will use the Linux or Windows version depending on the Docker host OS you're deploying, as shown in the following example:

- mcr.microsoft.com/dotnet/aspnet:7.0
  Multi-arch: .NET 7 runtime-only on Linux or Windows Nano Server depending on the Docker host OS

This way, when you pull an image from a Windows host, it will pull the Windows variant, and pulling the same image name from a Linux host will pull the Linux variant.

Multi-stage builds in Dockerfile

The Dockerfile is similar to a batch script. Similar to what you would do if you had to set up the machine from the command line.

It starts with a base image that sets up the initial context, it's like the startup filesystem, that sits on top of the host OS. It's not an OS, but you can think of it like "the" OS inside the container.

The execution of every command line creates a new layer on the filesystem with the changes from the previous one, so that, when combined, produce the resulting filesystem.

Since every new layer "rests" on top of the previous one and the resulting image size increases with every command, images can get very large if they have to include, for example, the SDK needed to build and publish an application.

This is where multi-stage builds get into the plot (from Docker 17.05 and higher) to do their magic.

The core idea is that you can separate the Dockerfile execution process in stages, where a stage is an initial image followed by one or more commands, and the last stage determines the final image size.

In short, multi-stage builds allow splitting the creation in different "phases" and then assemble the final image taking only the relevant directories from the intermediate stages. The general strategy to use this feature is:

1. Use a base SDK image (doesn't matter how large), with everything needed to build and publish the application to a folder and then
2. Use a base, small, runtime-only image and copy the publishing folder from the previous stage to produce a small final image.

Probably the best way to understand multi-stage is going through a Dockerfile in detail, line by line, so let's begin with the initial Dockerfile created by Visual Studio when adding Docker support to a project and will get into some optimizations later.

The initial Dockerfile might look something like this:

```
1  FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
2  WORKDIR /app
3  EXPOSE 80
4
5  FROM mcr.microsoft.com/dotnet/sdk:7.0 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"]
```

And these are the details, line by line:

- **Line #1:** Begin a stage with a "small" runtime-only base image, call it **base** for reference.
- **Line #2:** Create the **/app** directory in the image.
- **Line #3:** Expose port **80**.

- **Line #5:** Begin a new stage with the "large" image for building/publishing. Call it **build** for reference.
- **Line #6:** Create directory **/src** in the image.
- **Line #7:** Up to line 16, copy referenced **.csproj** project files to be able to restore packages later.
- **Line #17:** Restore packages for the **Catalog.API** project and the referenced projects.
- **Line #18:** Copy **all directory tree for the solution** (except the files/directories included in the **.dockerignore** file) to the **/src** directory in the image.
- **Line #19:** Change the current folder to the **Catalog.API** project.
- **Line #20:** Build the project (and other project dependencies) and output to the **/app** directory in the image.
- **Line #22:** Begin a new stage continuing from the build. Call it **publish** for reference.
- **Line #23:** Publish the project (and dependencies) and output to the **/app** directory in the image.
- **Line #25:** Begin a new stage continuing from **base** and call it **final**.
- **Line #26:** Change the current directory to **/app**.
- **Line #27:** Copy the **/app** directory from stage **publish** to the current directory.
- **Line #28:** Define the command to run when the container is started.

Now let's explore some optimizations to improve the whole process performance that, in the case of eShopOnContainers, means about 22 minutes or more to build the complete solution in Linux containers.

You'll take advantage of Docker's layer cache feature, which is quite simple: if the base image and the commands are the same as some previously executed, it can just use the resulting layer without the need to execute the commands, thus saving some time.

So, let's focus on the **build** stage, lines 5-6 are mostly the same, but lines 7-17 are different for every service from eShopOnContainers, so they have to execute every single time, however if you changed lines 7-16 to:

```
COPY . .
```

Then it would be just the same for every service, it would copy the whole solution and would create a larger layer but:

1. The copy process would only be executed the first time (and when rebuilding if a file is changed) and would use the cache for all other services and
2. Since the larger image occurs in an intermediate stage, it doesn't affect the final image size.

The next significant optimization involves the restore command executed in line 17, which is also different for every service of eShopOnContainers. If you change that line to just:

```
RUN dotnet restore
```

It would restore the packages for the whole solution, but then again, it would do it just once, instead of the 15 times with the current strategy.

However, dotnet restore only runs if there's a single project or solution file in the folder, so achieving this is a bit more complicated and the way to solve it, without getting into too many details, is this:

1. Add the following lines to **.dockerignore**:
   - *.sln, to ignore all solution files in the main folder tree
   - !eShopOnContainers-ServicesAndWebApps.sln, to include only this solution file.
2. Include the /ignoreprojectextensions:.dcproj argument to dotnet restore, so it also ignores the docker-compose project and only restores the packages for the eShopOnContainers-ServicesAndWebApps solution.

For the final optimization, it just happens that line 20 is redundant, as line 23 also builds the application and comes, in essence, right after line 20, so there goes another time-consuming command.

The resulting file is then:

```
 1  FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
 2  WORKDIR /app
 3  EXPOSE 80
 4
 5  FROM mcr.microsoft.com/dotnet/sdk:7.0 AS publish
 6  WORKDIR /src
 7  COPY . .
 8  RUN dotnet restore /ignoreprojectextensions:.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"]
```

Creating your base image from scratch

You can create your own Docker base image from scratch. This scenario is not recommended for someone who is starting with Docker, but if you want to set the specific bits of your own base image, you can do so.

Additional resources

- **Multi-arch .NET Core images**. https://github.com/dotnet/announcements/issues/14

- **Create a base image**. Official Docker documentation. https://docs.docker.com/develop/develop-images/baseimages/

Step 3. Create your custom Docker images and embed your application or service in them

For each service in your application, you need to create a related image. If your application is made up of a single service or web application, you just need a single image.

Note that the Docker images are built automatically for you in Visual Studio. The following steps are only needed for the editor/CLI workflow and explained for clarity about what happens underneath.

You, as a developer, need to develop and test locally until you push a completed feature or change to your source control system (for example, to GitHub). This means that you need to create the Docker images and deploy containers to a local Docker host (Windows or Linux VM) and run, test, and debug against those local containers.

To create a custom image in your local environment by using Docker CLI and your Dockerfile, you can use the docker build command, as in Figure 5-5.

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

Figure 5-5. Creating a custom Docker image

Optionally, instead of directly running docker build from the project folder, you can first generate a deployable folder with the required .NET libraries and binaries by running dotnet publish, and then use the docker build command.

This will create a Docker image with the name cesardl/netcore-webapi-microservice-docker:first. In this case, :first is a tag that represents a specific version. You can repeat this step for each custom image you need to create for your composed Docker application.

When an application is made of multiple containers (that is, it is a multi-container application), you can also use the docker-compose up --build command to build all the related images with a single command by using the metadata exposed in the related docker-compose.yml files.

You can find the existing images in your local repository by using the docker images command, as shown in Figure 5-6.

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

Figure 5-6. Viewing existing images using the docker images command

Creating Docker images with Visual Studio

When you use Visual Studio to create a project with Docker support, you don't explicitly create an image. Instead, the image is created for you when you press F5 (or Ctrl+F5) to run the dockerized application or service. This step is automatic in Visual Studio and you won't see it happen, but it's important that you know what's going on underneath.

Step 4. Define your services in docker-compose.yml when building a multi-container Docker application

The docker-compose.yml file lets you define a set of related services to be deployed as a composed application with deployment commands. It also configures its dependency relations and runtime configuration.

To use a docker-compose.yml file, you need to create the file in your main or root solution folder, with content similar to that in the following example:

```
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"
```

This docker-compose.yml file is a simplified and merged version. It contains static configuration data for each container (like the name of the custom image), which is always required, and configuration information that might depend on the deployment environment, like the connection string. In later sections, you will learn how to split the docker-compose.yml configuration into multiple docker-compose files and override values depending on the environment and execution type (debug or release).

The docker-compose.yml file example defines four services: the webmvc service (a web application), two microservices (ordering-api and basket-api), and one data source container, sqldata, based on SQL Server for Linux running as a container. Each service will be deployed as a container, so a Docker image is required for each.

The docker-compose.yml file specifies not only what containers are being used, but how they are individually configured. For instance, the webmvc container definition in the .yml file:

- Uses a pre-built eshop/web:latest image. However, you could also configure the image to be built as part of the docker-compose execution with an additional configuration based on a build: section in the docker-compose file.
- Initializes two environment variables (CatalogUrl and OrderingUrl).
- Forwards the exposed port 80 on the container to the external port 80 on the host machine.
- Links the web app to the catalog and ordering service with the depends_on setting. This causes the service to wait until those services are started.

We will revisit the docker-compose.yml file in a later section when we cover how to implement microservices and multi-container apps.

Working with docker-compose.yml in Visual Studio 2022

Besides adding a Dockerfile to a project, as we mentioned before, Visual Studio 2017 (from version 15.8 on) can add orchestrator support for Docker Compose to a solution.

When you add container orchestrator support, as shown in Figure 5-7, for the first time, Visual Studio creates the Dockerfile for the project and creates a new (service section) project in your solution with

several global docker-compose*.yml files, and then adds the project to those files. You can then open the docker-compose.yml files and update them with additional features.

Repeat this operation for every project you want to include in the docker-compose.yml file.

At the time of this writing, Visual Studio supports **Docker Compose** orchestrators.

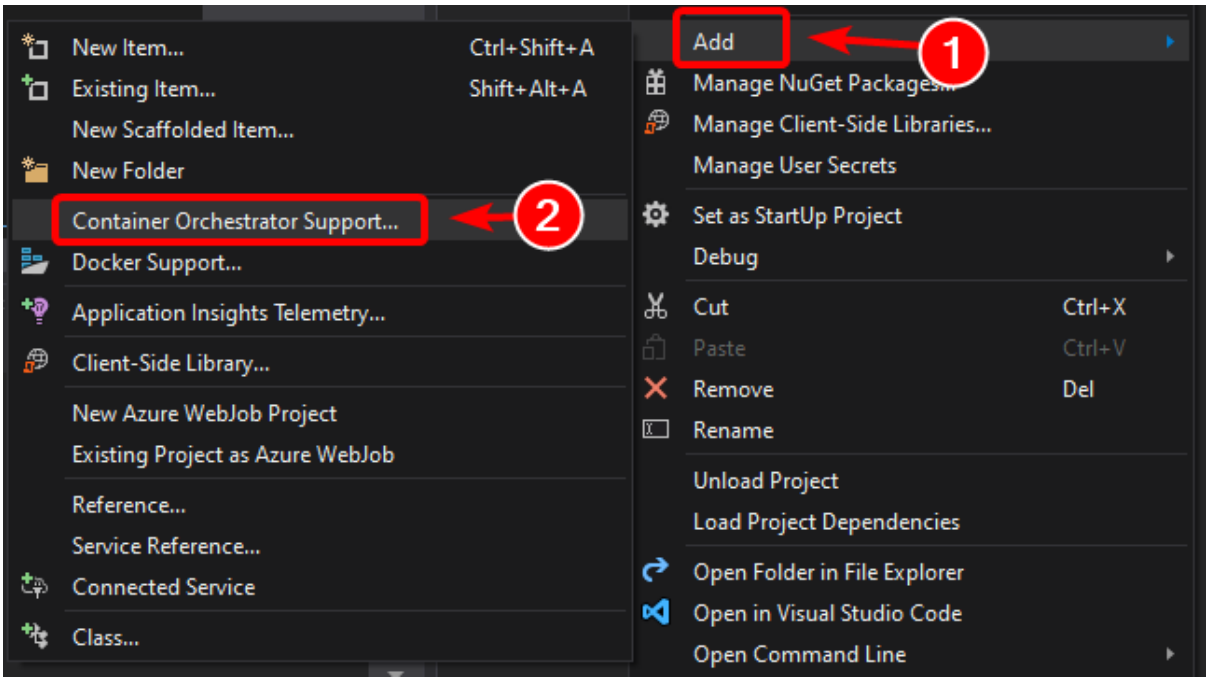


Figure 5-7. Adding Docker support in Visual Studio 2022 by right-clicking an ASP.NET Core project

After you add orchestrator support to your solution in Visual Studio, you will also see a new node (in the docker-compose.dcproj project file) in Solution Explorer that contains the added docker-compose.yml files, as shown in Figure 5-8.

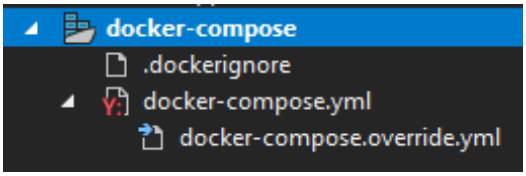


Figure 5-8. The docker-compose tree node added in Visual Studio 2022 Solution Explorer

You could deploy a multi-container application with a single docker-compose.yml file by using the docker-compose up command. However, Visual Studio adds a group of them so you can override values depending on the environment (development or production) and execution type (release or debug). This capability will be explained in later sections.

Step 5. Build and run your Docker application

If your application only has a single container, you can run it by deploying it to your Docker host (VM or physical server). However, if your application contains multiple services, you can deploy it as a composed application, either using a single CLI command (docker-compose up), or with Visual Studio, which will use that command under the covers. Let's look at the different options.

Option A: Running a single-container application

Using Docker CLI

You can run a Docker container using the docker run command, as shown in Figure 5-9:

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

The above command will create a new container instance from the specified image, every time it's run. You can use the --name parameter to give a name to the container and then use docker start {name} (or use the container ID or automatic name) to run an existing container instance.

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

Figure 5-9. Running a Docker container using the docker run command

In this case, the command binds the internal port 5000 of the container to port 80 of the host machine. This means that the host is listening on port 80 and forwarding to port 5000 on the container.

The hash shown is the container ID and it's also assigned a random readable name if the --name option is not used.

Using Visual Studio

If you haven't added container orchestrator support, you can also run a single container app in Visual Studio by pressing Ctrl+F5 and you can also use F5 to debug the application within the container. The container runs locally using docker run.

Option B: Running a multi-container application

In most enterprise scenarios, a Docker application will be composed of multiple services, which means you need to run a multi-container application, as shown in Figure 5-10.

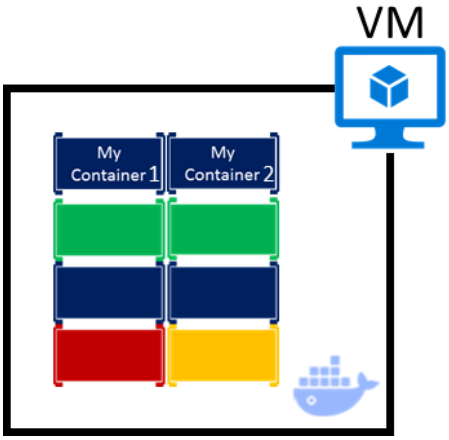


Figure 5-10. VM with Docker containers deployed

Using Docker CLI

To run a multi-container application with the Docker CLI, you use the docker-compose up command. This command uses the **docker-compose.yml** file that you have at the solution level to deploy a multi-container application. Figure 5-11 shows the results when running the command from your main solution directory, which contains the docker-compose.yml file.

```
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.
```

Figure 5-11. Example results when running the docker-compose up command

After the docker-compose up command runs, the application and its related containers are deployed into your Docker host, as depicted in Figure 5-10.

Using Visual Studio

Running a multi-container application using Visual Studio 2019 can't get any simpler. You just press Ctrl+F5 to run or F5 to debug, as usual, setting up the **docker-compose** project as the startup project. Visual Studio handles all needed setup, so you can create breakpoints as usual and debug what finally become independent processes running in "remote servers", with the debugger already attached, just like that.

As mentioned before, each time you add Docker solution support to a project within a solution, that project is configured in the global (solution-level) docker-compose.yml file, which lets you run or debug the whole solution at once. Visual Studio will start one container for each project that has Docker solution support enabled, and perform all the internal steps for you (dotnet publish, docker build, etc.).

If you want to take a peek at all the drudgery, take a look at the file:

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

The important point here is that, as shown in Figure 5-12, in Visual Studio 2019 there is an additional **Docker** command for the F5 key action. This option lets you run or debug a multi-container application by running all the containers that are defined in the docker-compose.yml files at the solution level. The ability to debug multiple-container solutions means that you can set several breakpoints, each breakpoint in a different project (container), and while debugging from Visual Studio you will stop at breakpoints defined in different projects and running on different containers.

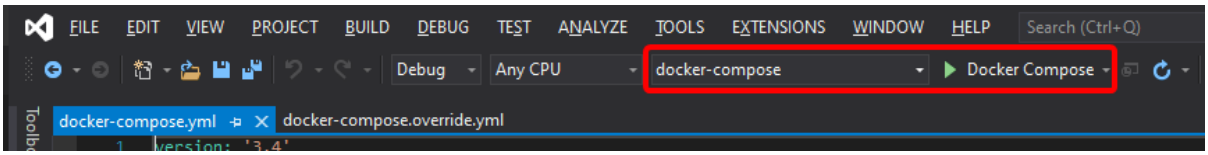


Figure 5-12. Running multi-container apps in Visual Studio 2022

Additional resources

- **Deploy an ASP.NET container to a remote Docker host**
  https://learn.microsoft.com/visualstudio/containers/hosting-web-apps-in-docker

A note about testing and deploying with orchestrators

The docker-compose up and docker run commands (or running and debugging the containers in Visual Studio) are adequate for testing containers in your development environment. But you should not use this approach for production deployments, where you should target orchestrators like Kubernetes or Service Fabric. If you're using Kubernetes, you have to use pods to organize containers and services to network them. You also use deployments to organize pod creation and modification.

Step 6. Test your Docker application using your local Docker host

This step will vary depending on what your application is doing. In a simple .NET Web application that is deployed as a single container or service, you can access the service by opening a browser on the Docker host and navigating to that site, as shown in Figure 5-13. (If the configuration in the Dockerfile maps the container to a port on the host that is anything other than 80, include the host port in the URL.)

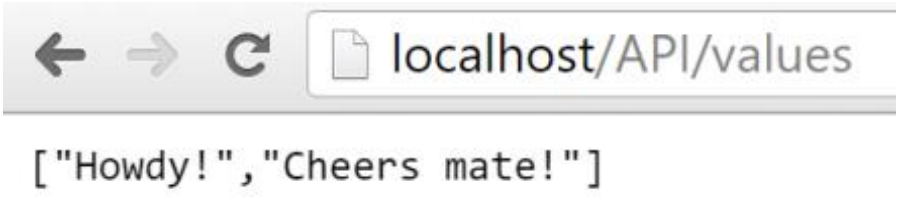


Figure 5-13. Example of testing your Docker application locally using localhost

If localhost is not pointing to the Docker host IP (by default, when using Docker CE, it should), to navigate to your service, use the IP address of your machine's network card.

This URL in the browser uses port 80 for the particular container example being discussed. However, internally the requests are being redirected to port 5000, because that was how it was deployed with the docker run command, as explained in a previous step.

You can also test the application using curl from the terminal, as shown in Figure 5-14. In a Docker installation on Windows, the default Docker Host IP is always 10.0.75.1 in addition to your machine's actual IP address.

```
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

                    ["Howdy!","Cheers mate!"]
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
```

Figure 5-14. Example of testing your Docker application locally using curl

Testing and debugging containers with Visual Studio 2022

When running and debugging the containers with Visual Studio 2022, you can debug the .NET application in much the same way as you would when running without containers.

Testing and debugging without Visual Studio

If you're developing using the editor/CLI approach, debugging containers is more difficult and you'll probably want to debug by generating traces.

Additional resources

- **Quickstart: Docker in Visual Studio.**
  https://learn.microsoft.com/visualstudio/containers/container-tools
- **Debugging apps in a local Docker container**
  https://learn.microsoft.com/visualstudio/containers/edit-and-refresh

Simplified workflow when developing containers with Visual Studio

Effectively, the workflow when using Visual Studio is a lot simpler than if you use the editor/CLI approach. Most of the steps required by Docker related to the Dockerfile and docker-compose.yml files are hidden or simplified by Visual Studio, as shown in Figure 5-15.

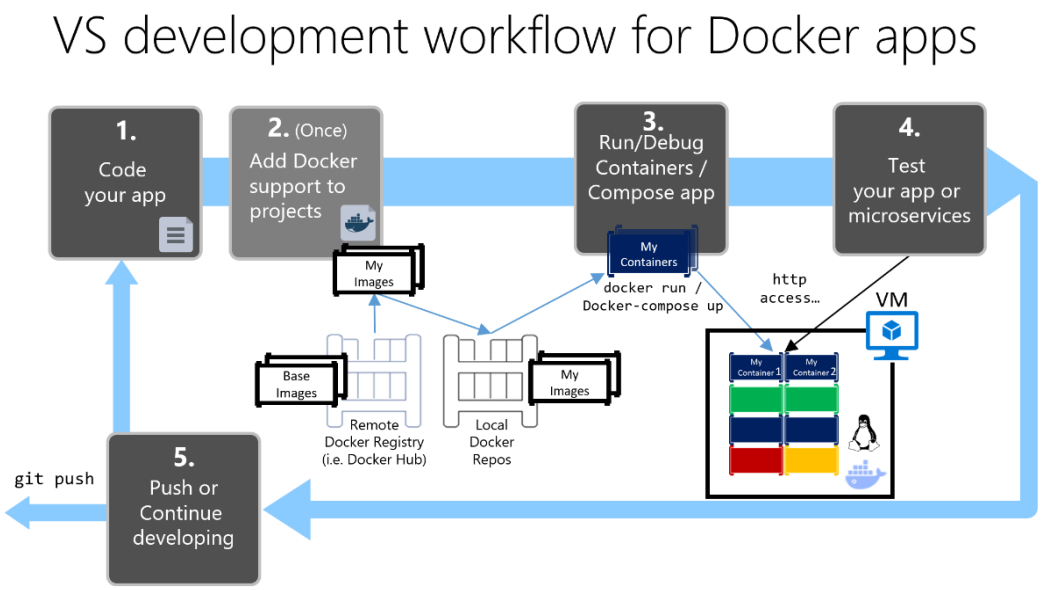


Figure 5-15. Simplified workflow when developing with Visual Studio

In addition, you need to perform step 2 (adding Docker support to your projects) just once. Therefore, the workflow is similar to your usual development tasks when using .NET for any other development. You need to know what is going on under the covers (the image build process, what base images you're using, deployment of containers, etc.) and sometimes you will also need to edit the Dockerfile or docker-compose.yml file to customize behaviors. But most of the work is greatly simplified by using Visual Studio, making you a lot more productive.

Using PowerShell commands in a Dockerfile to set up Windows Containers

Windows Containers allow you to convert your existing Windows applications into Docker images and deploy them with the same tools as the rest of the Docker ecosystem. To use Windows Containers, you run PowerShell commands in the Dockerfile, as shown in the following example:

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

In this case, we are using a Windows Server Core base image (the FROM setting) and installing IIS with a PowerShell command (the RUN setting). In a similar way, you could also use PowerShell commands to set up additional components like ASP.NET 4.x, .NET Framework 4.6, or any other Windows software. For example, the following command in a Dockerfile sets up ASP.NET 4.5:

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

Additional resources

- **aspnet-docker/Dockerfile.** Example PowerShell commands to run from dockerfiles to include Windows features.

https://github.com/Microsoft/aspnet-docker/blob/master/4.7.1-windowsservercore-ltsc2016/runtime/Dockerfile

CHAPTER 5

Designing and Developing Multi-Container and Microservice-Based .NET Applications

Developing containerized microservice applications means you are building multi-container applications. However, a multi-container application could also be simpler—for example, a three-tier application—and might not be built using a microservice architecture.

Earlier we raised the question "Is Docker necessary when building a microservice architecture?" The answer is a clear no. Docker is an enabler and can provide significant benefits, but containers and Docker are not a hard requirement for microservices. As an example, you could create a microservices-based application with or without Docker when using Azure Service Fabric, which supports microservices running as simple processes or as Docker containers.

However, if you know how to design and develop a microservices-based application that is also based on Docker containers, you will be able to design and develop any other, simpler application model. For example, you might design a three-tier application that also requires a multi-container approach. Because of that, and because microservice architectures are an important trend within the container world, this section focuses on a microservice architecture implementation using Docker containers.

Design a microservice-oriented application

This section focuses on developing a hypothetical server-side enterprise application.

Application specifications

The hypothetical application handles requests by executing business logic, accessing databases, and then returning HTML, JSON, or XML responses. We will say that the application must support various clients, including desktop browsers running Single Page Applications (SPAs), traditional web apps, mobile web apps, and native mobile apps. The application might also expose an API for third parties

to consume. It should also be able to integrate its microservices or external applications asynchronously, so that approach will help resiliency of the microservices in the case of partial failures.

The application will consist of these types of components:

- Presentation components. These components are responsible for handling the UI and consuming remote services.
- Domain or business logic. This component is the application's domain logic.
- Database access logic. This component consists of data access components responsible for accessing databases (SQL or NoSQL).
- Application integration logic. This component includes a messaging channel, based on message brokers.

The application will require high scalability, while allowing its vertical subsystems to scale out autonomously, because certain subsystems will require more scalability than others.

The application must be able to be deployed in multiple infrastructure environments (multiple public clouds and on-premises) and ideally should be cross-platform, able to move from Linux to Windows (or vice versa) easily.

Development team context

We also assume the following about the development process for the application:

- You have multiple dev teams focusing on different business areas of the application.
- New team members must become productive quickly, and the application must be easy to understand and modify.
- The application will have a long-term evolution and ever-changing business rules.
- You need good long-term maintainability, which means having agility when implementing new changes in the future while being able to update multiple subsystems with minimum impact on the other subsystems.
- You want to practice continuous integration and continuous deployment of the application.
- You want to take advantage of emerging technologies (frameworks, programming languages, etc.) while evolving the application. You do not want to make full migrations of the application when moving to new technologies, because that would result in high costs and impact the predictability and stability of the application.

Choosing an architecture

What should the application deployment architecture be? The specifications for the application, along with the development context, strongly suggest that you should architect the application by decomposing it into autonomous subsystems in the form of collaborating microservices and containers, where a microservice is a container.

In this approach, each service (container) implements a set of cohesive and narrowly related functions. For example, an application might consist of services such as the catalog service, ordering service, basket service, user profile service, etc.

Microservices communicate using protocols such as HTTP (REST), but also asynchronously (for example, using AMQP) whenever possible, especially when propagating updates with integration events.

Microservices are developed and deployed as containers independently of one another. This approach means that a development team can be developing and deploying a certain microservice without impacting other subsystems.

Each microservice has its own database, allowing it to be fully decoupled from other microservices. When necessary, consistency between databases from different microservices is achieved using application-level integration events (through a logical event bus), as handled in Command and Query Responsibility Segregation (CQRS). Because of that, the business constraints must embrace eventual consistency between the multiple microservices and related databases.

eShopOnContainers: A reference application for .NET and microservices deployed using containers

So that you can focus on the architecture and technologies instead of thinking about a hypothetical business domain that you might not know, we have selected a well-known business domain—namely, a simplified e-commerce (e-shop) application that presents a catalog of products, takes orders from customers, verifies inventory, and performs other business functions. This container-based application source code is available in the eShopOnContainers GitHub repo.

The application consists of multiple subsystems, including several store UI front ends (a Web application and a native mobile app), along with the back-end microservices and containers for all the required server-side operations with several API Gateways as consolidated entry points to the internal microservices. Figure 6-1 shows the architecture of the reference application.

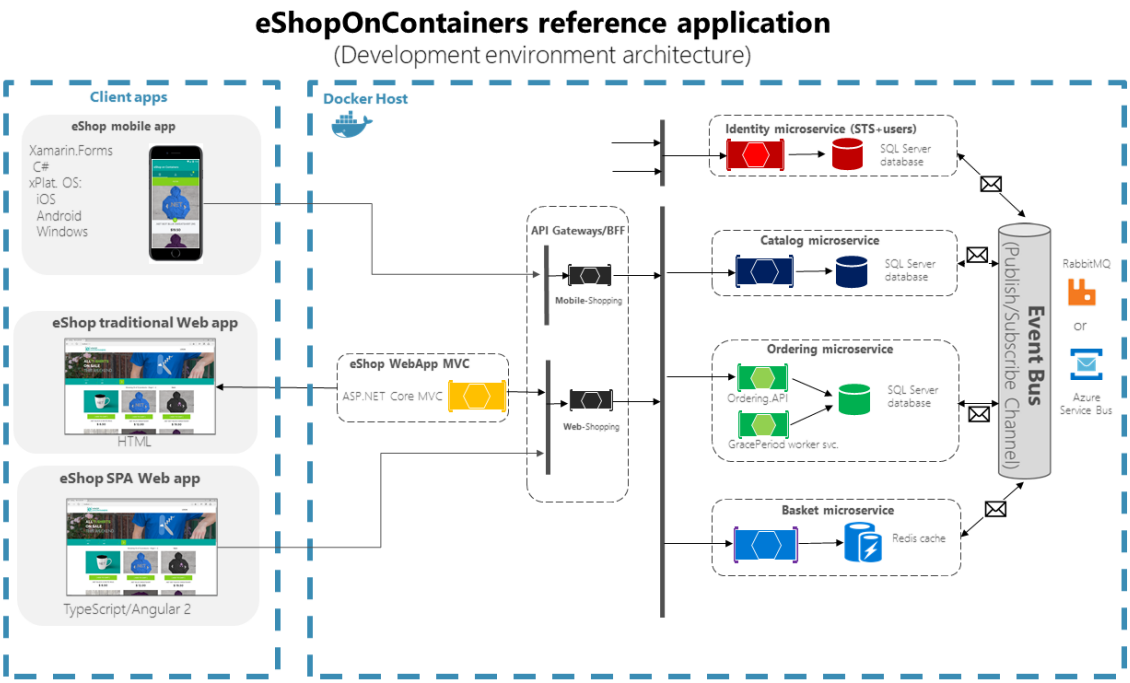


Figure 6-1. The eShopOnContainers reference application architecture for development environment

The above diagram shows that Mobile and SPA clients communicate to single API gateway endpoints, that then communicate to microservices. Traditional web clients communicate to MVC microservice, that communicates to microservices through the API gateway.

Hosting environment. In Figure 6-1, you see several containers deployed within a single Docker host. That would be the case when deploying to a single Docker host with the docker-compose up command. However, if you are using an orchestrator or container cluster, each container could be running in a different host (node), and any node could be running any number of containers, as we explained earlier in the architecture section.

Communication architecture. The eShopOnContainers application uses two communication types, depending on the kind of the functional action (queries versus updates and transactions):

- Http client-to-microservice communication through API Gateways. This approach is used for queries and when accepting update or transactional commands from the client apps. The approach using API Gateways is explained in detail in later sections.
- Asynchronous event-based communication. This communication occurs through an event bus to propagate updates across microservices or to integrate with external applications. The event bus can be implemented with any messaging-broker infrastructure technology like RabbitMQ, or using higher-level (abstraction-level) service buses like Azure Service Bus, NServiceBus, MassTransit, or Brighter.

The application is deployed as a set of microservices in the form of containers. Client apps can communicate with those microservices running as containers through the public URLs published by the API Gateways.

Data sovereignty per microservice

In the sample application, each microservice owns its own database or data source, although all SQL Server databases are deployed as a single container. This design decision was made only to make it easy for a developer to get the code from GitHub, clone it, and open it in Visual Studio or Visual Studio Code. Or alternatively, it makes it easy to compile the custom Docker images using the .NET CLI and the Docker CLI, and then deploy and run them in a Docker development environment. Either way, using containers for data sources lets developers build and deploy in a matter of minutes without having to provision an external database or any other data source with hard dependencies on infrastructure (cloud or on-premises).

In a real production environment, for high availability and for scalability, the databases should be based on database servers in the cloud or on-premises, but not in containers.

Therefore, the units of deployment for microservices (and even for databases in this application) are Docker containers, and the reference application is a multi-container application that embraces microservices principles.

Additional resources

- **eShopOnContainers GitHub repo. Source code for the reference application** https://aka.ms/eShopOnContainers/

Benefits of a microservice-based solution

A microservice-based solution like this has many benefits:

Each microservice is relatively small—easy to manage and evolve. Specifically:

- It is easy for a developer to understand and get started quickly with good productivity.
- Containers start fast, which makes developers more productive.
- An IDE like Visual Studio can load smaller projects fast, making developers productive.
- Each microservice can be designed, developed, and deployed independently of other microservices, which provide agility because it is easier to deploy new versions of microservices frequently.

It is possible to scale out individual areas of the application. For instance, the catalog service or the basket service might need to be scaled out, but not the ordering process. A microservices infrastructure will be much more efficient with regard to the resources used when scaling out than a monolithic architecture would be.

You can divide the development work between multiple teams. Each service can be owned by a single development team. Each team can manage, develop, deploy, and scale their service independently of the rest of the teams.

Issues are more isolated. If there is an issue in one service, only that service is initially impacted (except when the wrong design is used, with direct dependencies between microservices), and other services can continue to handle requests. In contrast, one malfunctioning component in a monolithic

deployment architecture can bring down the entire system, especially when it involves resources, such as a memory leak. Additionally, when an issue in a microservice is resolved, you can deploy just the affected microservice without impacting the rest of the application.

You can use the latest technologies. Because you can start developing services independently and run them side by side (thanks to containers and .NET), you can start using the latest technologies and frameworks expediently instead of being stuck on an older stack or framework for the whole application.

Downsides of a microservice-based solution

A microservice-based solution like this also has some drawbacks:

Distributed application. Distributing the application adds complexity for developers when they are designing and building the services. For example, developers must implement inter-service communication using protocols like HTTP or AMQP, which adds complexity for testing and exception handling. It also adds latency to the system.

Deployment complexity. An application that has dozens of microservices types and needs high scalability (it needs to be able to create many instances per service and balance those services across many hosts) means a high degree of deployment complexity for IT operations and management. If you are not using a microservice-oriented infrastructure (like an orchestrator and scheduler), that additional complexity can require far more development efforts than the business application itself.

Atomic transactions. Atomic transactions between multiple microservices usually are not possible. The business requirements have to embrace eventual consistency between multiple microservices.

Increased global resource needs (total memory, drives, and network resources for all the servers or hosts). In many cases, when you replace a monolithic application with a microservices approach, the amount of initial global resources needed by the new microservice-based application will be larger than the infrastructure needs of the original monolithic application. This approach is because the higher degree of granularity and distributed services requires more global resources. However, given the low cost of resources in general and the benefit of being able to scale out certain areas of the application compared to long-term costs when evolving monolithic applications, the increased use of resources is usually a good tradeoff for large, long-term applications.

Issues with direct client-to-microservice communication. When the application is large, with dozens of microservices, there are challenges and limitations if the application requires direct client-to-microservice communications. One problem is a potential mismatch between the needs of the client and the APIs exposed by each of the microservices. In certain cases, the client application might need to make many separate requests to compose the UI, which can be inefficient over the Internet and would be impractical over a mobile network. Therefore, requests from the client application to the back-end system should be minimized.

Another problem with direct client-to-microservice communications is that some microservices might be using protocols that are not Web-friendly. One service might use a binary protocol, while another service might use AMQP messaging. Those protocols are not firewall-friendly and are best used internally. Usually, an application should use protocols such as HTTP and WebSockets for communication outside of the firewall.

Yet another drawback with this direct client-to-service approach is that it makes it difficult to refactor the contracts for those microservices. Over time developers might want to change how the system is partitioned into services. For example, they might merge two services or split a service into two or more services. However, if clients communicate directly with the services, performing this kind of refactoring can break compatibility with client apps.

As mentioned in the architecture section, when designing and building a complex application based on microservices, you might consider the use of multiple fine-grained API Gateways instead of the simpler direct client-to-microservice communication approach.

Partitioning the microservices. Finally, no matter, which approach you take for your microservice architecture, another challenge is deciding how to partition an end-to-end application into multiple microservices. As noted in the architecture section of the guide, there are several techniques and approaches you can take. Basically, you need to identify areas of the application that are decoupled from the other areas and that have a low number of hard dependencies. In many cases, this approach is aligned to partitioning services by use case. For example, in our e-shop application, we have an ordering service that is responsible for all the business logic related to the order process. We also have the catalog service and the basket service that implement other capabilities. Ideally, each service should have only a small set of responsibilities. This approach is similar to the single responsibility principle (SRP) applied to classes, which states that a class should only have one reason to change. But in this case, it is about microservices, so the scope will be larger than a single class. Most of all, a microservice has to be autonomous, end to end, including responsibility for its own data sources.

External versus internal architecture and design patterns

The external architecture is the microservice architecture composed by multiple services, following the principles described in the architecture section of this guide. However, depending on the nature of each microservice, and independently of high-level microservice architecture you choose, it is common and sometimes advisable to have different internal architectures, each based on different patterns, for different microservices. The microservices can even use different technologies and programming languages. Figure 6-2 illustrates this diversity.

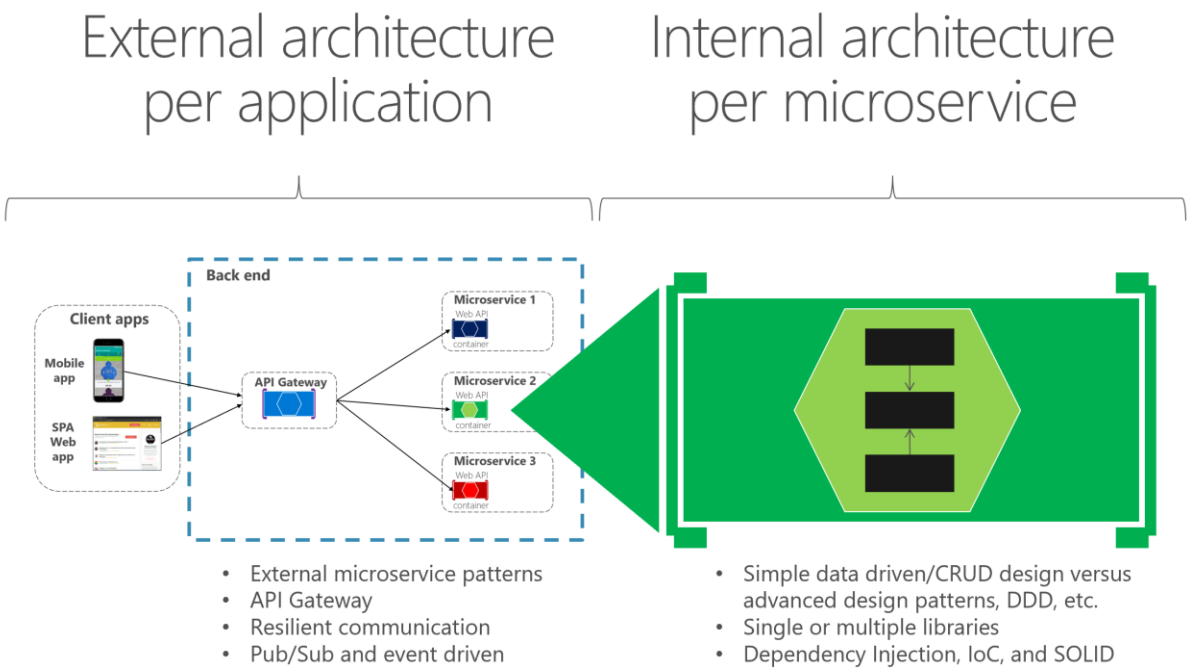


Figure 6-2. External versus internal architecture and design

For instance, in our *eShopOnContainers* sample, the catalog, basket, and user profile microservices are simple (basically, CRUD subsystems). Therefore, their internal architecture and design is straightforward. However, you might have other microservices, such as the ordering microservice, which is more complex and represents ever-changing business rules with a high degree of domain complexity. In cases like these, you might want to implement more advanced patterns within a particular microservice, like the ones defined with domain-driven design (DDD) approaches, as we are doing in the *eShopOnContainers* ordering microservice. (We will review these DDD patterns in the section later that explains the implementation of the *eShopOnContainers* ordering microservice.)

Another reason for a different technology per microservice might be the nature of each microservice. For example, it might be better to use a functional programming language like F#, or even a language like R if you are targeting AI and machine learning domains, instead of a more object-oriented programming language like C#.

The bottom line is that each microservice can have a different internal architecture based on different design patterns. Not all microservices should be implemented using advanced DDD patterns, because that would be over-engineering them. Similarly, complex microservices with ever-changing business logic should not be implemented as CRUD components, or you can end up with low-quality code.

The new world: multiple architectural patterns and polyglot microservices

There are many architectural patterns used by software architects and developers. The following are a few (mixing architecture styles and architecture patterns):

- Simple CRUD, single-tier, single-layer.

- Traditional N-Layered.
- Domain-Driven Design N-layered.
- Clean Architecture (as used with eShopOnWeb)
- Command and Query Responsibility Segregation (CQRS).
- Event-Driven Architecture (EDA).

You can also build microservices with many technologies and languages, such as ASP.NET Core Web APIs, NancyFx, ASP.NET Core SignalR (available with .NET Core 2 or later), F#, Node.js, Python, Java, C++, GoLang, and more.

The important point is that no particular architecture pattern or style, nor any particular technology, is right for all situations. Figure 6-3 shows some approaches and technologies (although not in any particular order) that could be used in different microservices.

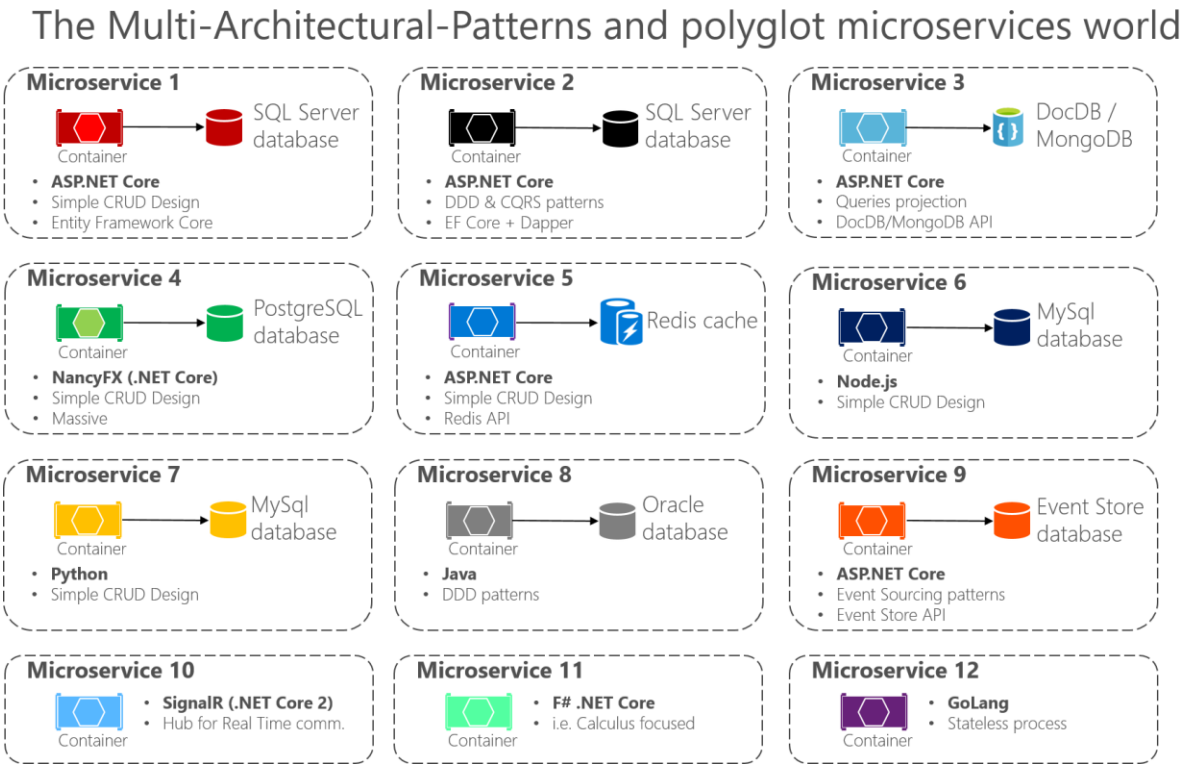


Figure 6-3. Multi-architectural patterns and the polyglot microservices world

Multi-architectural pattern and polyglot microservices means you can mix and match languages and technologies to the needs of each microservice and still have them talking to each other. As shown in Figure 6-3, in applications composed of many microservices (Bounded Contexts in domain-driven design terminology, or simply "subsystems" as autonomous microservices), you might implement each microservice in a different way. Each might have a different architecture pattern and use different languages and databases depending on the application's nature, business requirements, and priorities. In some cases, the microservices might be similar. But that is not usually the case, because each subsystem's context boundary and requirements are usually different.

For instance, for a simple CRUD maintenance application, it might not make sense to design and implement DDD patterns. But for your core domain or core business, you might need to apply more advanced patterns to tackle business complexity with ever-changing business rules.

Especially when you deal with large applications composed by multiple subsystems, you should not apply a single top-level architecture based on a single architecture pattern. For instance, CQRS should not be applied as a top-level architecture for a whole application, but might be useful for a specific set of services.

There is no silver bullet or a right architecture pattern for every given case. You cannot have "one architecture pattern to rule them all." Depending on the priorities of each microservice, you must choose a different approach for each, as explained in the following sections.

Creating a simple data-driven CRUD microservice

This section outlines how to create a simple microservice that performs create, read, update, and delete (CRUD) operations on a data source.

Designing a simple CRUD microservice

From a design point of view, this type of containerized microservice is very simple. Perhaps the problem to solve is simple, or perhaps the implementation is only a proof of concept.

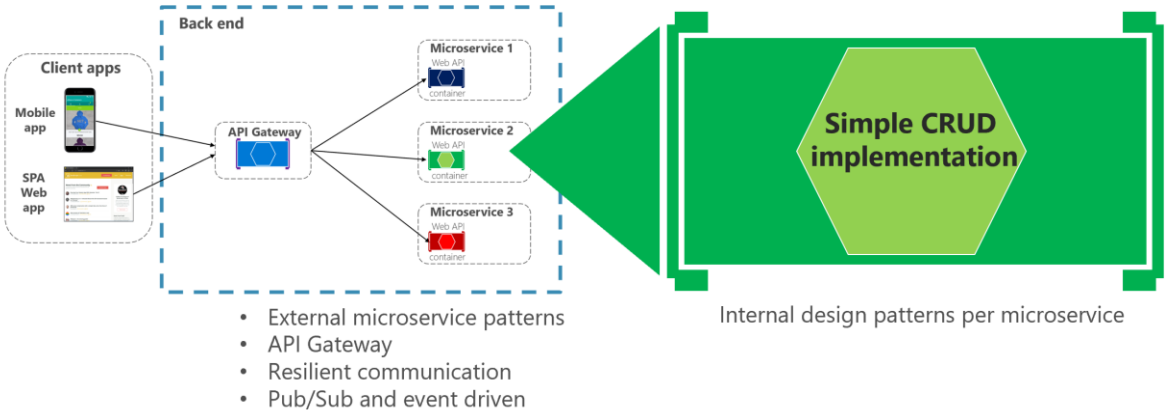


Figure 6-4. Internal design for simple CRUD microservices

An example of this kind of simple data-drive service is the catalog microservice from the eShopOnContainers sample application. This type of service implements all its functionality in a single ASP.NET Core Web API project that includes classes for its data model, its business logic, and its data access code. It also stores its related data in a database running in SQL Server (as another container for dev/test purposes), but could also be any regular SQL Server host, as shown in Figure 6-5.

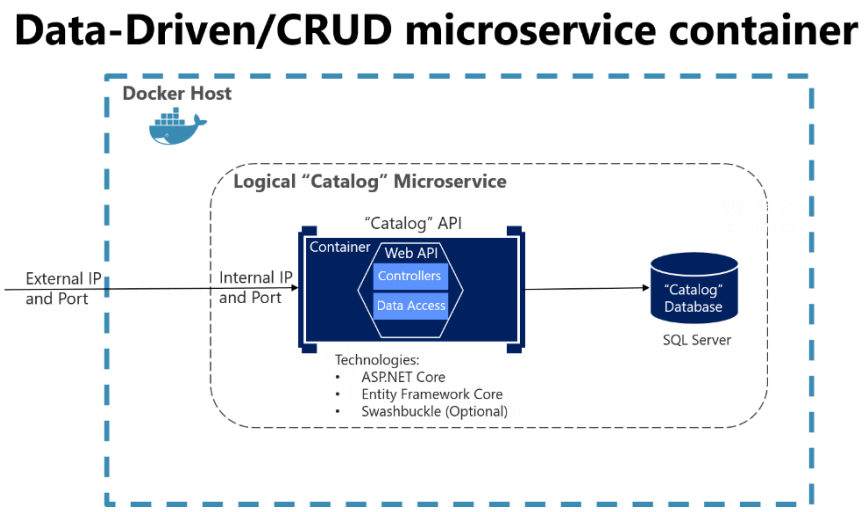


Figure 6-5. Simple data-driven/CRUD microservice design

The previous diagram shows the logical Catalog microservice, that includes its Catalog database, which can be or not in the same Docker host. Having the database in the same Docker host might be good for development, but not for production. When you are developing this kind of service, you only need ASP.NET Core and a data-access API or ORM like Entity Framework Core. You could also generate Swagger metadata automatically through Swashbuckle to provide a description of what your service offers, as explained in the next section.

Note that running a database server like SQL Server within a Docker container is great for development environments, because you can have all your dependencies up and running without needing to provision a database in the cloud or on-premises. This approach is convenient when running integration tests. However, for production environments, running a database server in a container is not recommended, because you usually do not get high availability with that approach. For a production environment in Azure, it is recommended that you use Azure SQL DB or any other database technology that can provide high availability and high scalability. For example, for a NoSQL approach, you might choose CosmosDB.

Finally, by editing the Dockerfile and docker-compose.yml metadata files, you can configure how the image of this container will be created—what base image it will use, plus design settings such as internal and external names and TCP ports.

Implementing a simple CRUD microservice with ASP.NET Core

To implement a simple CRUD microservice using .NET and Visual Studio, you start by creating a simple ASP.NET Core Web API project (running on .NET so it can run on a Linux Docker host), as shown in Figure 6-6.

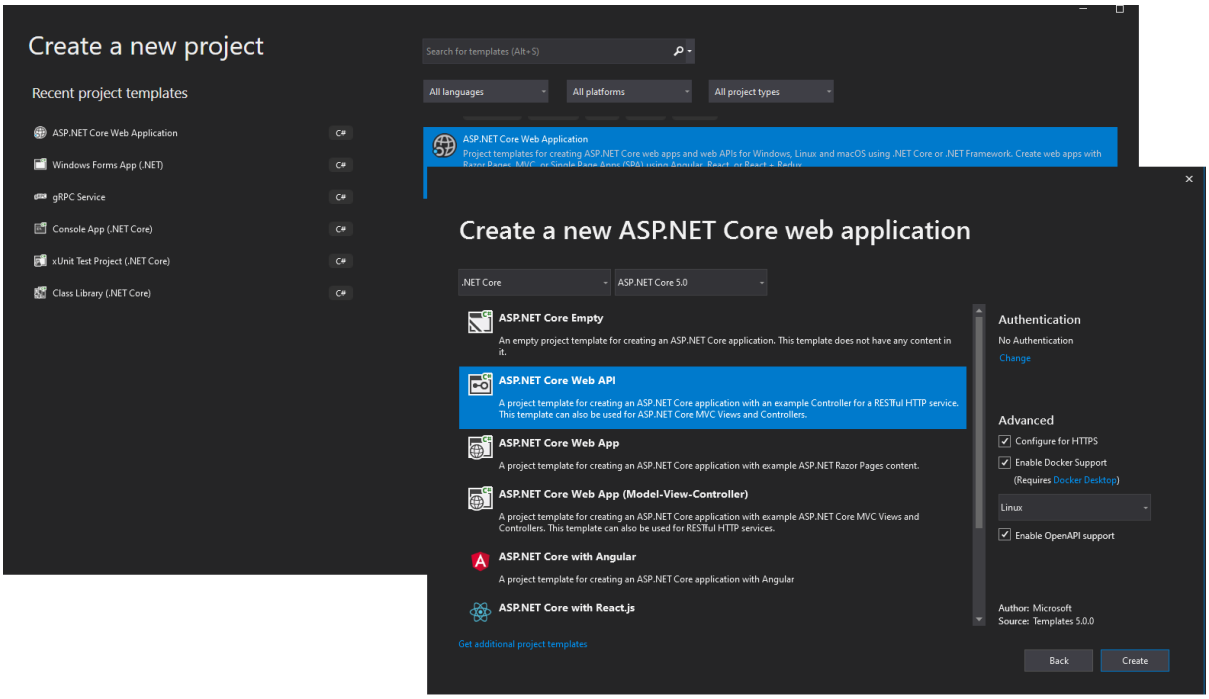


Figure 6-6. Creating an ASP.NET Core Web API project in Visual Studio 2019

To create an ASP.NET Core Web API Project, first select an ASP.NET Core Web Application and then select the API type. After creating the project, you can implement your MVC controllers as you would in any other Web API project, using the Entity Framework API or other API. In a new Web API project, you can see that the only dependency you have in that microservice is on ASP.NET Core itself. Internally, within the *Microsoft.AspNetCore.All* dependency, it is referencing Entity Framework and many other .NET NuGet packages, as shown in Figure 6-7.

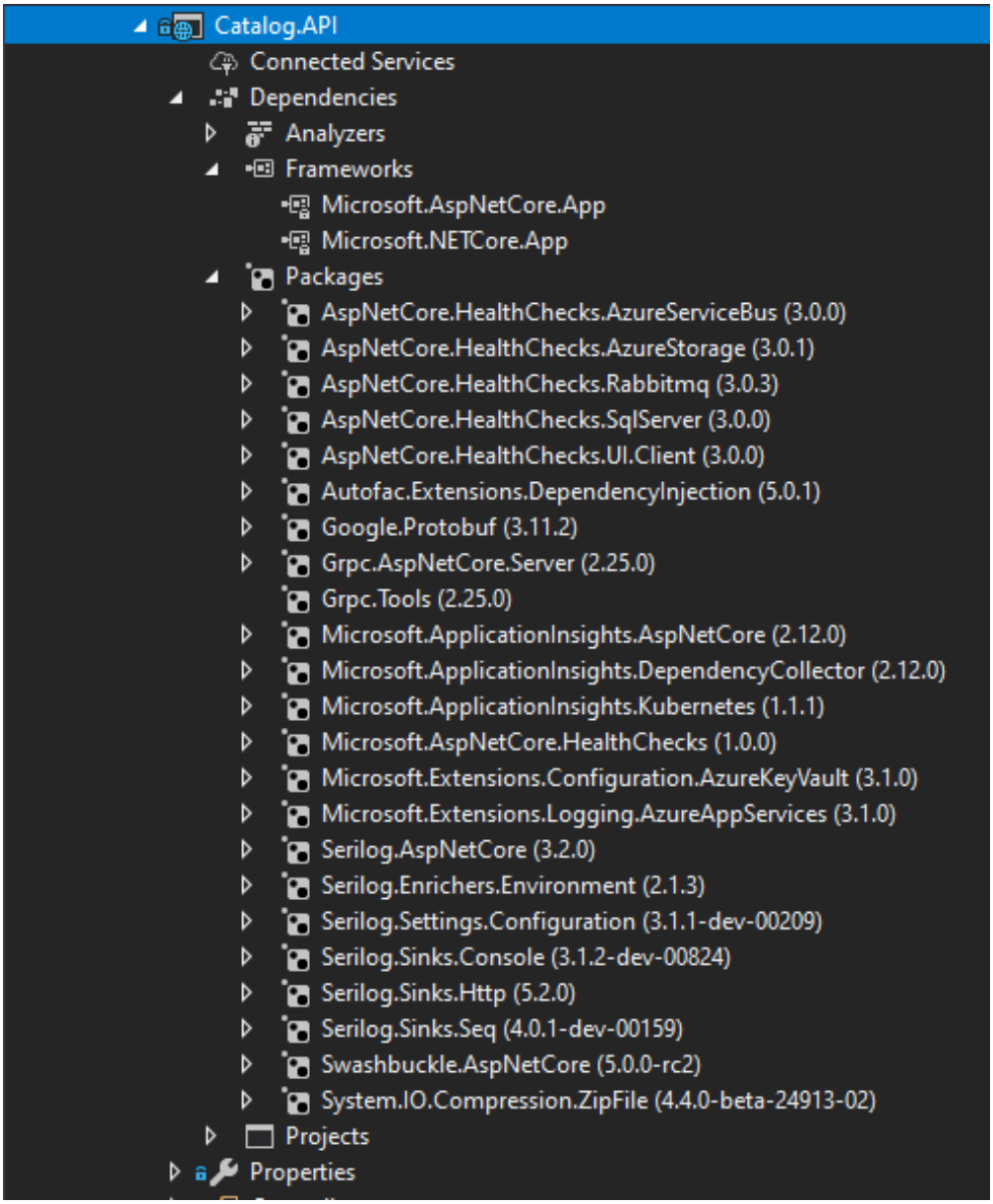


Figure 6-7. Dependencies in a simple CRUD Web API microservice

The API project includes references to Microsoft.AspNetCore.App NuGet package, that includes references to all essential packages. It could include some other packages as well.

Implementing CRUD Web API services with Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology. EF Core is an object-relational mapper (ORM) that enables .NET developers to work with a database using .NET objects.

The catalog microservice uses EF and the SQL Server provider because its database is running in a container with the SQL Server for Linux Docker image. However, the database could be deployed into

any SQL Server, such as Windows on-premises or Azure SQL DB. The only thing you would need to change is the connection string in the ASP.NET Web API microservice.

The data model

With EF Core, data access is performed by using a model. A model is made up of (domain model) entity classes and a derived context (DbContext) that represents a session with the database, allowing you to query and save data. You can generate a model from an existing database, manually code a model to match your database, or use EF migrations technique to create a database from your model, using the code-first approach (that makes it easy to evolve the database as your model changes over time). For the catalog microservice, the last approach has been used. You can see an example of the CatalogItem entity class in the following code example, which is a simple Plain Old Class Object (POCO) entity class.

```
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 ...
}
```

You also need a DbContext that represents a session with the database. For the catalog microservice, the CatalogContext class derives from the DbContext base class, as shown in the following example:

```
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 ...
}
```

You can have additional DbContext implementations. For example, in the sample Catalog.API microservice, there's a second DbContext named CatalogContextSeed where it automatically populates the sample data the first time it tries to access the database. This method is useful for demo data and for automated testing scenarios, as well.

Within the DbContext, you use the OnModelCreating method to customize object/database entity mappings and other EF extensibility points.

Querying data from Web API controllers

Instances of your entity classes are typically retrieved from the database using Language-Integrated Query (LINQ), as shown in the following example:

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

Saving data

Data is created, deleted, and modified in the database using instances of your entity classes. You could add code like the following hard-coded example (mock data, in this case) to your Web API controllers.

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

Dependency Injection in ASP.NET Core and Web API controllers

In ASP.NET Core, you can use Dependency Injection (DI) out of the box. You do not need to set up a third-party Inversion of Control (IoC) container, although you can plug your preferred IoC container into the ASP.NET Core infrastructure if you want. In this case, it means that you can directly inject the required EF DBContext or additional repositories through the controller constructor.

In the CatalogController class mentioned earlier, CatalogContext (which inherits from DbContext) type is injected along with the other required objects in the CatalogController() constructor.

An important configuration to set up in the Web API project is the DbContext class registration into the service's IoC container. You typically do so in the *Program.cs* file by calling the builder.Services.AddDbContext<CatalogContext>() method, as shown in the following **simplified** example:

```
// Additional code...

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

Additional resources

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

The DB connection string and environment variables used by Docker containers

You can use the ASP.NET Core settings and add a ConnectionString property to your settings.json file as shown in the following example:

```
{
    "ConnectionString": "Server=tcp:127.0.0.1,5433;Initial
Catalog=Microsoft.eShopOnContainers.Services.CatalogDb;User Id=sa;Password=[PLACEHOLDER]",
    "ExternalCatalogBaseUrl": "http://host.docker.internal:5101",
    "Logging": {
        "IncludeScopes": false,
        "LogLevel": {
            "Default": "Debug",
            "System": "Information",
            "Microsoft": "Information"
        }
    }
}
```

The settings.json file can have default values for the ConnectionString property or for any other property. However, those properties will be overridden by the values of environment variables that you specify in the docker-compose.override.yml file, when using Docker.

From your docker-compose.yml or docker-compose.override.yml files, you can initialize those environment variables so that Docker will set them up as OS environment variables for you, as shown in the following docker-compose.override.yml file (the connection string and other lines wrap in this example, but it would not wrap in your own file).

```
# docker-compose.override.yml

#
catalog-api:
  environment:
    -
ConnectionString=Server=sqldata;Database=Microsoft.eShopOnContainers.Services.CatalogDb;Use
r Id=sa;Password=[PLACEHOLDER]
    # Additional environment variables for this service
  ports:
    - "5101:80"
```

The docker-compose.yml files at the solution level are not only more flexible than configuration files at the project or microservice level, but also more secure if you override the environment variables declared at the docker-compose files with values set from your deployment tools, like from Azure DevOps Services Docker deployment tasks.

Finally, you can get that value from your code by using builder.Configuration\["ConnectionString"\], as shown in an earlier code example.

However, for production environments, you might want to explore additional ways on how to store secrets like the connection strings. An excellent way to manage application secrets is using Azure Key Vault.

Azure Key Vault helps to store and safeguard cryptographic keys and secrets used by your cloud applications and services. A secret is anything you want to keep strict control of, like API keys, connection strings, passwords, etc. and strict control includes usage logging, setting expiration, managing access, *among others*.

Azure Key Vault allows a detailed control level of the application secrets usage without the need to let anyone know them. The secrets can even be rotated for enhanced security without disrupting development or operations.

Applications have to be registered in the organization's Active Directory, so they can use the Key Vault.

You can check the *Key Vault Concepts documentation* for more details.

Implementing versioning in ASP.NET Web APIs

As business requirements change, new collections of resources may be added, the relationships between resources might change, and the structure of the data in resources might be amended. Updating a Web API to handle new requirements is a relatively straightforward process, but you must consider the effects that such changes will have on client applications consuming the Web API. Although the developer designing and implementing a Web API has full control over that API, the developer does not have the same degree of control over client applications that might be built by third-party organizations operating remotely.

Versioning enables a Web API to indicate the features and resources that it exposes. A client application can then submit requests to a specific version of a feature or resource. There are several approaches to implement versioning:

- URI versioning
- Query string versioning
- Header versioning

Query string and URI versioning are the simplest to implement. Header versioning is a good approach. However, header versioning is not as explicit and straightforward as URI versioning. Because URL versioning is the simplest and most explicit, the eShopOnContainers sample application uses URI versioning.

With URI versioning, as in the eShopOnContainers sample application, each time you modify the Web API or change the schema of resources, you add a version number to the URI for each resource. Existing URIs should continue to operate as before, returning resources that conform to the schema that matches the requested version.

As shown in the following code example, the version can be set by using the Route attribute in the Web API controller, which makes the version explicit in the URI (v1 in this case).

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

This versioning mechanism is simple and depends on the server routing the request to the appropriate endpoint. However, for a more sophisticated versioning and the best method when using REST, you should use hypermedia and implement HATEOAS (Hypertext as the Engine of Application State).

Additional resources

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

Generating Swagger description metadata from your ASP.NET Core Web API

Swagger is a commonly used open source framework backed by a large ecosystem of tools that helps you design, build, document, and consume your RESTful APIs. It is becoming the standard for the APIs description metadata domain. You should include Swagger description metadata with any kind of microservice, either data-driven microservices or more advanced domain-driven microservices (as explained in the following section).

The heart of Swagger is the Swagger specification, which is API description metadata in a JSON or YAML file. The specification creates the RESTful contract for your API, detailing all its resources and operations in both a human- and machine-readable format for easy development, discovery, and integration.

The specification is the basis of the OpenAPI Specification (OAS) and is developed in an open, transparent, and collaborative community to standardize the way RESTful interfaces are defined.

The specification defines the structure for how a service can be discovered and how its capabilities understood. For more information, including a web editor and examples of Swagger specifications from companies like Spotify, Uber, Slack, and Microsoft, see the Swagger site (https://swagger.io).

Why use Swagger?

The main reasons to generate Swagger metadata for your APIs are the following.

Ability for other products to automatically consume and integrate your APIs. Dozens of products and commercial tools and many libraries and frameworks support Swagger. Microsoft has high-level products and tools that can automatically consume Swagger-based APIs, such as the following:

- AutoRest. You can automatically generate .NET client classes for calling Swagger. This tool can be used from the CLI and it also integrates with Visual Studio for easy use through the GUI.
- Microsoft Flow. You can automatically use and integrate your API into a high-level Microsoft Flow workflow, with no programming skills required.
- Microsoft PowerApps. You can automatically consume your API from PowerApps mobile apps built with PowerApps Studio, with no programming skills required.
- Azure App Service Logic Apps. You can automatically use and integrate your API into an Azure App Service Logic App, with no programming skills required.

Ability to automatically generate API documentation. When you create large-scale RESTful APIs, such as complex microservice-based applications, you need to handle many endpoints with different data models used in the request and response payloads. Having proper documentation and having a solid API explorer, as you get with Swagger, is key for the success of your API and adoption by developers.

Swagger's metadata is what Microsoft Flow, PowerApps, and Azure Logic Apps use to understand how to use APIs and connect to them.

There are several options to automate Swagger metadata generation for ASP.NET Core REST API applications, in the form of functional API help pages, based on *swagger-ui*.

Probably the best know is Swashbuckle, which is currently used in eShopOnContainers and we'll cover in some detail in this guide but there's also the option to use NSwag, which can generate Typescript and C# API clients, as well as C# controllers, from a Swagger or OpenAPI specification and even by scanning the .dll that contains the controllers, using NSwagStudio.

How to automate API Swagger metadata generation with the Swashbuckle NuGet package

Generating Swagger metadata manually (in a JSON or YAML file) can be tedious work. However, you can automate API discovery of ASP.NET Web API services by using the Swashbuckle NuGet package to dynamically generate Swagger API metadata.

Swashbuckle automatically generates Swagger metadata for your ASP.NET Web API projects. It supports ASP.NET Core Web API projects and the traditional ASP.NET Web API and any other flavor,

such as Azure API App, Azure Mobile App, Azure Service Fabric microservices based on ASP.NET. It also supports plain Web API deployed on containers, as in for the reference application.

Swashbuckle combines API Explorer and Swagger or swagger-ui to provide a rich discovery and documentation experience for your API consumers. In addition to its Swagger metadata generator engine, Swashbuckle also contains an embedded version of swagger-ui, which it will automatically serve up once Swashbuckle is installed.

This means you can complement your API with a nice discovery UI to help developers to use your API. It requires a small amount of code and maintenance because it is automatically generated, allowing you to focus on building your API. The result for the API Explorer looks like Figure 6-8.

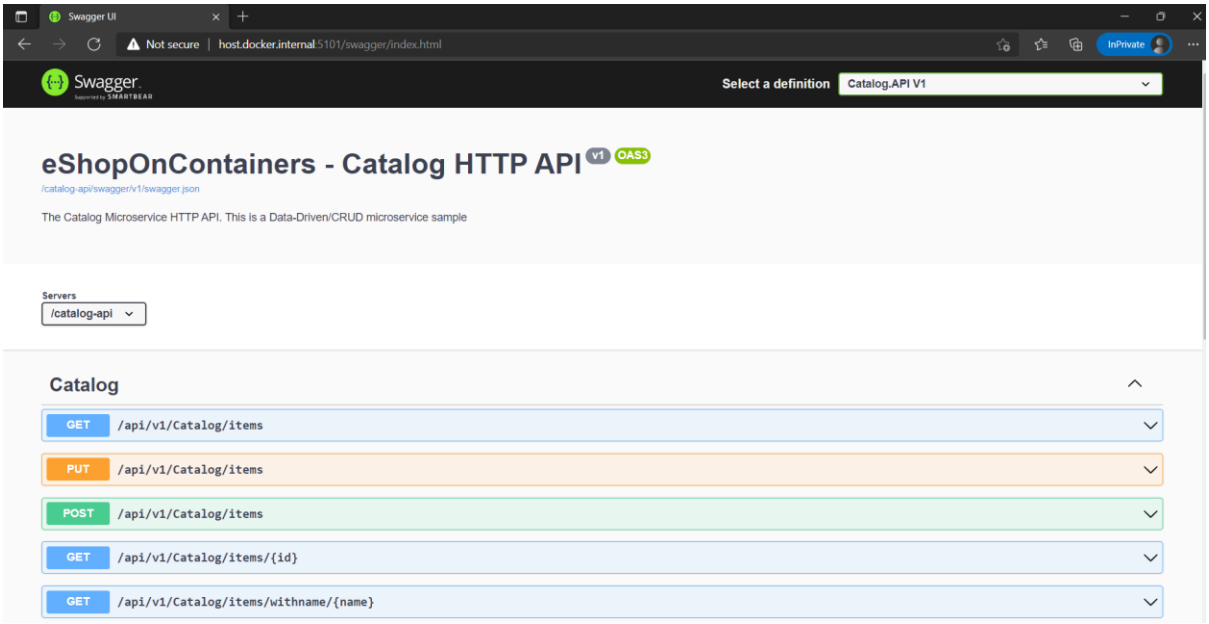


Figure 6-8. Swashbuckle API Explorer based on Swagger metadata—eShopOnContainers catalog microservice

The Swashbuckle generated Swagger UI API documentation includes all published actions. The API explorer is not the most important thing here. Once you have a Web API that can describe itself in Swagger metadata, your API can be used seamlessly from Swagger-based tools, including client proxy-class code generators that can target many platforms. For example, as mentioned, AutoRest automatically generates .NET client classes. But additional tools like swagger-codegen are also available, which allow code generation of API client libraries, server stubs, and documentation automatically.

Currently, Swashbuckle consists of five internal NuGet packages under the high-level metapackage Swashbuckle.AspNetCore for ASP.NET Core applications.

After you have installed these NuGet packages in your Web API project, you need to configure Swagger in the *Program.cs* class, as in the following **simplified** code:

```
// Add framework services.

builder.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 startup code...

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

Once this is done, you can start your application and browse the following Swagger JSON and
UI endpoints using URLs like these:

:::{custom-style=CodeBox}
```console
  http://<your-root-url>/swagger/v1/swagger.json

  http://<your-root-url>/swagger/
```
```

You previously saw the generated UI created by Swashbuckle for a URL like http://<your-root-url>/swagger. In Figure 6-9, you can also see how you can test any API method.
```

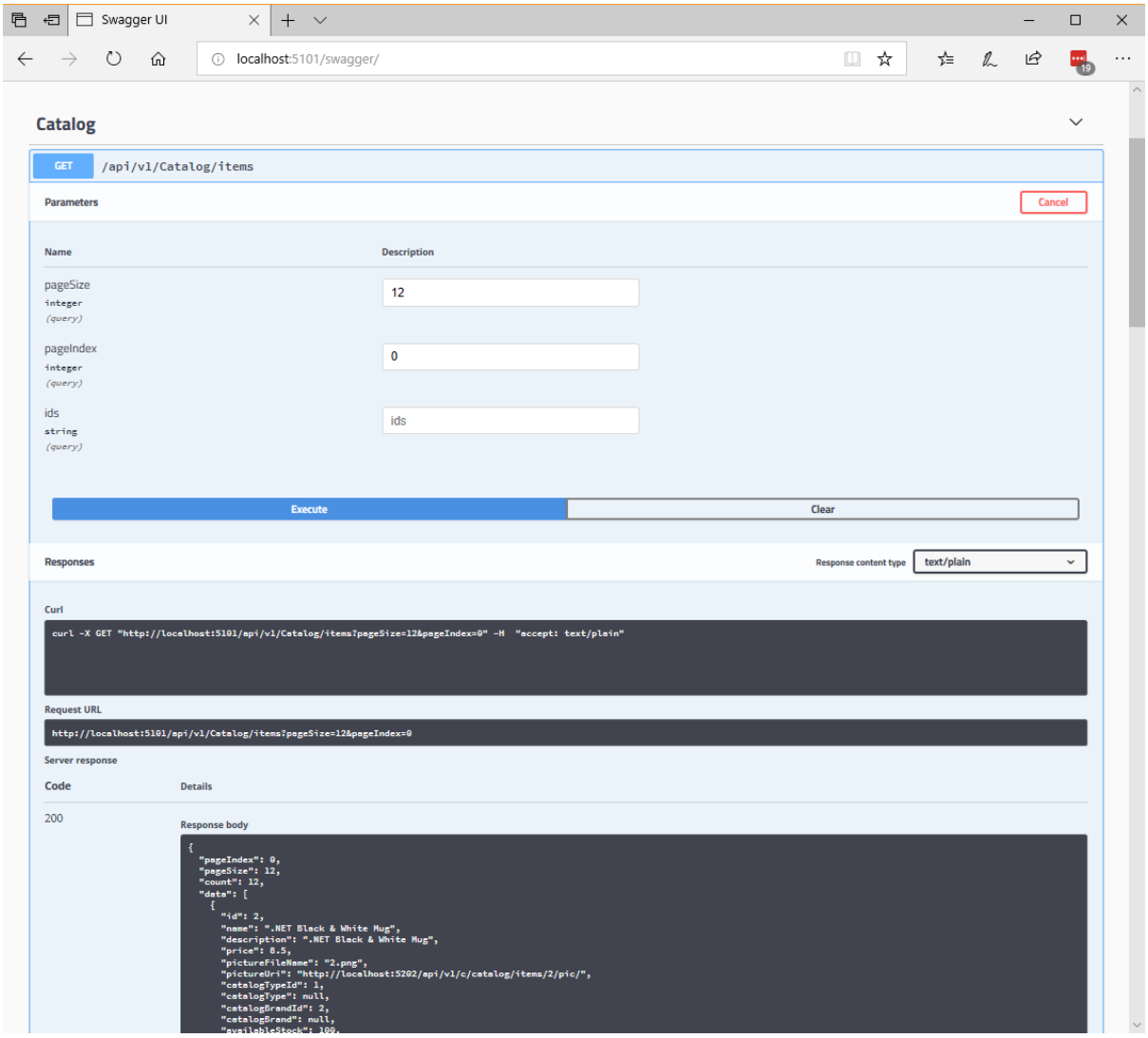


Figure 6-9. Swashbuckle UI testing the Catalog/Items API method

The Swagger UI API detail shows a sample of the response and can be used to execute the real API, which is great for developer discovery. Figure 6-10 shows the Swagger JSON metadata generated from the eShopOnContainers microservice (which is what the tools use underneath) when you request http://<your-root-url>/swagger/v1/swagger.json using Postman.

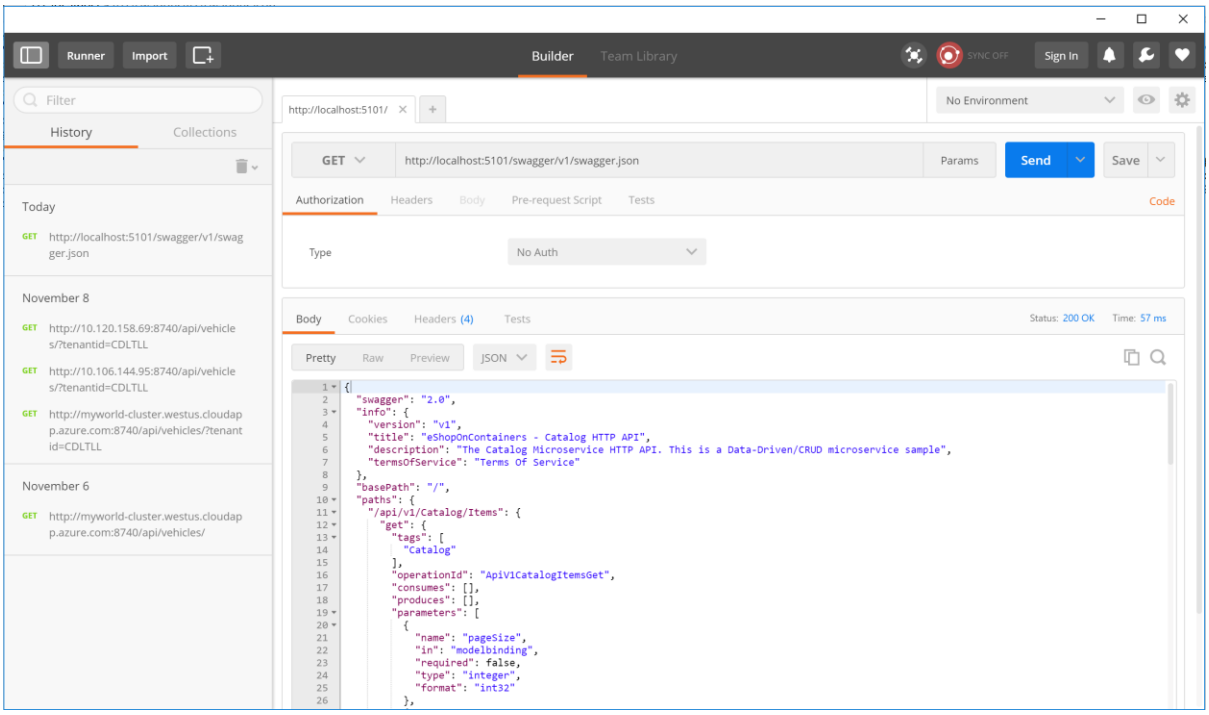


Figure 6-10. Swagger JSON metadata

It is that simple. And because it is automatically generated, the Swagger metadata will grow when you add more functionality to your API.

Additional resources

- **ASP.NET Web API Help Pages using Swagger**
  https://learn.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger
- **Get started with Swashbuckle and ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/tutorials/getting-started-with-swashbuckle
- **Get started with NSwag and ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/tutorials/getting-started-with-nswag

Defining your multi-container application with docker-compose.yml

In this guide, the docker-compose.yml file was introduced in the section Step 4. Define your services in docker-compose.yml when building a multi-container Docker application. However, there are additional ways to use the docker-compose files that are worth exploring in further detail.

For example, you can explicitly describe how you want to deploy your multi-container application in the docker-compose.yml file. Optionally, you can also describe how you are going to build your custom Docker images. (Custom Docker images can also be built with the Docker CLI.)

Basically, you define each of the containers you want to deploy plus certain characteristics for each container deployment. Once you have a multi-container deployment description file, you can deploy the whole solution in a single action orchestrated by the docker-compose up CLI command, or you can deploy it transparently from Visual Studio. Otherwise, you would need to use the Docker CLI to deploy container-by-container in multiple steps by using the docker run command from the command line. Therefore, each service defined in docker-compose.yml must specify exactly one image or build. Other keys are optional, and are analogous to their docker run command-line counterparts.

The following YAML code is the definition of a possible global but single docker-compose.yml file for the eShopOnContainers sample. This code is not the actual docker-compose file from eShopOnContainers. Instead, it is a simplified and consolidated version in a single file, which is not the best way to work with docker-compose files, as will be explained later.

```
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=[PLACEHOLDER]
    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=[PLACEHOLDER]
    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=[PLACEHOLDER]
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"

  basketdata:
    image: redis
```

The root key in this file is services. Under that key, you define the services you want to deploy and run when you execute the docker-compose up command or when you deploy from Visual Studio by using this docker-compose.yml file. In this case, the docker-compose.yml file has multiple services defined, as described in the following table.

Service name	Description
webmvc	Container including the ASP.NET Core MVC application consuming the microservices from server-side C#
catalog-api	Container including the Catalog ASP.NET Core Web API microservice
ordering-api	Container including the Ordering ASP.NET Core Web API microservice
sqldata	Container running SQL Server for Linux, holding the microservices databases
basket-api	Container with the Basket ASP.NET Core Web API microservice
basketdata	Container running the REDIS cache service, with the basket database as a REDIS cache

A simple Web Service API container

Focusing on a single container, the catalog-api container-microservice has a straightforward definition:

```
  catalog-api:
    image: eshop/catalog-api
    environment:
      - ConnectionString=Server=sqldata;Initial Catalog=CatalogData;User
Id=sa;Password=[PLACEHOLDER]
    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
```

This containerized service has the following basic configuration:

- It is based on the custom **eshop/catalog-api** image. For simplicity's sake, there is no build: key setting in the file. This means that the image must have been previously built (with docker build) or have been downloaded (with the docker pull command) from any Docker registry.
- It defines an environment variable named ConnectionString with the connection string to be used by Entity Framework to access the SQL Server instance that contains the catalog data model. In this case, the same SQL Server container is holding multiple databases. Therefore, you need less memory in your development machine for Docker. However, you could also deploy one SQL Server container for each microservice database.
- The SQL Server name is **sqldata**, which is the same name used for the container that is running the SQL Server instance for Linux. This is convenient; being able to use this name resolution (internal to the Docker host) will resolve the network address so you don't need to know the internal IP for the containers you are accessing from other containers.

Because the connection string is defined by an environment variable, you could set that variable through a different mechanism and at a different time. For example, you could set a different connection string when deploying to production in the final hosts, or by doing it from your CI/CD pipelines in Azure DevOps Services or your preferred DevOps system.

- It exposes port 80 for internal access to the **catalog-api** service within the Docker host. The host is currently a Linux VM because it is based on a Docker image for Linux, but you could configure the container to run on a Windows image instead.
- It forwards the exposed port 80 on the container to port 5101 on the Docker host machine (the Linux VM).
- It links the web service to the **sqldata** service (the SQL Server instance for Linux database running in a container). When you specify this dependency, the catalog-api container will not start until the sqldata container has already started; this aspect is important because catalog-api needs to have the SQL Server database up and running first. However, this kind of container dependency is not enough in many cases, because Docker checks only at the container level. Sometimes the service (in this case SQL Server) might still not be ready, so it is advisable to implement retry logic with exponential backoff in your client microservices. That way, if a dependency container is not ready for a short time, the application will still be resilient.
- It is configured to allow access to external servers: the extra_hosts setting allows you to access external servers or machines outside of the Docker host (that is, outside the default Linux VM,

which is a development Docker host), such as a local SQL Server instance on your development PC.

There are also other, more advanced docker-compose.yml settings that we'll discuss in the following sections.

Using docker-compose files to target multiple environments

The docker-compose.*.yml files are definition files and can be used by multiple infrastructures that understand that format. The most straightforward tool is the docker-compose command.

Therefore, by using the docker-compose command you can target the following main scenarios.

Development environments

When you develop applications, it is important to be able to run an application in an isolated development environment. You can use the docker-compose CLI command to create that environment or Visual Studio, which uses docker-compose under the covers.

The docker-compose.yml file allows you to configure and document all your application's service dependencies (other services, cache, databases, queues, etc.). Using the docker-compose CLI command, you can create and start one or more containers for each dependency with a single command (docker-compose up).

The docker-compose.yml files are configuration files interpreted by Docker engine but also serve as convenient documentation files about the composition of your multi-container application.

Testing environments

An important part of any continuous deployment (CD) or continuous integration (CI) process are the unit tests and integration tests. These automated tests require an isolated environment so they are not impacted by the users or any other change in the application's data.

With Docker Compose, you can create and destroy that isolated environment very easily in a few commands from your command prompt or scripts, like the following commands:

```
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
```

Production deployments

You can also use Compose to deploy to a remote Docker Engine. A typical case is to deploy to a single Docker host instance (like a production VM or server provisioned with Docker Machine).

If you are using any other orchestrator (Azure Service Fabric, Kubernetes, etc.), you might need to add setup and metadata configuration settings like those in docker-compose.yml, but in the format required by the other orchestrator.

In any case, docker-compose is a convenient tool and metadata format for development, testing and production workflows, although the production workflow might vary on the orchestrator you are using.

Using multiple docker-compose files to handle several environments

When targeting different environments, you should use multiple compose files. This approach lets you create multiple configuration variants depending on the environment.

Overriding the base docker-compose file

You could use a single docker-compose.yml file as in the simplified examples shown in previous sections. However, that is not recommended for most applications.

By default, Compose reads two files, a docker-compose.yml and an optional docker-compose.override.yml file. As shown in Figure 6-11, when you are using Visual Studio and enabling Docker support, Visual Studio also creates an additional docker-compose.vs.debug.g.yml file for debugging the application, you can take a look at this file in folder obj\Docker\ in the main solution folder.

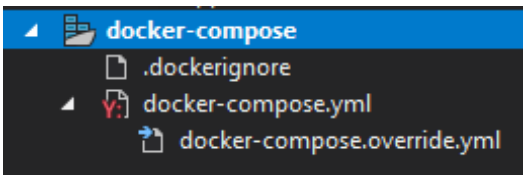


Figure 6-11. docker-compose files in Visual Studio 2019

docker-compose project file structure:

- *.dockerignore* - used to ignore files
- *docker-compose.yml* - used to compose microservices
- *docker-compose.override.yml* - used to configure microservices environment

You can edit the docker-compose files with any editor, like Visual Studio Code or Sublime, and run the application with the docker-compose up command.

By convention, the docker-compose.yml file contains your base configuration and other static settings. That means that the service configuration should not change depending on the deployment environment you are targeting.

The docker-compose.override.yml file, as its name suggests, contains configuration settings that override the base configuration, such as configuration that depends on the deployment environment. You can have multiple override files with different names also. The override files usually contain additional information needed by the application but specific to an environment or to a deployment.

Targeting multiple environments

A typical use case is when you define multiple compose files so you can target multiple environments, like production, staging, CI, or development. To support these differences, you can split your Compose configuration into multiple files, as shown in Figure 6-12.

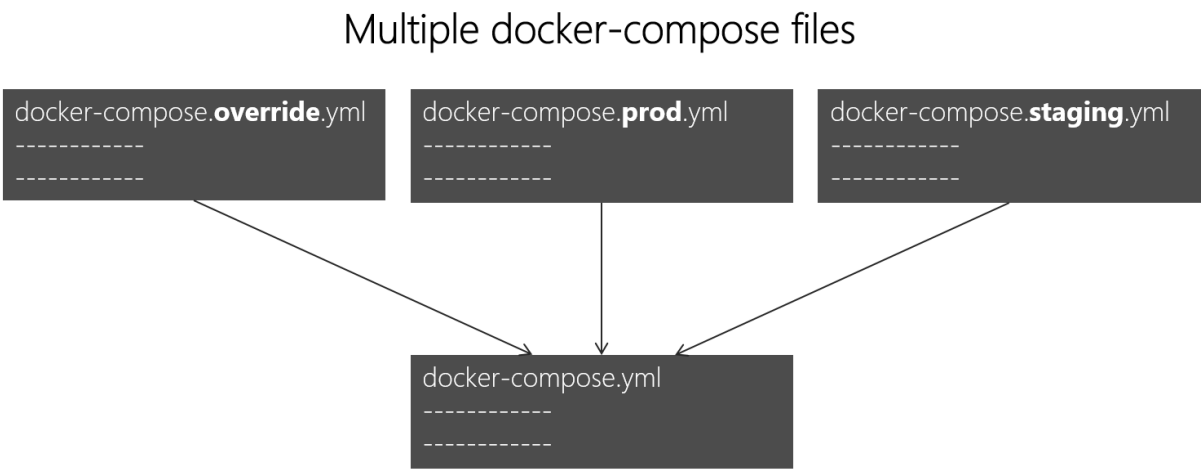


Figure 6-12. Multiple docker-compose files overriding values in the base docker-compose.yml file

You can combine multiple docker-compose*.yml files to handle different environments. You start with the base docker-compose.yml file. This base file contains the base or static configuration settings that do not change depending on the environment. For example, the eShopOnContainers app has the following docker-compose.yml file (simplified with fewer services) as the base file.

```
#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:2019-latest

  nosqldata:
    image: mongo

  basketdata:
    image: redis

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

The values in the base docker-compose.yml file should not change because of different target deployment environments.

If you focus on the webmvc service definition, for instance, you can see how that information is much the same no matter what environment you might be targeting. You have the following information:

- The service name: webmvc.
- The container's custom image: eshop/webmvc.
- The command to build the custom Docker image, indicating which Dockerfile to use.
- Dependencies on other services, so this container does not start until the other dependency containers have started.

You can have additional configuration, but the important point is that in the base docker-compose.yml file, you just want to set the information that is common across environments. Then in the docker-compose.override.yml or similar files for production or staging, you should place configuration that is specific for each environment.

Usually, the docker-compose.override.yml is used for your development environment, as in the following example from 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=${ESHOP_AZURE_CATALOG_DB:-
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.CatalogDb;User
Id=sa;Password=[PLACEHOLDER]}
      - PicBaseUrl=${ESHOP_AZURE_STORAGE_CATALOG_URL:-
http://host.docker.internal:5202/api/v1/catalog/items/[0]/pic/}
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - AzureStorageAccountName=${ESHOP_AZURE_STORAGE_CATALOG_NAME}
      - AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_CATALOG_KEY}
      - UseCustomizationData=True
      - AzureServiceBusEnabled=False
      - AzureStorageEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${ORCHESTRATOR_TYPE}
    ports:
      - "5101:80"

  marketing-api:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://0.0.0.0:80
      - ConnectionString=${ESHOP_AZURE_MARKETING_DB:-
Server=sqldata;Database=Microsoft.eShopOnContainers.Services.MarketingDb;User
Id=sa;Password=[PLACEHOLDER]}
      - MongoConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}
      - MongoDatabase=MarketingDb
      - EventBusConnection=${ESHOP_AZURE_SERVICE_BUS:-rabbitmq}
      - EventBusUserName=${ESHOP_SERVICE_BUS_USERNAME}
      - EventBusPassword=${ESHOP_SERVICE_BUS_PASSWORD}
      - identityUrl=http://identity-api
      - IdentityUrlExternal=http://${ESHOP_EXTERNAL_DNS_NAME_OR_IP}:5105
      - CampaignDetailFunctionUri=${ESHOP_AZUREFUNC_CAMPAIGN_DETAILS_URI}
      - PicBaseUrl=${ESHOP_AZURE_STORAGE_MARKETING_URL:-
http://host.docker.internal:5110/api/v1/campaigns/[0]/pic/}
      - AzureStorageAccountName=${ESHOP_AZURE_STORAGE_MARKETING_NAME}
      - AzureStorageAccountKey=${ESHOP_AZURE_STORAGE_MARKETING_KEY}
      - AzureServiceBusEnabled=False
      - AzureStorageEnabled=False
      - ApplicationInsights__InstrumentationKey=${INSTRUMENTATION_KEY}
      - OrchestratorType=${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=[PLACEHOLDER]
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
  nosqldata:
    ports:
      - "27017:27017"
  basketdata:
    ports:
      - "6379:6379"
  rabbitmq:
    ports:
      - "15672:15672"
      - "5672:5672"
```

In this example, the development override configuration exposes some ports to the host, defines environment variables with redirect URLs, and specifies connection strings for the development environment. These settings are all just for the development environment.

When you run docker-compose up (or launch it from Visual Studio), the command reads the overrides automatically as if it were merging both files.

Suppose that you want another Compose file for the production environment, with different configuration values, ports, or connection strings. You can create another override file, like file named docker-compose.prod.yml with different settings and environment variables. That file might be stored in a different Git repo or managed and secured by a different team.

How to deploy with a specific override file

To use multiple override files, or an override file with a different name, you can use the -f option with the docker-compose command and specify the files. Compose merges files in the order they are specified on the command line. The following example shows how to deploy with override files.

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

Using environment variables in docker-compose files

It is convenient, especially in production environments, to be able to get configuration information from environment variables, as we have shown in previous examples. You can reference an environment variable in your docker-compose files using the syntax ${MY_VAR}. The following line from a docker-compose.prod.yml file shows how to reference the value of an environment variable.

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

Environment variables are created and initialized in different ways, depending on your host environment (Linux, Windows, Cloud cluster, etc.). However, a convenient approach is to use an .env file. The docker-compose files support declaring default environment variables in the .env file. These values for the environment variables are the default values. But they can be overridden by the values you might have defined in each of your environments (host OS or environment variables from your cluster). You place this .env file in the folder where the docker-compose command is executed from.

The following example shows an .env file like the .env file for the eShopOnContainers application.

```
# .env file

ESHOP_EXTERNAL_DNS_NAME_OR_IP=host.docker.internal

ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=10.121.122.92
```

Docker-compose expects each line in an .env file to be in the format <variable>=<value>.

The values set in the run-time environment always override the values defined inside the .env file. In a similar way, values passed via command-line arguments also override the default values set in the .env file.

Additional resources

- **Overview of Docker Compose**
  https://docs.docker.com/compose/overview/
- **Multiple Compose files**
  https://docs.docker.com/compose/multiple-compose-files/

Building optimized ASP.NET Core Docker images

If you are exploring Docker and .NET on sources on the Internet, you will find Dockerfiles that demonstrate the simplicity of building a Docker image by copying your source into a container. These examples suggest that by using a simple configuration, you can have a Docker image with the environment packaged with your application. The following example shows a simple Dockerfile in this vein.

```
FROM mcr.microsoft.com/dotnet/sdk:7.0
WORKDIR /app
ENV ASPNETCORE_URLS http://+:80
EXPOSE 80
COPY . .
RUN dotnet restore
ENTRYPOINT ["dotnet", "run"]
```

A Dockerfile like this will work. However, you can substantially optimize your images, especially your production images.

In the container and microservices model, you are constantly starting containers. The typical way of using containers does not restart a sleeping container, because the container is disposable. Orchestrators (like Kubernetes and Azure Service Fabric) create new instances of images. What this means is that you would need to optimize by precompiling the application when it is built so the instantiation process will be faster. When the container is started, it should be ready to run. Don't restore and compile at run time using the dotnet restore and dotnet build CLI commands as you may see in blog posts about .NET and Docker.

The .NET team has been doing important work to make .NET and ASP.NET Core a container-optimized framework. Not only is .NET a lightweight framework with a small memory footprint; the team has focused on optimized Docker images for three main scenarios and published them in the Docker Hub registry at *dotnet/*, beginning with version 2.1:

1. **Development**: The priority is the ability to quickly iterate and debug changes, and where size is secondary.
2. **Build**: The priority is compiling the application, and the image includes binaries and other dependencies to optimize binaries.
3. **Production**: The focus is fast deploying and starting of containers, so these images are limited to the binaries and content needed to run the application.

The .NET team provides four basic variants in dotnet/ (at Docker Hub):

1. **sdk**: for development and build scenarios
2. **aspnet**: for ASP.NET production scenarios
3. **runtime**: for .NET production scenarios
4. **runtime-deps**: for production scenarios of self-contained applications

For faster startup, runtime images also automatically set aspnetcore_urls to port 80 and use Ngen to create a native image cache of assemblies.

Additional resources

- **Building Optimized Docker Images with ASP.NET Core**
  https://learn.microsoft.com/archive/blogs/stevelasker/building-optimized-docker-images-with-asp-net-core
- **Building Docker Images for .NET Applications**
  https://learn.microsoft.com/dotnet/core/docker/building-net-docker-images

Use a database server running as a container

You can have your databases (SQL Server, PostgreSQL, MySQL, etc.) on regular standalone servers, in on-premises clusters, or in PaaS services in the cloud like Azure SQL DB. However, for development and test environments, having your databases running as containers is convenient, because you don't

have any external dependency and simply running the docker-compose up command starts the whole application. Having those databases as containers is also great for integration tests, because the database is started in the container and is always populated with the same sample data, so tests can be more predictable.

SQL Server running as a container with a microservice-related database

In eShopOnContainers, there's a container named sqldata, as defined in the docker-compose.yml file, that runs a SQL Server for Linux instance with the SQL databases for all microservices that need one.

A key point in microservices is that each microservice owns its related data, so it should have its own database. However, the databases can be anywhere. In this case, they are all in the same container to keep Docker memory requirements as low as possible. Keep in mind that this is a good-enough solution for development and, perhaps, testing but not for production.

The SQL Server container in the sample application is configured with the following YAML code in the docker-compose.yml file, which is executed when you run docker-compose up. Note that the YAML code has consolidated configuration information from the generic docker-compose.yml file and the docker-compose.override.yml file. (Usually you would separate the environment settings from the base or static information related to the SQL Server image.)

```
  sqldata:
    image: mcr.microsoft.com/mssql/server:2017-latest
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"
```

In a similar way, instead of using docker-compose, the following docker run command can run that container:

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

However, if you are deploying a multi-container application like eShopOnContainers, it is more convenient to use the docker-compose up command so that it deploys all the required containers for the application.

When you start this SQL Server container for the first time, the container initializes SQL Server with the password that you provide. Once SQL Server is running as a container, you can update the database by connecting through any regular SQL connection, such as from SQL Server Management Studio, Visual Studio, or C# code.

The eShopOnContainers application initializes each microservice database with sample data by seeding it with data on startup, as explained in the following section.

Having SQL Server running as a container is not just useful for a demo where you might not have access to an instance of SQL Server. As noted, it is also great for development and testing

environments so that you can easily run integration tests starting from a clean SQL Server image and known data by seeding new sample data.

Additional resources

- **Run the SQL Server Docker image on Linux, Mac, or Windows**
  https://learn.microsoft.com/sql/linux/sql-server-linux-setup-docker
- **Connect and query SQL Server on Linux with sqlcmd**
  https://learn.microsoft.com/sql/linux/sql-server-linux-connect-and-query-sqlcmd

Seeding with test data on Web application startup

To add data to the database when the application starts up, you can add code like the following to the Main method in the Program class of the Web API project:

```
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();
    }
}
```

There's an important caveat when applying migrations and seeding a database during container startup. Since the database server might not be available for whatever reason, you must handle retries while waiting for the server to be available. This retry logic is handled by the MigrateDbContext() extension method, as shown in the following code:

```
public static IWebHost MigrateDbContext<TContext>(
    this IWebHost host,
    Action<TContext,
    IServiceProvider> 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;
}
```

The following code in the custom CatalogContextSeed class populates the data.

```
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" }
        };
    }
}
```

When you run integration tests, having a way to generate data consistent with your integration tests is useful. Being able to create everything from scratch, including an instance of SQL Server running on a container, is great for test environments.

EF Core InMemory database versus SQL Server running as a container

Another good choice when running tests is to use the Entity Framework InMemory database provider. You can specify that configuration in the ConfigureServices method of the Startup class in your Web API project:

```
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 ...
}
```

There is an important catch, though. The in-memory database does not support many constraints that are specific to a particular database. For instance, you might add a unique index on a column in your EF Core model and write a test against your in-memory database to check that it does not let you add a duplicate value. But when you are using the in-memory database, you cannot handle unique indexes on a column. Therefore, the in-memory database does not behave exactly the same as a real SQL Server database—it does not emulate database-specific constraints.

Even so, an in-memory database is still useful for testing and prototyping. But if you want to create accurate integration tests that take into account the behavior of a specific database implementation, you need to use a real database like SQL Server. For that purpose, running SQL Server in a container is a great choice and more accurate than the EF Core InMemory database provider.

Using a Redis cache service running in a container

You can run Redis on a container, especially for development and testing and for proof-of-concept scenarios. This scenario is convenient, because you can have all your dependencies running on containers—not just for your local development machines, but for your testing environments in your CI/CD pipelines.

However, when you run Redis in production, it is better to look for a high-availability solution like Redis Microsoft Azure, which runs as a PaaS (Platform as a Service). In your code, you just need to change your connection strings.

Redis provides a Docker image with Redis. That image is available from Docker Hub at this URL:

https://hub.docker.com/_/redis/

You can directly run a Docker Redis container by executing the following Docker CLI command in your command prompt:

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

The Redis image includes expose:6379 (the port used by Redis), so standard container linking will make it automatically available to the linked containers.

In eShopOnContainers, the basket-api microservice uses a Redis cache running as a container. That basketdata container is defined as part of the multi-container *docker-compose.yml* file, as shown in the following example:

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

This code in the docker-compose.yml defines a container named basketdata based on the redis image and publishing the port 6379 internally. This configuration means that it will only be accessible from other containers running within the Docker host.

Finally, in the *docker-compose.override.yml* file, the basket-api microservice for the eShopOnContainers sample defines the connection string to use for that Redis container:

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

As mentioned before, the name of the microservice basketdata is resolved by Docker's internal network DNS.

Implementing event-based communication between microservices (integration events)

As described earlier, when you use event-based communication, a microservice publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event, it can update its own business entities, which might lead to more events being published. This is the essence of the eventual consistency concept. This publish/subscribe system is usually performed by using an implementation of an event bus. The event bus can be designed as an interface with the API needed to subscribe and unsubscribe to events and to publish events. It can also have one or more implementations based on any inter-process or messaging communication, such as a messaging queue or a service bus that supports asynchronous communication and a publish/subscribe model.

You can use events to implement business transactions that span multiple services, which give you eventual consistency between those services. An eventually consistent transaction consists of a series

of distributed actions. At each action, the microservice updates a business entity and publishes an event that triggers the next action. Figure 6-18 below, shows a PriceUpdated event published through an event bus, so the price update is propagated to the Basket and other microservices.

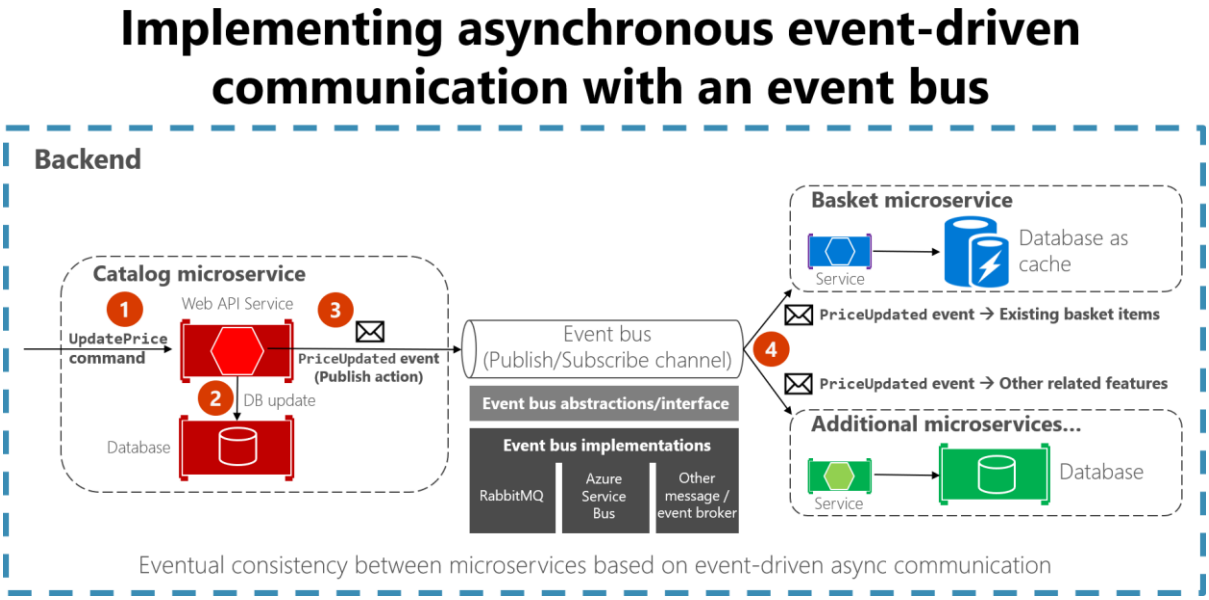


Figure 6-18. Event-driven communication based on an event bus

This section describes how you can implement this type of communication with .NET by using a generic event bus interface, as shown in Figure 6-18. There are multiple potential implementations, each using a different technology or infrastructure such as RabbitMQ, Azure Service Bus, or any other third-party open-source or commercial service bus.

Using message brokers and service buses for production systems

As noted in the architecture section, you can choose from multiple messaging technologies for implementing your abstract event bus. But these technologies are at different levels. For instance, RabbitMQ, a messaging broker transport, is at a lower level than commercial products like Azure Service Bus, NServiceBus, MassTransit, or Brighter. Most of these products can work on top of either RabbitMQ or Azure Service Bus. Your choice of product depends on how many features and how much out-of-the-box scalability you need for your application.

For implementing just an event bus proof-of-concept for your development environment, as in the eShopOnContainers sample, a simple implementation on top of RabbitMQ running as a container might be enough. But for mission-critical and production systems that need high scalability, you might want to evaluate and use Azure Service Bus.

If you require high-level abstractions and richer features like Sagas for long-running processes that make distributed development easier, other commercial and open-source service buses like NServiceBus, MassTransit, and Brighter are worth evaluating. In this case, the abstractions and API to use would usually be directly the ones provided by those high-level service buses instead of your own abstractions (like the simple event bus abstractions provided at eShopOnContainers). For that matter,

you can research the forked eShopOnContainers using NServiceBus (additional derived sample implemented by Particular Software).

Of course, you could always build your own service bus features on top of lower-level technologies like RabbitMQ and Docker, but the work needed to "reinvent the wheel" might be too costly for a custom enterprise application.

To reiterate: the sample event bus abstractions and implementation showcased in the eShopOnContainers sample are intended to be used only as a proof of concept. Once you have decided that you want to have asynchronous and event-driven communication, as explained in the current section, you should choose the service bus product that best fits your needs for production.

Integration events

Integration events are used for bringing domain state in sync across multiple microservices or external systems. This functionality is done by publishing integration events outside the microservice. When an event is published to multiple receiver microservices (to as many microservices as are subscribed to the integration event), the appropriate event handler in each receiver microservice handles the event.

An integration event is basically a data-holding class, as in the following example:

```
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;
    }
}
```

The integration events can be defined at the application level of each microservice, so they are decoupled from other microservices, in a way comparable to how ViewModels are defined in the server and client. What is not recommended is sharing a common integration events library across multiple microservices; doing that would be coupling those microservices with a single event definition data library. You do not want to do that for the same reasons that you do not want to share a common domain model across multiple microservices: microservices must be completely autonomous. For more information, see this blog post on the amount of data to put in events. Be careful not to take this too far, as this other blog post describes the problem data deficient messages can produce. Your design of your events should aim to be "just right" for the needs of their consumers.

There are only a few kinds of libraries you should share across microservices. One is libraries that are final application blocks, like the Event Bus client API, as in eShopOnContainers. Another is libraries that constitute tools that could also be shared as NuGet components, like JSON serializers.

The event bus

An event bus allows publish/subscribe-style communication between microservices without requiring the components to explicitly be aware of each other, as shown in Figure 6-19.

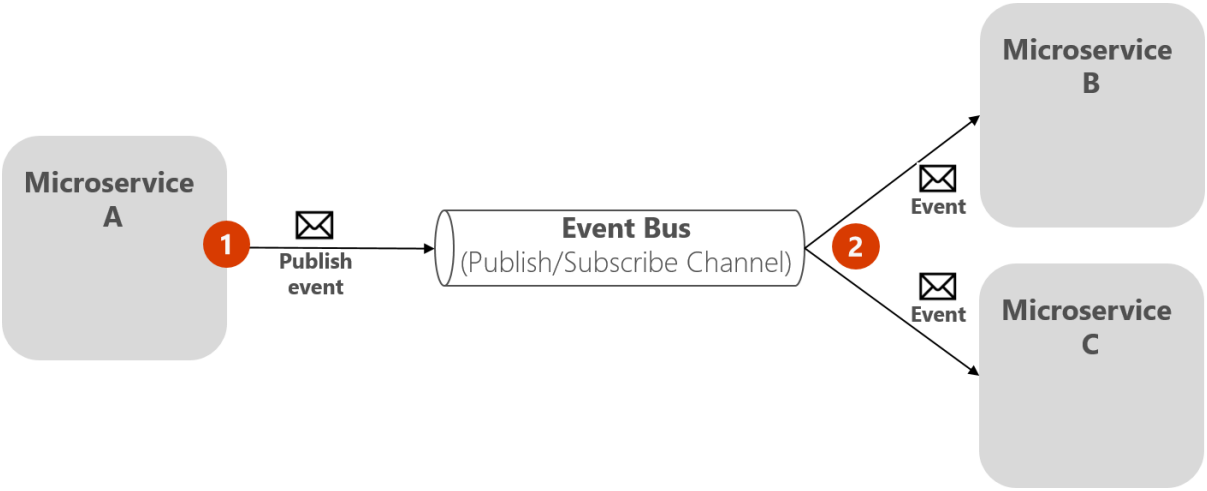


Figure 6-19. Publish/subscribe basics with an event bus

The above diagram shows that microservice A publishes to Event Bus, which distributes to subscribing microservices B and C, without the publisher needing to know the subscribers. The event bus is related to the Observer pattern and the publish-subscribe pattern.

Observer pattern

In the Observer pattern, your primary object (known as the Observable) notifies other interested objects (known as Observers) with relevant information (events).

Publish/Subscribe (Pub/Sub) pattern

The purpose of the Publish/Subscribe pattern is the same as the Observer pattern: you want to notify other services when certain events take place. But there is an important difference between the Observer and Pub/Sub patterns. In the observer pattern, the broadcast is performed directly from the observable to the observers, so they "know" each other. But when using a Pub/Sub pattern, there is a third component, called broker, or message broker or event bus, which is known by both the publisher and subscriber. Therefore, when using the Pub/Sub pattern the publisher and the subscribers are precisely decoupled thanks to the mentioned event bus or message broker.

The middleman or event bus

How do you achieve anonymity between publisher and subscriber? An easy way is let a middleman take care of all the communication. An event bus is one such middleman.

An event bus is typically composed of two parts:

- The abstraction or interface.
- One or more implementations.

In Figure 6-19 you can see how, from an application point of view, the event bus is nothing more than a Pub/Sub channel. The way you implement this asynchronous communication can vary. It can have multiple implementations so that you can swap between them, depending on the environment requirements (for example, production versus development environments).

In Figure 6-20, you can see an abstraction of an event bus with multiple implementations based on infrastructure messaging technologies like RabbitMQ, Azure Service Bus, or another event/message broker.

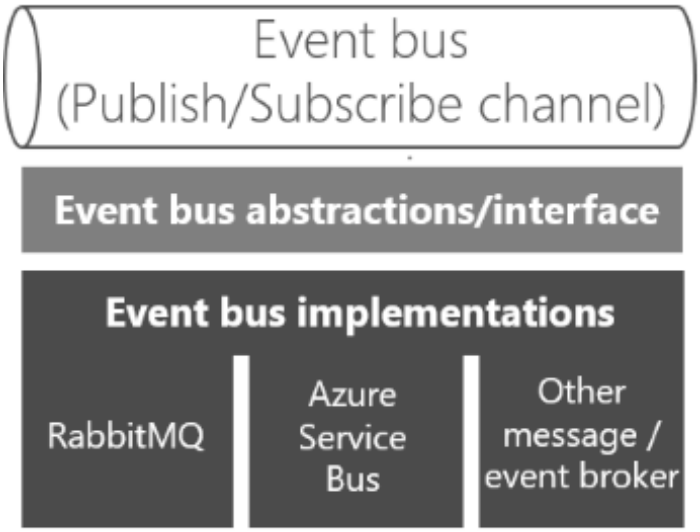


Figure 6- 20. Multiple implementations of an event bus

It's good to have the event bus defined through an interface so it can be implemented with several technologies, like RabbitMQ, Azure Service bus or others. However, and as mentioned previously, using your own abstractions (the event bus interface) is good only if you need basic event bus features supported by your abstractions. If you need richer service bus features, you should probably use the API and abstractions provided by your preferred commercial service bus instead of your own abstractions.

Defining an event bus interface

Let's start with some implementation code for the event bus interface and possible implementations for exploration purposes. The interface should be generic and straightforward, as in the following interface.

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

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

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

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

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

The Publish method is straightforward. The event bus will broadcast the integration event passed to it to any microservice, or even an external application, subscribed to that event. This method is used by the microservice that is publishing the event.

The Subscribe methods (you can have several implementations depending on the arguments) are used by the microservices that want to receive events. This method has two arguments. The first is the integration event to subscribe to (IntegrationEvent). The second argument is the integration event handler (or callback method), named IIntegrationEventHandler<T>, to be executed when the receiver microservice gets that integration event message.

Additional resources

Some production-ready messaging solutions:

- **Azure Service Bus**
  https://learn.microsoft.com/azure/service-bus-messaging/
- **NServiceBus**
  https://particular.net/nservicebus
- **MassTransit**
  https://masstransit-project.com/

Implementing an event bus with RabbitMQ for the development or test environment

We should start by saying that if you create your custom event bus based on RabbitMQ running in a container, as the eShopOnContainers application does, it should be used only for your development and test environments. Don't use it for your production environment, unless you are building it as a part of a production-ready service bus as described in the Additional resources section below. A simple custom event bus might be missing many production-ready critical features that a commercial service bus has.

One of the event bus custom implementations in eShopOnContainers is basically a library using the RabbitMQ API. (There's another implementation based on Azure Service Bus.)

The event bus implementation with RabbitMQ lets microservices subscribe to events, publish events, and receive events, as shown in Figure 6-21.

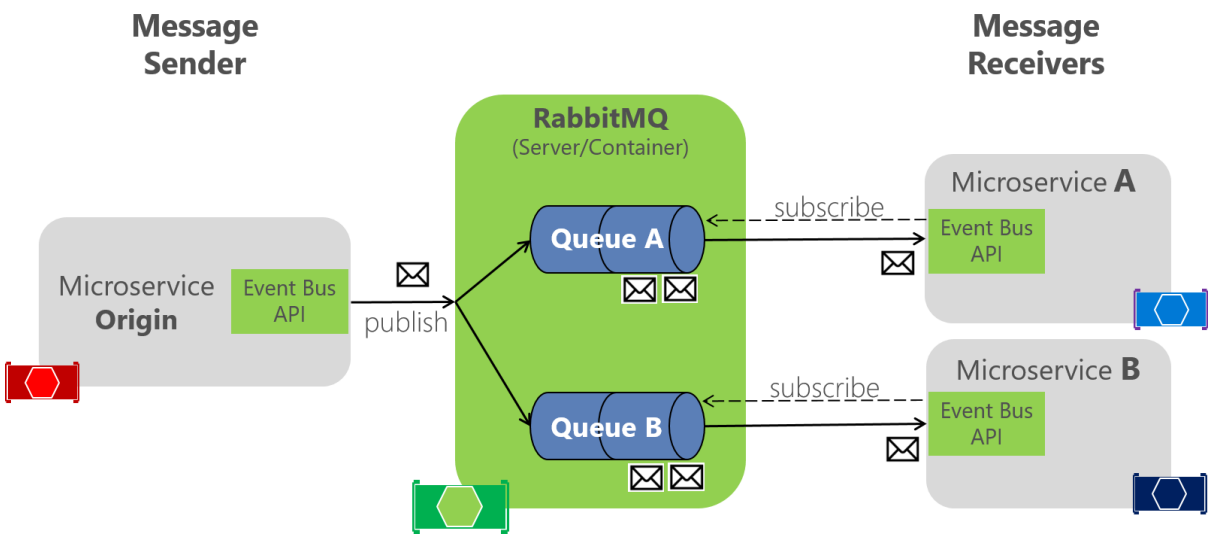


Figure 6-21. RabbitMQ implementation of an event bus

RabbitMQ functions as an intermediary between message publisher and subscribers, to handle distribution. In the code, the EventBusRabbitMQ class implements the generic IEventBus interface. This implementation is based on Dependency Injection so that you can swap from this dev/test version to a production version.

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

The RabbitMQ implementation of a sample dev/test event bus is boilerplate code. It has to handle the connection to the RabbitMQ server and provide code for publishing a message event to the queues. It also has to implement a dictionary of collections of integration event handlers for each event type; these event types can have a different instantiation and different subscriptions for each receiver microservice, as shown in Figure 6-21.

Implementing a simple publish method with RabbitMQ

The following code is a ***simplified*** version of an event bus implementation for RabbitMQ, to showcase the whole scenario. You don't really handle the connection this way. To see the full implementation, see the actual code in the dotnet-architecture/eShopOnContainers repository.

```
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);
        }
    }
}
```

The actual code of the Publish method in the eShopOnContainers application is improved by using a Polly retry policy, which retries the task some times in case the RabbitMQ container is not ready. This scenario can occur when docker-compose is starting the containers; for example, the RabbitMQ container might start more slowly than the other containers.

As mentioned earlier, there are many possible configurations in RabbitMQ, so this code should be used only for dev/test environments.

Implementing the subscription code with the RabbitMQ API

As with the publish code, the following code is a simplification of part of the event bus implementation for RabbitMQ. Again, you usually do not need to change it unless you are improving it.

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

    public void Subscribe<T, TH>()
        where T : IntegrationEvent
        where TH : IIntegrationEventHandler<T>
    {
        var eventName = _subsManager.GetEventKey<T>();

        var containsKey = _subsManager.HasSubscriptionsForEvent(eventName);
        if (!containsKey)
        {
            if (!_persistentConnection.IsConnected)
            {
                _persistentConnection.TryConnect();
            }

            using (var channel = _persistentConnection.CreateModel())
            {
                channel.QueueBind(queue: _queueName,
                                  exchange: BROKER_NAME,
                                  routingKey: eventName);
            }
        }

        _subsManager.AddSubscription<T, TH>();
    }
}
```

Each event type has a related channel to get events from RabbitMQ. You can then have as many event handlers per channel and event type as needed.

The Subscribe method accepts an IIntegrationEventHandler object, which is like a callback method in the current microservice, plus its related IntegrationEvent object. The code then adds that event handler to the list of event handlers that each integration event type can have per client microservice. If the client code has not already been subscribed to the event, the code creates a channel for the event type so it can receive events in a push style from RabbitMQ when that event is published from any other service.

As mentioned above, the event bus implemented in eShopOnContainers has only an educational purpose, since it only handles the main scenarios, so it's not ready for production.

For production scenarios check the additional resources below, specific for RabbitMQ, and the Implementing event-based communication between microservices section.

Additional resources

A production-ready solution with support for RabbitMQ.

- **NServiceBus** - Fully-supported commercial service bus with advanced management and monitoring tooling for .NET
  https://particular.net/
- **EasyNetQ** - Open Source .NET API client for RabbitMQ
  https://easynetq.com/
- **MassTransit** - Free, open-source distributed application framework for .NET
  https://masstransit-project.com/
- **Rebus** - Open source .NET Service Bus
  https://github.com/rebus-org/Rebus

Subscribing to events

The first step for using the event bus is to subscribe the microservices to the events they want to receive. That functionality should be done in the receiver microservices.

The following simple code shows what each receiver microservice needs to implement when starting the service (that is, in the Startup class) so it subscribes to the events it needs. In this case, the basket-api microservice needs to subscribe to ProductPriceChangedIntegrationEvent and the OrderStartedIntegrationEvent messages.

For instance, when subscribing to the ProductPriceChangedIntegrationEvent event, that makes the basket microservice aware of any changes to the product price and lets it warn the user about the change if that product is in the user's basket.

```
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();

eventBus.Subscribe<ProductPriceChangedIntegrationEvent,
```

```
                    ProductPriceChangedIntegrationEventHandler>();

eventBus.Subscribe<OrderStartedIntegrationEvent,
                    OrderStartedIntegrationEventHandler>();
```

After this code runs, the subscriber microservice will be listening through RabbitMQ channels. When any message of type ProductPriceChangedIntegrationEvent arrives, the code invokes the event handler that is passed to it and processes the event.

Publishing events through the event bus

Finally, the message sender (origin microservice) publishes the integration events with code similar to the following example. (This approach is a simplified example that does not take atomicity into account.) You would implement similar code whenever an event must be propagated across multiple microservices, usually right after committing data or transactions from the origin microservice.

First, the event bus implementation object (based on RabbitMQ or based on a service bus) would be injected at the controller constructor, as in the following code:

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

Then you use it from your controller's methods, like in the UpdateProduct method:

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

In this case, since the origin microservice is a simple CRUD microservice, that code is placed right into a Web API controller.

In more advanced microservices, like when using CQRS approaches, it can be implemented in the CommandHandler class, within the Handle() method.

Designing atomicity and resiliency when publishing to the event bus

When you publish integration events through a distributed messaging system like your event bus, you have the problem of atomically updating the original database and publishing an event (that is, either both operations complete or none of them). For instance, in the simplified example shown earlier, the code commits data to the database when the product price is changed and then publishes a ProductPriceChangedIntegrationEvent message. Initially, it might look essential that these two operations be performed atomically. However, if you are using a distributed transaction involving the database and the message broker, as you do in older systems like Microsoft Message Queuing (MSMQ), this approach is not recommended for the reasons described by the CAP theorem.

Basically, you use microservices to build scalable and highly available systems. Simplifying somewhat, the CAP theorem says that you cannot build a (distributed) database (or a microservice that owns its model) that's continually available, strongly consistent, *and* tolerant to any partition. You must choose two of these three properties.

In microservices-based architectures, you should choose availability and tolerance, and you should de-emphasize strong consistency. Therefore, in most modern microservice-based applications, you usually do not want to use distributed transactions in messaging, as you do when you implement distributed transactions based on the Windows Distributed Transaction Coordinator (DTC) with MSMQ.

Let's go back to the initial issue and its example. If the service crashes after the database is updated (in this case, right after the line of code with _context.SaveChangesAsync()), but before the integration event is published, the overall system could become inconsistent. This approach might be business critical, depending on the specific business operation you are dealing with.

As mentioned earlier in the architecture section, you can have several approaches for dealing with this issue:

- Using the full Event Sourcing pattern.
- Using transaction log mining.
- Using the Outbox pattern. This is a transactional table to store the integration events (extending the local transaction).

For this scenario, using the full Event Sourcing (ES) pattern is one of the best approaches, if not *the* best. However, in many application scenarios, you might not be able to implement a full ES system. ES means storing only domain events in your transactional database, instead of storing current state

data. Storing only domain events can have great benefits, such as having the history of your system available and being able to determine the state of your system at any moment in the past. However, implementing a full ES system requires you to rearchitect most of your system and introduces many other complexities and requirements. For example, you would want to use a database specifically made for event sourcing, such as Event Store, or a document-oriented database such as Azure Cosmos DB, MongoDB, Cassandra, CouchDB, or RavenDB. ES is a great approach for this problem, but not the easiest solution unless you are already familiar with event sourcing.

The option to use transaction log mining initially looks transparent. However, to use this approach, the microservice has to be coupled to your RDBMS transaction log, such as the SQL Server transaction log. This approach is probably not desirable. Another drawback is that the low-level updates recorded in the transaction log might not be at the same level as your high-level integration events. If so, the process of reverse-engineering those transaction log operations can be difficult.

A balanced approach is a mix of a transactional database table and a simplified ES pattern. You can use a state such as "ready to publish the event," which you set in the original event when you commit it to the integration events table. You then try to publish the event to the event bus. If the publish-event action succeeds, you start another transaction in the origin service and move the state from "ready to publish the event" to "event already published."

If the publish-event action in the event bus fails, the data still will not be inconsistent within the origin microservice—it is still marked as "ready to publish the event," and with respect to the rest of the services, it will eventually be consistent. You can always have background jobs checking the state of the transactions or integration events. If the job finds an event in the "ready to publish the event" state, it can try to republish that event to the event bus.

Notice that with this approach, you are persisting only the integration events for each origin microservice, and only the events that you want to communicate to other microservices or external systems. In contrast, in a full ES system, you store all domain events as well.

Therefore, this balanced approach is a simplified ES system. You need a list of integration events with their current state ("ready to publish" versus "published"). But you only need to implement these states for the integration events. And in this approach, you do not need to store all your domain data as events in the transactional database, as you would in a full ES system.

If you are already using a relational database, you can use a transactional table to store integration events. To achieve atomicity in your application, you use a two-step process based on local transactions. Basically, you have an IntegrationEvent table in the same database where you have your domain entities. That table works as an insurance for achieving atomicity so that you include persisted integration events into the same transactions that are committing your domain data.

Step by step, the process goes like this:

1. The application begins a local database transaction.
2. It then updates the state of your domain entities and inserts an event into the integration event table.
3. Finally, it commits the transaction, so you get the desired atomicity and then

4. You publish the event somehow (next).

When implementing the steps of publishing the events, you have these choices:

- Publish the integration event right after committing the transaction and use another local transaction to mark the events in the table as being published. Then, use the table just as an artifact to track the integration events in case of issues in the remote microservices, and perform compensatory actions based on the stored integration events.
- Use the table as a kind of queue. A separate application thread or process queries the integration event table, publishes the events to the event bus, and then uses a local transaction to mark the events as published.

Figure 6-22 shows the architecture for the first of these approaches.

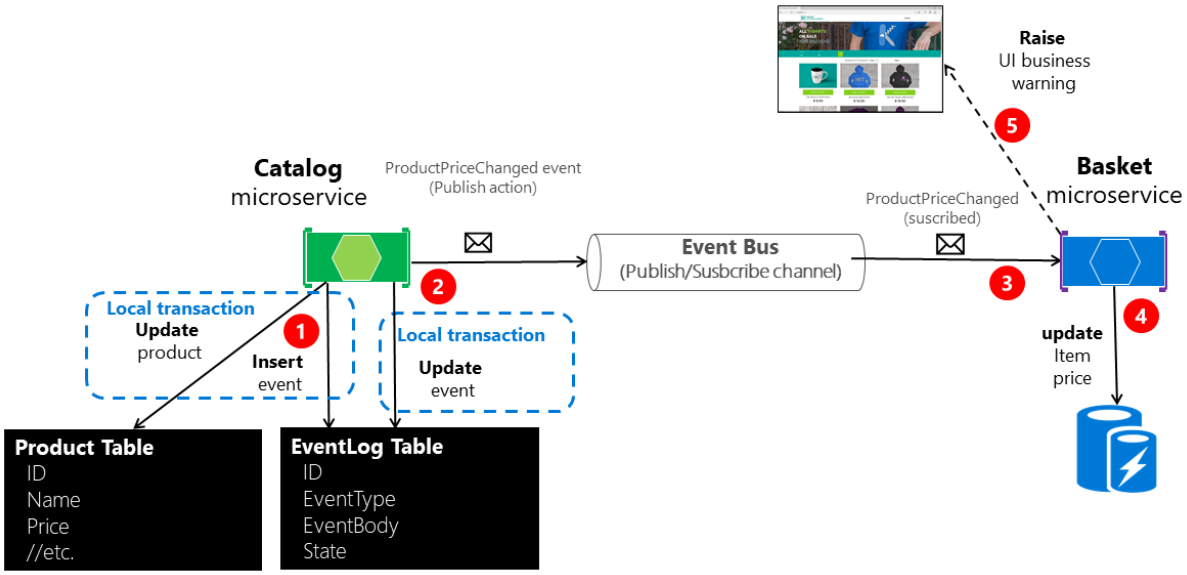


Figure 6-22. Atomicity when publishing events to the event bus

The approach illustrated in Figure 6-22 is missing an additional worker microservice that is in charge of checking and confirming the success of the published integration events. In case of failure, that additional checker worker microservice can read events from the table and republish them, that is, repeat step number 2.

About the second approach: you use the EventLog table as a queue and always use a worker microservice to publish the messages. In that case, the process is like that shown in Figure 6-23. This shows an additional microservice, and the table is the single source when publishing events.

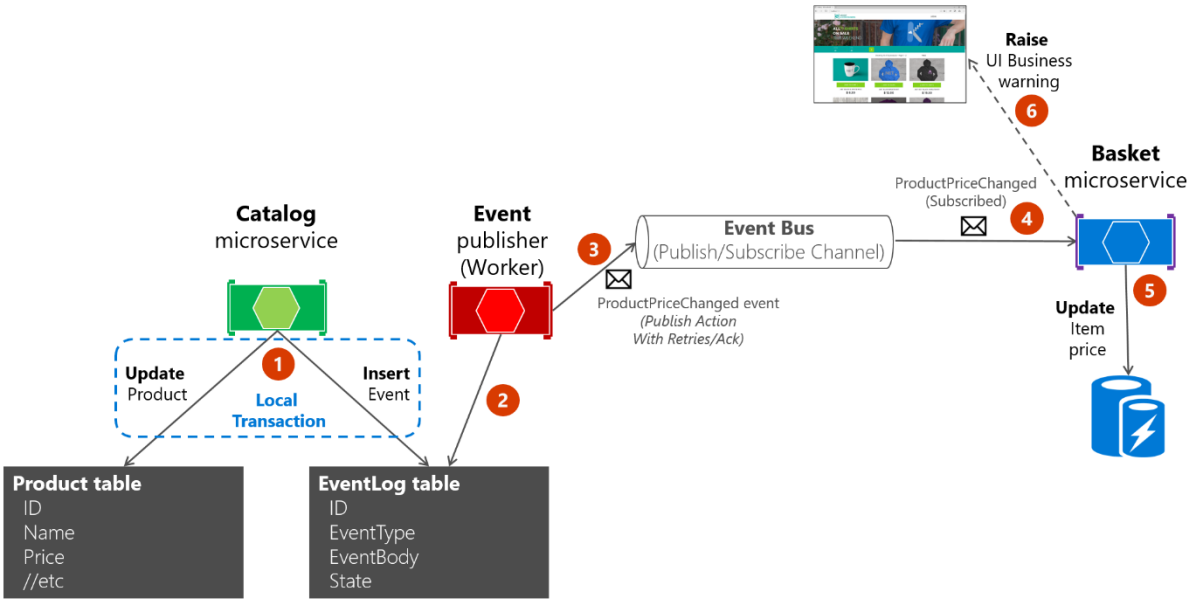


Figure 6-23. Atomicity when publishing events to the event bus with a worker microservice

For simplicity, the eShopOnContainers sample uses the first approach (with no additional processes or checker microservices) plus the event bus. However, the eShopOnContainers sample is not handling all possible failure cases. In a real application deployed to the cloud, you must embrace the fact that issues will arise eventually, and you must implement that check and resend logic. Using the table as a queue can be more effective than the first approach if you have that table as a single source of events when publishing them (with the worker) through the event bus.

Implementing atomicity when publishing integration events through the event bus

The following code shows how you can create a single transaction involving multiple DbContext objects—one context related to the original data being updated, and the second context related to the IntegrationEventLog table.

The transaction in the example code below will not be resilient if connections to the database have any issue at the time when the code is running. This can happen in cloud-based systems like Azure SQL DB, which might move databases across servers. For implementing resilient transactions across multiple contexts, see the Implementing resilient Entity Framework Core SQL connections section later in this guide.

For clarity, the following example shows the whole process in a single piece of code. However, the eShopOnContainers implementation is refactored and splits this logic into multiple classes so it's easier to maintain.

```
// 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();

           await _integrationEventLogService.SaveEventAsync(priceChangedEvent);

           transaction.Commit();
        }

      // Publish the integration event through the event bus
      _eventBus.Publish(priceChangedEvent);

      _integrationEventLogService.MarkEventAsPublishedAsync(
                                              priceChangedEvent);
  }

  return Ok();
}
```

After the ProductPriceChangedIntegrationEvent integration event is created, the transaction that stores the original domain operation (update the catalog item) also includes the persistence of the event in the EventLog table. This makes it a single transaction, and you will always be able to check whether event messages were sent.

The event log table is updated atomically with the original database operation, using a local transaction against the same database. If any of the operations fail, an exception is thrown and the transaction rolls back any completed operation, thus maintaining consistency between the domain operations and the event messages saved to the table.

Receiving messages from subscriptions: event handlers in receiver microservices

In addition to the event subscription logic, you need to implement the internal code for the integration event handlers (like a callback method). The event handler is where you specify where the event messages of a certain type will be received and processed.

An event handler first receives an event instance from the event bus. Then it locates the component to be processed related to that integration event, propagating and persisting the event as a change in state in the receiver microservice. For example, if a ProductPriceChanged event originates in the catalog microservice, it is handled in the basket microservice and changes the state in this receiver basket microservice as well, as shown in the following code.

```
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);
            }
        }
    }
}
```

The event handler needs to verify whether the product exists in any of the basket instances. It also updates the item price for each related basket line item. Finally, it creates an alert to be displayed to the user about the price change, as shown in Figure 6-24.

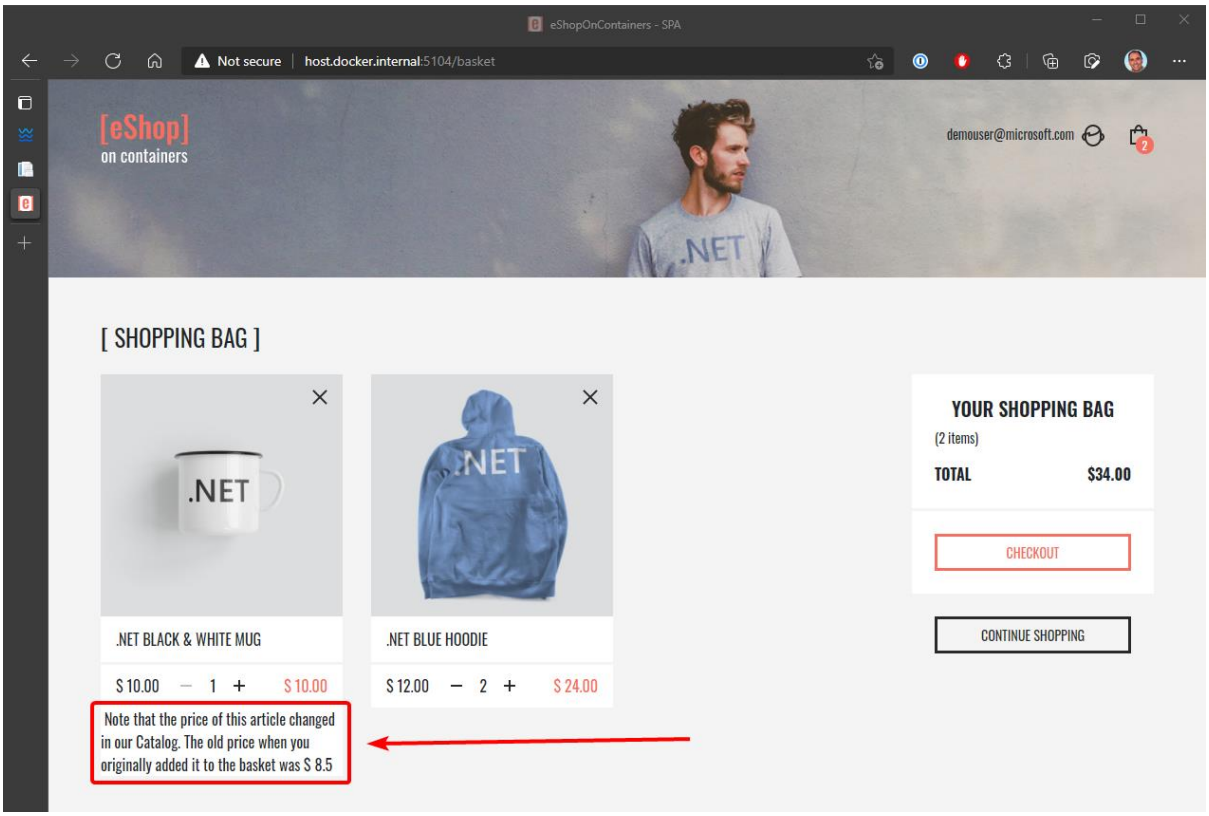


Figure 6-24. Displaying an item price change in a basket, as communicated by integration events

Idempotency in update message events

An important aspect of update message events is that a failure at any point in the communication should cause the message to be retried. Otherwise a background task might try to publish an event that has already been published, creating a race condition. Make sure that the updates are either idempotent or that they provide enough information to ensure that you can detect a duplicate, discard it, and send back only one response.

As noted earlier, idempotency means that an operation can be performed multiple times without changing the result. In a messaging environment, as when communicating events, an event is idempotent if it can be delivered multiple times without changing the result for the receiver microservice. This may be necessary because of the nature of the event itself, or because of the way the system handles the event. Message idempotency is important in any application that uses messaging, not just in applications that implement the event bus pattern.

An example of an idempotent operation is a SQL statement that inserts data into a table only if that data is not already in the table. It does not matter how many times you run that insert SQL statement; the result will be the same—the table will contain that data. Idempotency like this can also be necessary when dealing with messages if the messages could potentially be sent and therefore

processed more than once. For instance, if retry logic causes a sender to send exactly the same message more than once, you need to make sure that it is idempotent.

It is possible to design idempotent messages. For example, you can create an event that says "set the product price to $25" instead of "add $5 to the product price." You could safely process the first message any number of times and the result will be the same. That is not true for the second message. But even in the first case, you might not want to process the first event, because the system could also have sent a newer price-change event and you would be overwriting the new price.

Another example might be an order-completed event that's propagated to multiple subscribers. The app has to make sure that order information is updated in other systems only once, even if there are duplicated message events for the same order-completed event.

It is convenient to have some kind of identity per event so that you can create logic that enforces that each event is processed only once per receiver.

Some message processing is inherently idempotent. For example, if a system generates image thumbnails, it might not matter how many times the message about the generated thumbnail is processed; the outcome is that the thumbnails are generated and they are the same every time. On the other hand, operations such as calling a payment gateway to charge a credit card may not be idempotent at all. In these cases, you need to ensure that processing a message multiple times has the effect that you expect.

Additional resources

- **Honoring message idempotency**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591565(v=pandp.10)#honoring-message-idempotency

Deduplicating integration event messages

You can make sure that message events are sent and processed only once per subscriber at different levels. One way is to use a deduplication feature offered by the messaging infrastructure you are using. Another is to implement custom logic in your destination microservice. Having validations at both the transport level and the application level is your best bet.

Deduplicating message events at the EventHandler level

One way to make sure that an event is processed only once by any receiver is by implementing certain logic when processing the message events in event handlers. For example, that is the approach used in the eShopOnContainers application, as you can see in the source code of the UserCheckoutAcceptedIntegrationEventHandler class when it receives a UserCheckoutAcceptedIntegrationEvent integration event. (In this case, the CreateOrderCommand is wrapped with an IdentifiedCommand, using the eventMsg.RequestId as an identifier, before sending it to the command handler).

Deduplicating messages when using RabbitMQ

When intermittent network failures happen, messages can be duplicated, and the message receiver must be ready to handle these duplicated messages. If possible, receivers should handle messages in an idempotent way, which is better than explicitly handling them with deduplication.

According to the RabbitMQ documentation, "If a message is delivered to a consumer and then requeued (because it was not acknowledged before the consumer connection dropped, for example) then RabbitMQ will set the redelivered flag on it when it is delivered again (whether to the same consumer or a different one).

If the "redelivered" flag is set, the receiver must take that into account, because the message might already have been processed. But that is not guaranteed; the message might never have reached the receiver after it left the message broker, perhaps because of network issues. On the other hand, if the "redelivered" flag is not set, it is guaranteed that the message has not been sent more than once. Therefore, the receiver needs to deduplicate messages or process messages in an idempotent way only if the "redelivered" flag is set in the message.

Additional resources

- **Forked eShopOnContainers using NServiceBus (Particular Software)**
  https://go.particular.net/eShopOnContainers
- **Event Driven Messaging**
  https://patterns.arcitura.com/soa-patterns/design_patterns/event_driven_messaging
- **Jimmy Bogard. Refactoring Towards Resilience: Evaluating Coupling**
  https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/
- **Publish-Subscribe channel**
  https://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
- **Communicating Between Bounded Contexts**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591572(v=pandp.10)
- **Eventual Consistency**
  https://en.wikipedia.org/wiki/Eventual_consistency
- **Philip Brown. Strategies for Integrating Bounded Contexts**
  https://www.culttt.com/2014/11/26/strategies-integrating-bounded-contexts/
- **Chris Richardson. Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 2**
  https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson
- **Chris Richardson. Event Sourcing pattern**
  https://microservices.io/patterns/data/event-sourcing.html
- **Introducing Event Sourcing**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591559(v=pandp.10)

- **Event Store database**. Official site.
  https://geteventstore.com/
- **Patrick Nommensen. Event-Driven Data Management for Microservices**
  https://dzone.com/articles/event-driven-data-management-for-microservices-1
- **The CAP Theorem**
  https://en.wikipedia.org/wiki/CAP_theorem
- **What is CAP Theorem?**
  https://www.quora.com/What-Is-CAP-Theorem-1
- **Data Consistency Primer**
  https://learn.microsoft.com/previous-versions/msp-n-p/dn589800(v=pandp.10)
- **Rick Saling. The CAP Theorem: Why "Everything is Different" with the Cloud and Internet**
  https://learn.microsoft.com/archive/blogs/rickatmicrosoft/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/
- **Eric Brewer. CAP Twelve Years Later: How the "Rules" Have Changed**
  https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed
- **CAP, PACELC, and Microservices**
  https://ardalis.com/cap-pacelc-and-microservices/
- **Azure Service Bus. Brokered Messaging: Duplicate Detection**
  https://github.com/microsoftarchive/msdn-code-gallery-microsoft/tree/master/Windows%20Azure%20Product%20Team/Brokered%20Messaging%20Duplicate%20Detection
- **Reliability Guide** (RabbitMQ documentation)
  https://www.rabbitmq.com/reliability.html#consumer

Testing ASP.NET Core services and web apps

Controllers are a central part of any ASP.NET Core API service and ASP.NET MVC Web application. As such, you should have confidence they behave as intended for your application. Automated tests can provide you with this confidence and can detect errors before they reach production.

You need to test how the controller behaves based on valid or invalid inputs, and test controller responses based on the result of the business operation it performs. However, you should have these types of tests for your microservices:

- Unit tests. These tests ensure that individual components of the application work as expected. Assertions test the component API.
- Integration tests. These tests ensure that component interactions work as expected against external artifacts like databases. Assertions can test component API, UI, or the side effects of actions like database I/O, logging, etc.

- Functional tests for each microservice. These tests ensure that the application works as expected from the user's perspective.
- Service tests. These tests ensure that end-to-end service use cases, including testing multiple services at the same time, are tested. For this type of testing, you need to prepare the environment first. In this case, it means starting the services (for example, by using docker-compose up).

Implementing unit tests for ASP.NET Core Web APIs

Unit testing involves testing a part of an application in isolation from its infrastructure and dependencies. When you unit test controller logic, only the content of a single action or method is tested, not the behavior of its dependencies or of the framework itself. Unit tests do not detect issues in the interaction between components—that is the purpose of integration testing.

As you unit test your controller actions, make sure you focus only on their behavior. A controller unit test avoids things like filters, routing, or model binding (the mapping of request data to a ViewModel or DTO). Because they focus on testing just one thing, unit tests are generally simple to write and quick to run. A well-written set of unit tests can be run frequently without much overhead.

Unit tests are implemented based on test frameworks like xUnit.net, MSTest, Moq, or NUnit. For the eShopOnContainers sample application, we are using xUnit.

When you write a unit test for a Web API controller, you instantiate the controller class directly using the new keyword in C#, so that the test will run as fast as possible. The following example shows how to do this when using xUnit as the Test framework.

```
[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);
}
```

Implementing integration and functional tests for each microservice

As noted, integration tests and functional tests have different purposes and goals. However, the way you implement both when testing ASP.NET Core controllers is similar, so in this section we concentrate on integration tests.

Integration testing ensures that an application's components function correctly when assembled. ASP.NET Core supports integration testing using unit test frameworks and a built-in test web host that can be used to handle requests without network overhead.

Unlike unit testing, integration tests frequently involve application infrastructure concerns, such as a database, file system, network resources, or web requests and responses. Unit tests use fakes or mock objects in place of these concerns. But the purpose of integration tests is to confirm that the system works as expected with these systems, so for integration testing you do not use fakes or mock objects. Instead, you include the infrastructure, like database access or service invocation from other services.

Because integration tests exercise larger segments of code than unit tests, and because integration tests rely on infrastructure elements, they tend to be orders of magnitude slower than unit tests. Thus, it is a good idea to limit how many integration tests you write and run.

ASP.NET Core includes a built-in test web host that can be used to handle HTTP requests without network overhead, meaning that you can run those tests faster than when using a real web host. The test web host (TestServer) is available in a NuGet component as Microsoft.AspNetCore.TestHost. It can be added to integration test projects and used to host ASP.NET Core applications.

As you can see in the following code, when you create integration tests for ASP.NET Core controllers, you instantiate the controllers through the test host. This functionality is comparable to an HTTP request, but it runs faster.

```
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);
    }
}
```

Additional resources

- **Steve Smith. Testing controllers** (ASP.NET Core)
  https://learn.microsoft.com/aspnet/core/mvc/controllers/testing
- **Steve Smith. Integration testing** (ASP.NET Core)
  https://learn.microsoft.com/aspnet/core/test/integration-tests
- **Unit testing in .NET using dotnet test**
  https://learn.microsoft.com/dotnet/core/testing/unit-testing-with-dotnet-test
- **xUnit.net**. Official site.
  https://xunit.net/
- **Unit Test Basics.**
  https://learn.microsoft.com/visualstudio/test/unit-test-basics
- **Moq**. GitHub repo.
  https://github.com/moq/moq
- **NUnit**. Official site.
  https://nunit.org/

Implementing service tests on a multi-container application

As noted earlier, when you test multi-container applications, all the microservices need to be running within the Docker host or container cluster. End-to-end service tests that include multiple operations involving several microservices require you to deploy and start the whole application in the Docker host by running docker-compose up (or a comparable mechanism if you are using an orchestrator). Once the whole application and all its services is running, you can execute end-to-end integration and functional tests.

There are a few approaches you can use. In the docker-compose.yml file that you use to deploy the application at the solution level you can expand the entry point to use dotnet test. You can also use another compose file that would run your tests in the image you are targeting. By using another compose file for integration tests that includes your microservices and databases on containers, you can make sure that the related data is always reset to its original state before running the tests.

Once the compose application is up and running, you can take advantage of breakpoints and exceptions if you are running Visual Studio. Or you can run the integration tests automatically in your CI pipeline in Azure DevOps Services or any other CI/CD system that supports Docker containers.

Testing in eShopOnContainers

The reference application (eShopOnContainers) tests were recently restructured and now there are four categories:

1. **Unit** tests, just plain old regular unit tests, contained in the **{MicroserviceName}.UnitTests** projects

2. **Microservice functional/integration tests**, with test cases involving the infrastructure for each microservice but isolated from the others and are contained in the **{MicroserviceName}.FunctionalTests** projects.

3. **Application functional/integration tests**, which focus on microservices integration, with test cases that exert several microservices. These tests are located in project **Application.FunctionalTests**.

While unit and integration tests are organized in a test folder within the microservice project, application and load tests are managed separately under the root folder, as shown in Figure 6-25.

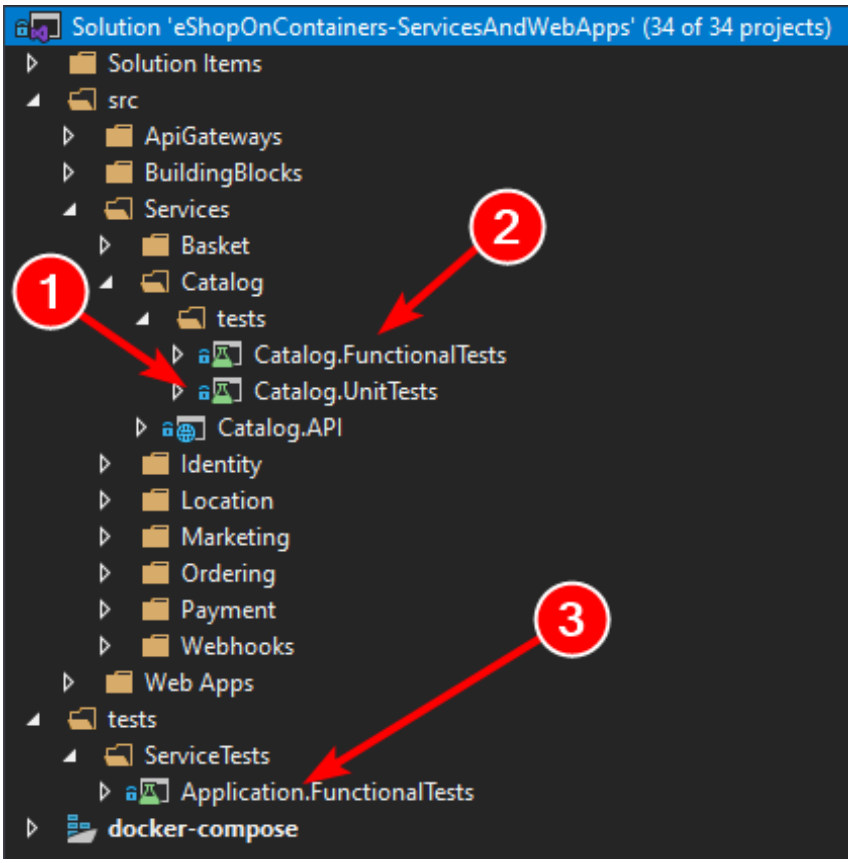


Figure 6-25. Test folder structure in eShopOnContainers

Microservice and Application functional/integration tests are run from Visual Studio, using the regular tests runner, but first you need to start the required infrastructure services, with a set of docker-compose files contained in the solution test folder:

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"
```

So, to run the functional/integration tests you must first run this command, from the solution test folder:

`docker-compose -f docker-compose-test.yml -f docker-compose-test.override.yml up`

As you can see, these docker-compose files only start the Redis, RabbitMQ, SQL Server, and MongoDB microservices.

Additional resources

- **Unit & Integration testing** on the eShopOnContainers https://github.com/dotnet-architecture/eShopOnContainers/wiki/Unit-and-integration-testing
- **Load testing** on the eShopOnContainers https://github.com/dotnet-architecture/eShopOnContainers/wiki/Load-testing

Implement background tasks in microservices with IHostedService and the BackgroundService class

Background tasks and scheduled jobs are something you might need to use in any application, whether or not it follows the microservices architecture pattern. The difference when using a microservices architecture is that you can implement the background task in a separate process/container for hosting so you can scale it down/up based on your need.

From a generic point of view, in .NET we called these type of tasks *Hosted Services*, because they are services/logic that you host within your host/application/microservice. Note that in this case, the hosted service simply means a class with the background task logic.

Since .NET Core 2.0, the framework provides a new interface named IHostedService helping you to easily implement hosted services. The basic idea is that you can register multiple background tasks (hosted services) that run in the background while your web host or host is running, as shown in the image 6-26.

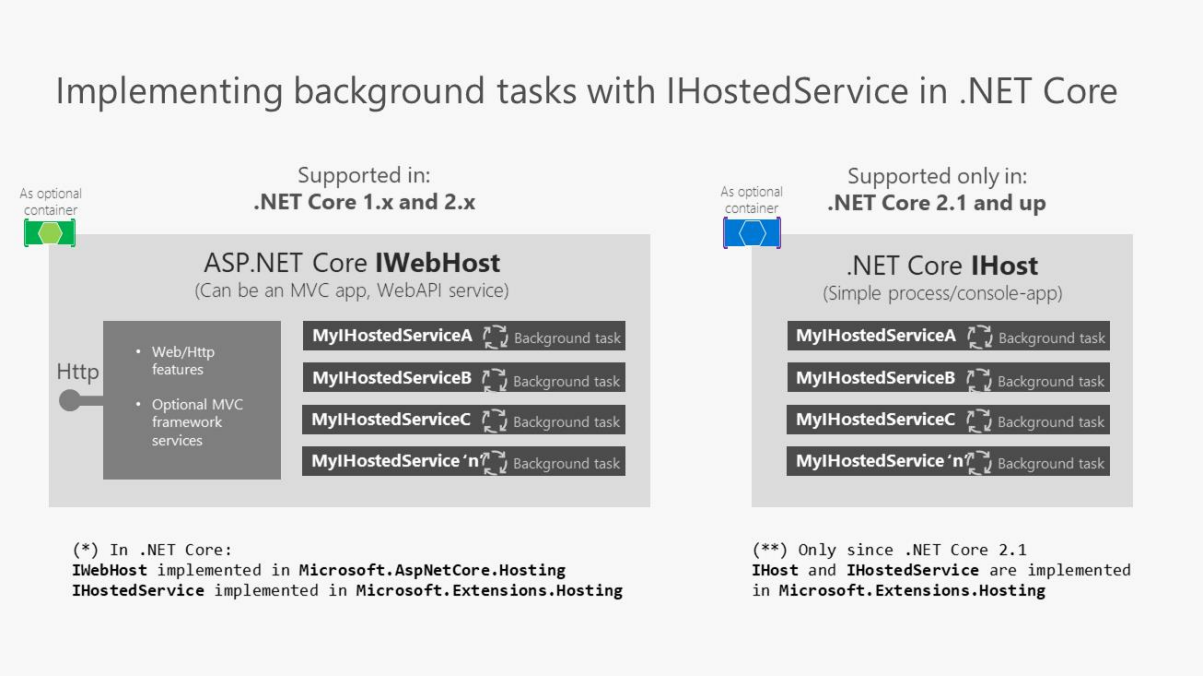


Figure 6-26. Using IHostedService in a WebHost vs. a Host

ASP.NET Core 1.x and 2.x support IWebHost for background processes in web apps. .NET Core 2.1 and later versions support IHost for background processes with plain console apps. Note the difference made between WebHost and Host.

A WebHost (base class implementing IWebHost) in ASP.NET Core 2.0 is the infrastructure artifact you use to provide HTTP server features to your process, such as when you're implementing an MVC web app or Web API service. It provides all the new infrastructure goodness in ASP.NET Core, enabling you to use dependency injection, insert middlewares in the request pipeline, and similar. The WebHost uses these very same IHostedServices for background tasks.

A Host (base class implementing IHost) was introduced in .NET Core 2.1. Basically, a Host allows you to have a similar infrastructure than what you have with WebHost (dependency injection, hosted services, etc.), but in this case, you just want to have a simple and lighter process as the host, with nothing related to MVC, Web API or HTTP server features.

Therefore, you can choose and either create a specialized host-process with IHost to handle the hosted services and nothing else, such a microservice made just for hosting the IHostedServices, or you can alternatively extend an existing ASP.NET Core WebHost, such as an existing ASP.NET Core Web API or MVC app.

Each approach has pros and cons depending on your business and scalability needs. The bottom line is basically that if your background tasks have nothing to do with HTTP (IWebHost) you should use IHost.

Registering hosted services in your WebHost or Host

Let's drill down further on the IHostedService interface since its usage is pretty similar in a WebHost or in a Host.

SignalR is one example of an artifact using hosted services, but you can also use it for much simpler things like:

- A background task polling a database looking for changes.
- A scheduled task updating some cache periodically.
- An implementation of QueueBackgroundWorkItem that allows a task to be executed on a background thread.
- Processing messages from a message queue in the background of a web app while sharing common services such as ILogger.
- A background task started with Task.Run().

You can basically offload any of those actions to a background task that implements IHostedService.

The way you add one or multiple IHostedServices into your WebHost or Host is by registering them up through the AddHostedService extension method in an ASP.NET Core WebHost (or in a Host in .NET Core 2.1 and above). Basically, you have to register the hosted services within application startup in *Program.cs*.

```
//Other DI registrations;

// Register Hosted Services
builder.Services.AddHostedService<GracePeriodManagerService>();
builder.Services.AddHostedService<MyHostedServiceB>();
builder.Services.AddHostedService<MyHostedServiceC>();
//...
```

In that code, the GracePeriodManagerService hosted service is real code from the Ordering business microservice in eShopOnContainers, while the other two are just two additional samples.

The IHostedService background task execution is coordinated with the lifetime of the application (host or microservice, for that matter). You register tasks when the application starts and you have the opportunity to do some graceful action or clean-up when the application is shutting down.

Without using IHostedService, you could always start a background thread to run any task. The difference is precisely at the app's shutdown time when that thread would simply be killed without having the opportunity to run graceful clean-up actions.

The IHostedService interface

When you register an IHostedService, .NET calls the StartAsync() and StopAsync() methods of your IHostedService type during application start and stop respectively. For more details, see IHostedService interface.

As you can imagine, you can create multiple implementations of IHostedService and register each of them in *Program.cs*, as shown previously. All those hosted services will be started and stopped along with the application/microservice.

As a developer, you are responsible for handling the stopping action of your services when StopAsync() method is triggered by the host.

Implementing IHostedService with a custom hosted service class deriving from the BackgroundService base class

You could go ahead and create your custom hosted service class from scratch and implement the IHostedService, as you need to do when using .NET Core 2.0 and later.

However, since most background tasks will have similar needs in regard to the cancellation tokens management and other typical operations, there is a convenient abstract base class you can derive from, named BackgroundService (available since .NET Core 2.1).

That class provides the main work needed to set up the background task.

The next code is the abstract BackgroundService base class as implemented in .NET.

```
// 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();
    }
}
```

When deriving from the previous abstract base class, thanks to that inherited implementation, you just need to implement the ExecuteAsync() method in your own custom hosted service class, as in the following simplified code from eShopOnContainers which is polling a database and publishing integration events into the Event Bus when needed.

```
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();

            try {
                    await Task.Delay(_settings.CheckUpdateTime, stoppingToken);
                }
            catch (TaskCanceledException exception) {
                    _logger.LogCritical(exception, "TaskCanceledException Error",
exception.Message);
```

```
                }
            }

            _logger.LogDebug($"GracePeriod background task is stopping.");
        }

        .../...
}
```

In this specific case for eShopOnContainers, it's executing an application method that's querying a database table looking for orders with a specific state and when applying changes, it is publishing integration events through the event bus (underneath it can be using RabbitMQ or Azure Service Bus).

Of course, you could run any other business background task, instead.

By default, the cancellation token is set with a 5 seconds timeout, although you can change that value when building your WebHost using the UseShutdownTimeout extension of the IWebHostBuilder. This means that our service is expected to cancel within 5 seconds otherwise it will be more abruptly killed.

The following code would be changing that time to 10 seconds.

```
WebHost.CreateDefaultBuilder(args)
    .UseShutdownTimeout(TimeSpan.FromSeconds(10))
    ...
```

Summary class diagram

The following image shows a visual summary of the classes and interfaces involved when implementing IHostedServices.

Class diagram with a custom IHostedService and related classes and interfaces

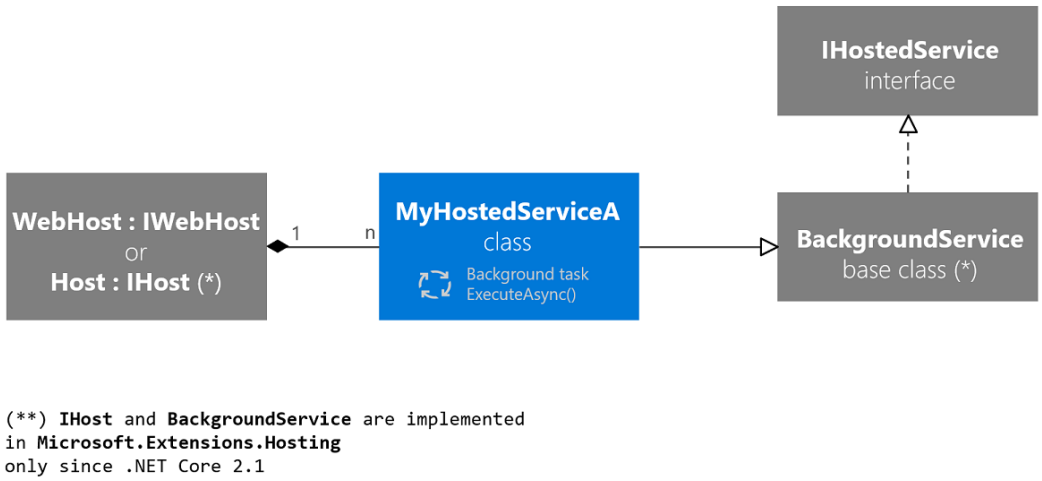


Figure 6-27. Class diagram showing the multiple classes and interfaces related to IHostedService

Class diagram: IWebHost and IHost can host many services, which inherit from BackgroundService, which implements IHostedService.

Deployment considerations and takeaways

It is important to note that the way you deploy your ASP.NET Core WebHost or .NET Host might impact the final solution. For instance, if you deploy your WebHost on IIS or a regular Azure App Service, your host can be shut down because of app pool recycles. But if you are deploying your host as a container into an orchestrator like Kubernetes, you can control the assured number of live instances of your host. In addition, you could consider other approaches in the cloud especially made for these scenarios, like Azure Functions. Finally, if you need the service to be running all the time and are deploying on a Windows Server you could use a Windows Service.

But even for a WebHost deployed into an app pool, there are scenarios like repopulating or flushing application's in-memory cache that would be still applicable.

The IHostedService interface provides a convenient way to start background tasks in an ASP.NET Core web application (in .NET Core 2.0 and later versions) or in any process/host (starting in .NET Core 2.1 with IHost). Its main benefit is the opportunity you get with the graceful cancellation to clean-up the code of your background tasks when the host itself is shutting down.

Additional resources

- **Building a scheduled task in ASP.NET Core/Standard 2.0**
  https://blog.maartenballiauw.be/post/2017/08/01/building-a-scheduled-cache-updater-in-aspnet-core-2.html
- **Implementing IHostedService in ASP.NET Core 2.0**
  https://www.stevejgordon.co.uk/asp-net-core-2-ihostedservice
- **GenericHost Sample using ASP.NET Core 2.1**
  https://github.com/aspnet/Hosting/tree/release/2.1/samples/GenericHostSample

Implement API Gateways with Ocelot

Important

The reference microservice application eShopOnContainers is currently using features provided by Envoy to implement the API Gateway instead of the earlier referenced Ocelot. We made this design choice because of Envoy's built-in support for the WebSocket protocol, required by the new gRPC inter-service communications implemented in eShopOnContainers. However, we've retained this section in the guide so you can consider Ocelot as a simple, capable, and lightweight API Gateway suitable for production-grade scenarios. Also, latest Ocelot version contains a breaking change on its json schema. Consider using Ocelot < v16.0.0, or use the key Routes instead of ReRoutes.

Architect and design your API Gateways

The following architecture diagram shows how API Gateways were implemented with Ocelot in eShopOnContainers.

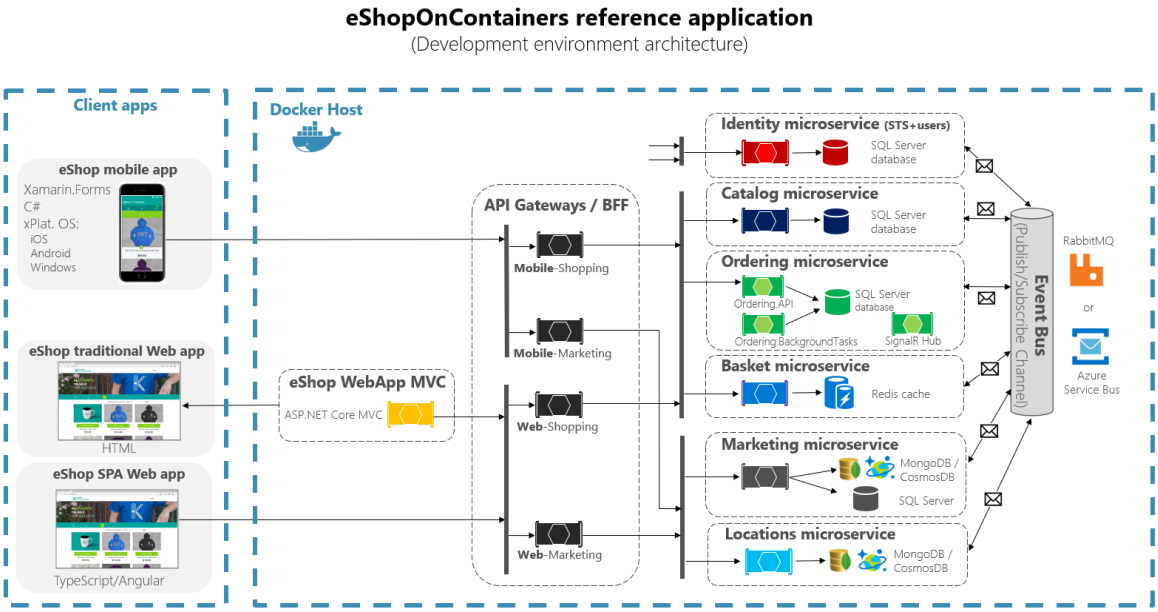


Figure 6-28. eShopOnContainers architecture with API Gateways

That diagram shows how the whole application is deployed into a single Docker host or development PC with "Docker for Windows" or "Docker for Mac". However, deploying into any orchestrator would be similar, but any container in the diagram could be scaled out in the orchestrator.

In addition, the infrastructure assets such as databases, cache, and message brokers should be offloaded from the orchestrator and deployed into high available systems for infrastructure, like Azure SQL Database, Azure Cosmos DB, Azure Redis, Azure Service Bus, or any HA clustering solution on-premises.

As you can also notice in the diagram, having several API Gateways allows multiple development teams to be autonomous (in this case Marketing features vs. Shopping features) when developing and deploying their microservices plus their own related API Gateways.

If you had a single monolithic API Gateway that would mean a single point to be updated by several development teams, which could couple all the microservices with a single part of the application.

Going much further in the design, sometimes a fine-grained API Gateway can also be limited to a single business microservice depending on the chosen architecture. Having the API Gateway's boundaries dictated by the business or domain will help you to get a better design.

For instance, fine granularity in the API Gateway tier can be especially useful for more advanced composite UI applications that are based on microservices, because the concept of a fine-grained API Gateway is similar to a UI composition service.

We delve into more details in the previous section Creating composite UI based on microservices.

As a key takeaway, for many medium- and large-size applications, using a custom-built API Gateway product is usually a good approach, but not as a single monolithic aggregator or unique central custom API Gateway unless that API Gateway allows multiple independent configuration areas for the several development teams creating autonomous microservices.

Sample microservices/containers to reroute through the API Gateways

As an example, eShopOnContainers has around six internal microservice-types that have to be published through the API Gateways, as shown in the following image.

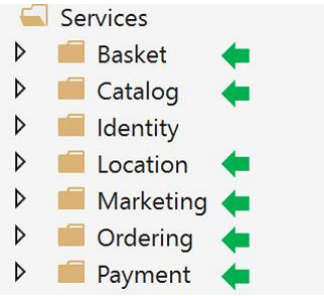


Figure 6-29. Microservice folders in eShopOnContainers solution in Visual Studio

About the Identity service, in the design it's left out of the API Gateway routing because it's the only cross-cutting concern in the system, although with Ocelot it's also possible to include it as part of the rerouting lists.

All those services are currently implemented as ASP.NET Core Web API services, as you can tell from the code. Let's focus on one of the microservices like the Catalog microservice code.

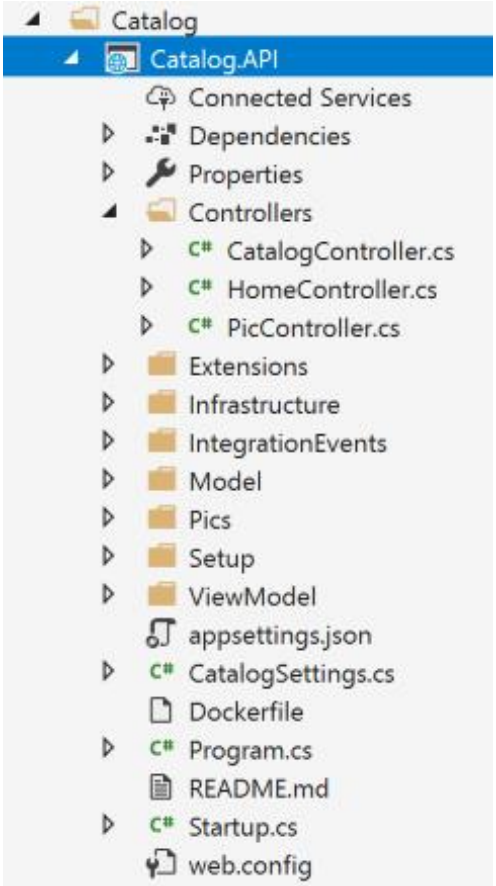


Figure 6-30. Sample Web API microservice (Catalog microservice)

You can see that the Catalog microservice is a typical ASP.NET Core Web API project with several controllers and methods like in the following code.

```
[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();
}
```

The HTTP request will end up running that kind of C# code accessing the microservice database and any additional required action.

Regarding the microservice URL, when the containers are deployed in your local development PC (local Docker host), each microservice's container always has an internal port (usually port 80) specified in its dockerfile, as in the following dockerfile:

```
FROM mcr.microsoft.com/dotnet/aspnet:7.0 AS base
WORKDIR /app
EXPOSE 80
```

The port 80 shown in the code is internal within the Docker host, so it can't be reached by client apps.

Client apps can access only the external ports (if any) published when deploying with docker-compose.

Those external ports shouldn't be published when deploying to a production environment. For this specific reason, why you want to use the API Gateway, to avoid the direct communication between the client apps and the microservices.

However, when developing, you want to access the microservice/container directly and run it through Swagger. That's why in eShopOnContainers, the external ports are still specified even when they won't be used by the API Gateway or the client apps.

Here's an example of the docker-compose.override.yml file for the Catalog microservice:

```
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).
```

You can see how in the docker-compose.override.yml configuration the internal port for the Catalog container is port 80, but the port for external access is 5101. But this port shouldn't be used by the application when using an API Gateway, only to debug, run, and test just the Catalog microservice.

Normally, you won't be deploying with docker-compose into a production environment because the right production deployment environment for microservices is an orchestrator like Kubernetes or Service Fabric. When deploying to those environments you use different configuration files where you won't publish directly any external port for the microservices but, you'll always use the reverse proxy from the API Gateway.

Run the catalog microservice in your local Docker host. Either run the full eShopOnContainers solution from Visual Studio (it runs all the services in the docker-compose files), or start the Catalog microservice with the following docker-compose command in CMD or PowerShell positioned at the folder where the docker-compose.yml and docker-compose.override.yml are placed.

```
docker-compose run --service-ports catalog-api
```

This command only runs the catalog-api service container plus dependencies that are specified in the docker-compose.yml. In this case, the SQL Server container and RabbitMQ container.

Then, you can directly access the Catalog microservice and see its methods through the Swagger UI accessing directly through that "external" port, in this case http://host.docker.internal:5101/swagger:

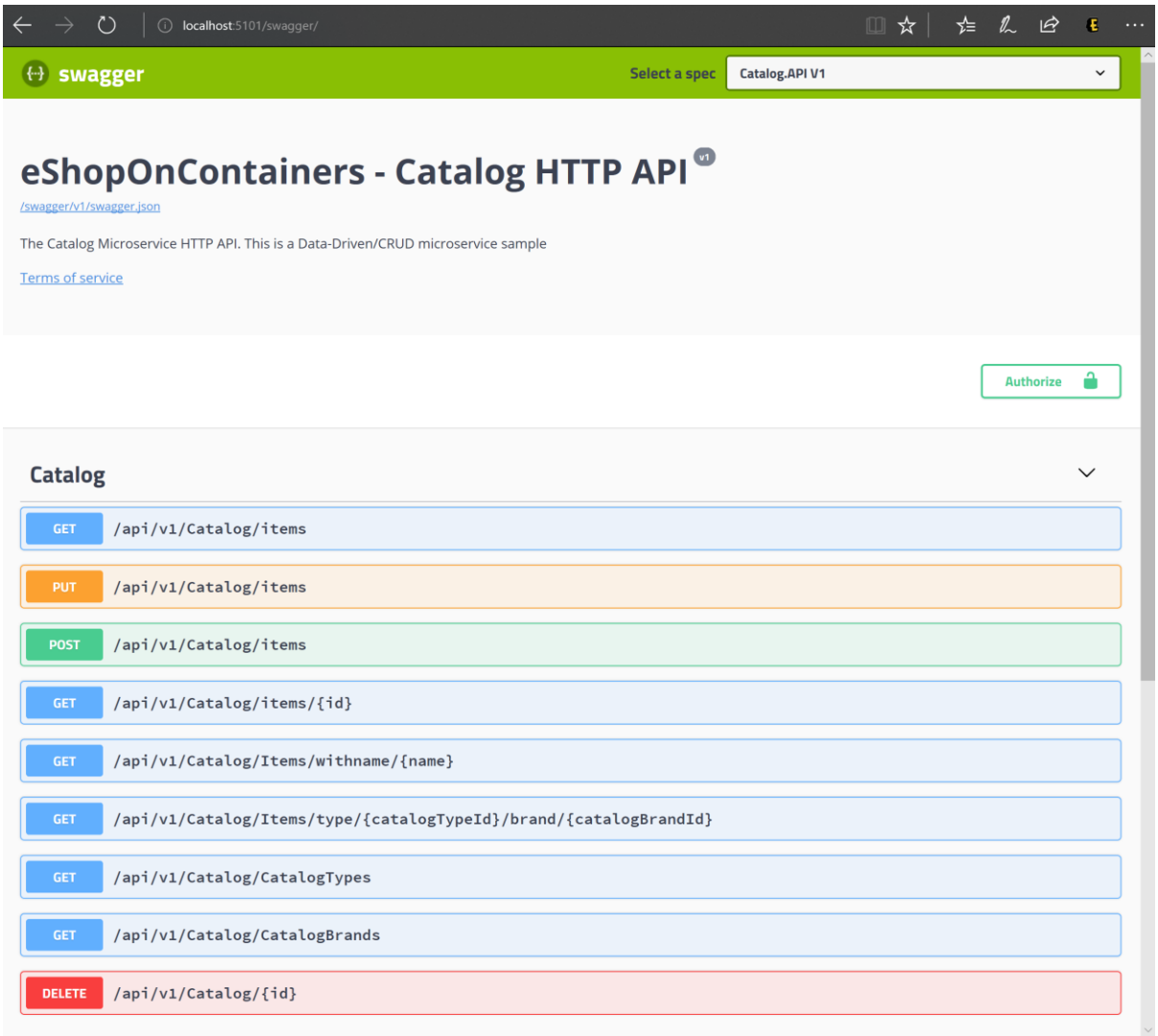


Figure 6-31. Testing the Catalog microservice with its Swagger UI

At this point, you could set a breakpoint in C# code in Visual Studio, test the microservice with the methods exposed in Swagger UI, and finally clean-up everything with the docker-compose down command.

However, direct-access communication to the microservice, in this case through the external port 5101, is precisely what you want to avoid in your application. And you can avoid that by setting the additional level of indirection of the API Gateway (Ocelot, in this case). That way, the client app won't directly access the microservice.

Implementing your API Gateways with Ocelot

Ocelot is basically a set of middleware that you can apply in a specific order.

Ocelot is designed to work with ASP.NET Core only. The latest version of the package is 18.0 which targets .NET 6 and hence is not suitable for .NET Framework applications.

You install Ocelot and its dependencies in your ASP.NET Core project with Ocelot's NuGet package, from Visual Studio.

```
Install-Package Ocelot
```

In eShopOnContainers, its API Gateway implementation is a simple ASP.NET Core WebHost project, and Ocelot's middleware handles all the API Gateway features, as shown in the following image:

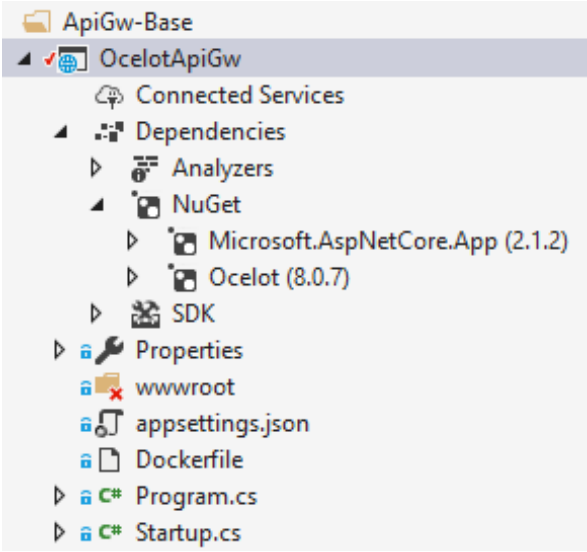


Figure 6-32. The OcelotApiGw base project in eShopOnContainers

This ASP.NET Core WebHost project is built with two simple files: Program.cs and Startup.cs.

The Program.cs just needs to create and configure the typical ASP.NET Core BuildWebHost.

```
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;
        }
    }
}
```

The important point here for Ocelot is the configuration.json file that you must provide to the builder through the AddJsonFile() method. That configuration.json is where you specify all the API Gateway ReRoutes, meaning the external endpoints with specific ports and the correlated internal endpoints, usually using different ports.

```
{
    "ReRoutes": [],
    "GlobalConfiguration": {}
}
```

There are two sections to the configuration. An array of ReRoutes and a GlobalConfiguration. The ReRoutes are the objects that tell Ocelot how to treat an upstream request. The Global configuration allows overrides of ReRoute specific settings. It's useful if you don't want to manage lots of ReRoute specific settings.

Here's a simplified example of [ReRoute configuration file] from one of the API Gateways from 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"
    }
  }
```

The main functionality of an Ocelot API Gateway is to take incoming HTTP requests and forward them on to a downstream service, currently as another HTTP request. Ocelot's describes the routing of one request to another as a ReRoute.

For instance, let's focus on one of the ReRoutes in the configuration.json from above, the configuration for the Basket microservice.

```
{
      "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": []
      }
}
```

The DownstreamPathTemplate, Scheme, and DownstreamHostAndPorts make the internal microservice URL that this request will be forwarded to.

The port is the internal port used by the service. When using containers, the port specified at its dockerfile.

The Host is a service name that depends on the service name resolution you are using. When using docker-compose, the services names are provided by the Docker Host, which is using the service names provided in the docker-compose files. If using an orchestrator like Kubernetes or Service Fabric, that name should be resolved by the DNS or name resolution provided by each orchestrator.

DownstreamHostAndPorts is an array that contains the host and port of any downstream services that you wish to forward requests to. Usually this configuration will just contain one entry but sometimes you might want to load balance requests to your downstream services and Ocelot lets you add more than one entry and then select a load balancer. But if using Azure and any orchestrator it is probably a better idea to load balance with the cloud and orchestrator infrastructure.

The UpstreamPathTemplate is the URL that Ocelot will use to identify which DownstreamPathTemplate to use for a given request from the client. Finally, the UpstreamHttpMethod is used so Ocelot can distinguish between different requests (GET, POST, PUT) to the same URL.

At this point, you could have a single Ocelot API Gateway (ASP.NET Core WebHost) using one or multiple merged configuration.json files or you can also store the configuration in a Consul KV store.

But as introduced in the architecture and design sections, if you really want to have autonomous microservices, it might be better to split that single monolithic API Gateway into multiple API Gateways and/or BFF (Backend for Frontend). For that purpose, let's see how to implement that approach with Docker containers.

Using a single Docker container image to run multiple different API Gateway / BFF container types

In eShopOnContainers, we're using a single Docker container image with the Ocelot API Gateway but then, at run time, we create different services/containers for each type of API-Gateway/BFF by providing a different configuration.json file, using a docker volume to access a different PC folder for each service.

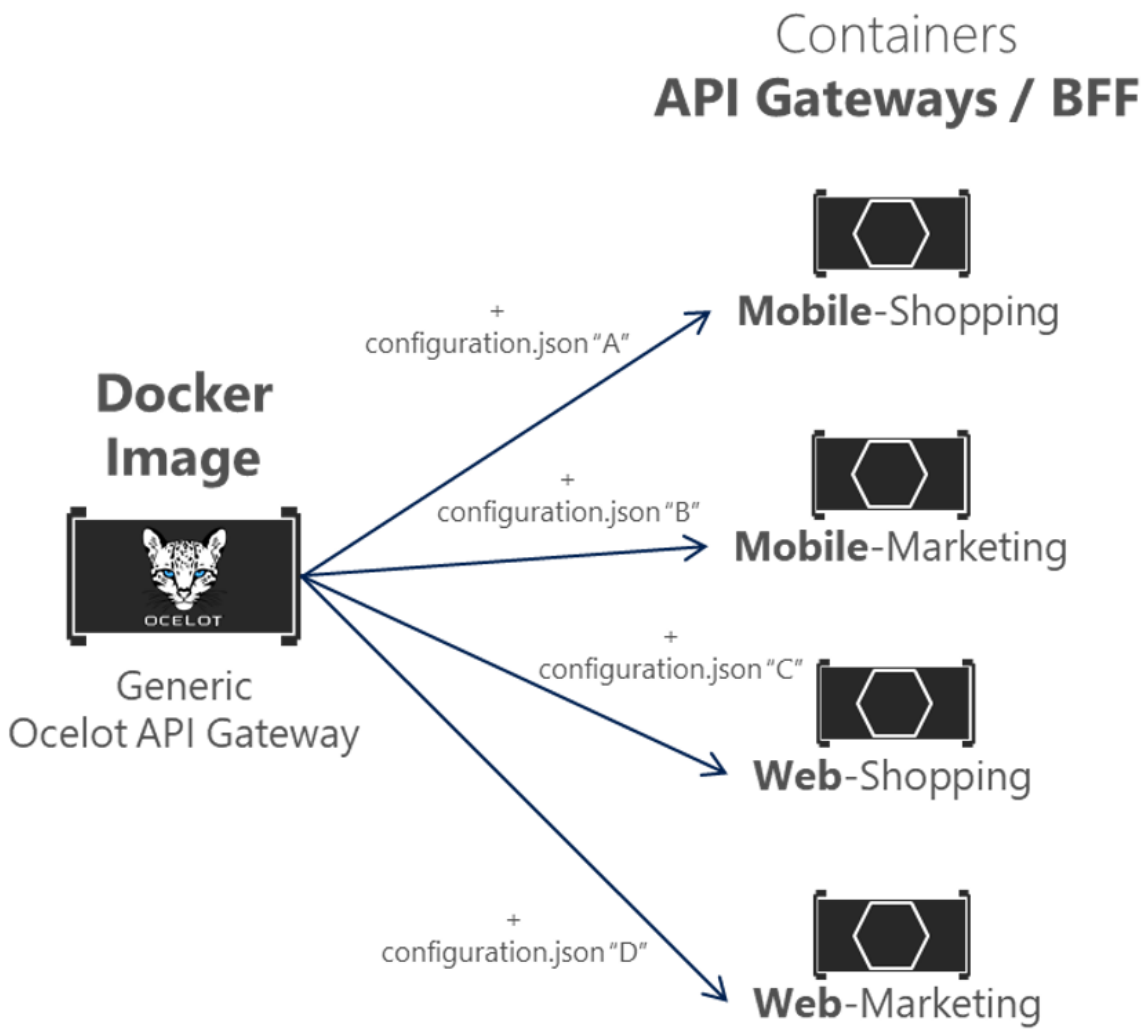


Figure 6-33. Reusing a single Ocelot Docker image across multiple API Gateway types

In eShopOnContainers, the "Generic Ocelot API Gateway Docker Image" is created with the project named 'OcelotApiGw' and the image name "eshop/ocelotapigw" that is specified in the docker-compose.yml file. Then, when deploying to Docker, there will be four API-Gateway containers created from that same Docker image, as shown in the following extract from the docker-compose.yml file.

```
  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
```

Additionally, as you can see in the following docker-compose.override.yml file, the only difference between those API Gateway containers is the Ocelot configuration file, which is different for each service container and it's specified at run time through a Docker volume.

```
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.Marketing/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.Marketing/apigw:/app/configuration
```

Because of that previous code, and as shown in the Visual Studio Explorer below, the only file needed to define each specific business/BFF API Gateway is just a configuration.json file, because the four API Gateways are based on the same Docker image.

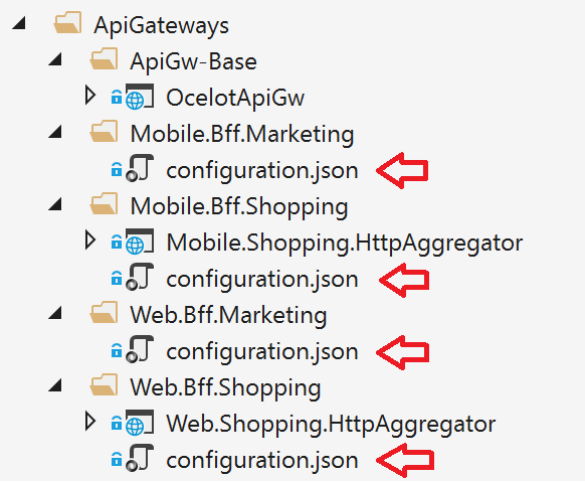


Figure 6-34. The only file needed to define each API Gateway / BFF with Ocelot is a configuration file

By splitting the API Gateway into multiple API Gateways, different development teams focusing on different subsets of microservices can manage their own API Gateways by using independent Ocelot configuration files. Plus, at the same time they can reuse the same Ocelot Docker image.

Now, if you run eShopOnContainers with the API Gateways (included by default in VS when opening eShopOnContainers-ServicesAndWebApps.sln solution or if running "docker-compose up"), the following sample routes will be performed.

For instance, when visiting the upstream URL http://host.docker.internal:5202/api/v1/c/catalog/items/2/ served by the webshoppingapigw API Gateway, you get the same result from the internal Downstream URL http://catalog-api/api/v1/2 within the Docker host, as in the following browser.

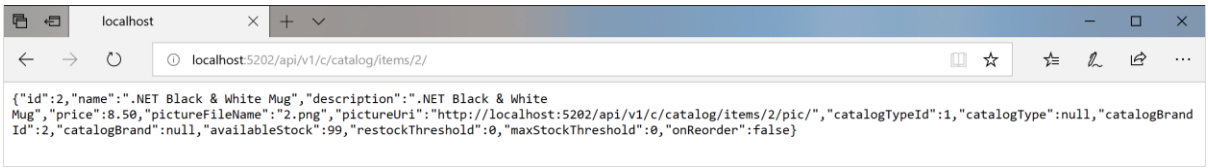


Figure 6-35. Accessing a microservice through a URL provided by the API Gateway

Because of testing or debugging reasons, if you wanted to directly access to the Catalog Docker container (only at the development environment) without passing through the API Gateway, since 'catalog-api' is a DNS resolution internal to the Docker host (service discovery handled by docker-compose service names), the only way to directly access the container is through the external port published in the docker-compose.override.yml, which is provided only for development tests, such as http://host.docker.internal:5101/api/v1/Catalog/items/1 in the following browser.

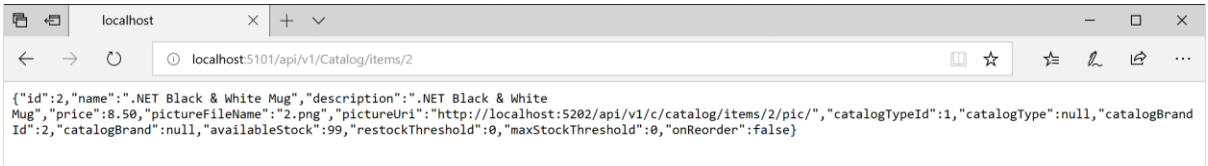


Figure 6-36. Direct access to a microservice for testing purposes

But the application is configured so it accesses all the microservices through the API Gateways, not through the direct port "shortcuts".

The Gateway aggregation pattern in eShopOnContainers

As introduced previously, a flexible way to implement requests aggregation is with custom services, by code. You could also implement request aggregation with the Request Aggregation feature in Ocelot, but it might not be as flexible as you need. Therefore, the selected way to implement aggregation in eShopOnContainers is with an explicit ASP.NET Core Web API service for each aggregator.

According to that approach, the API Gateway composition diagram is in reality a bit more extended when considering the aggregator services that are not shown in the simplified global architecture diagram shown previously.

In the following diagram, you can also see how the aggregator services work with their related API Gateways.

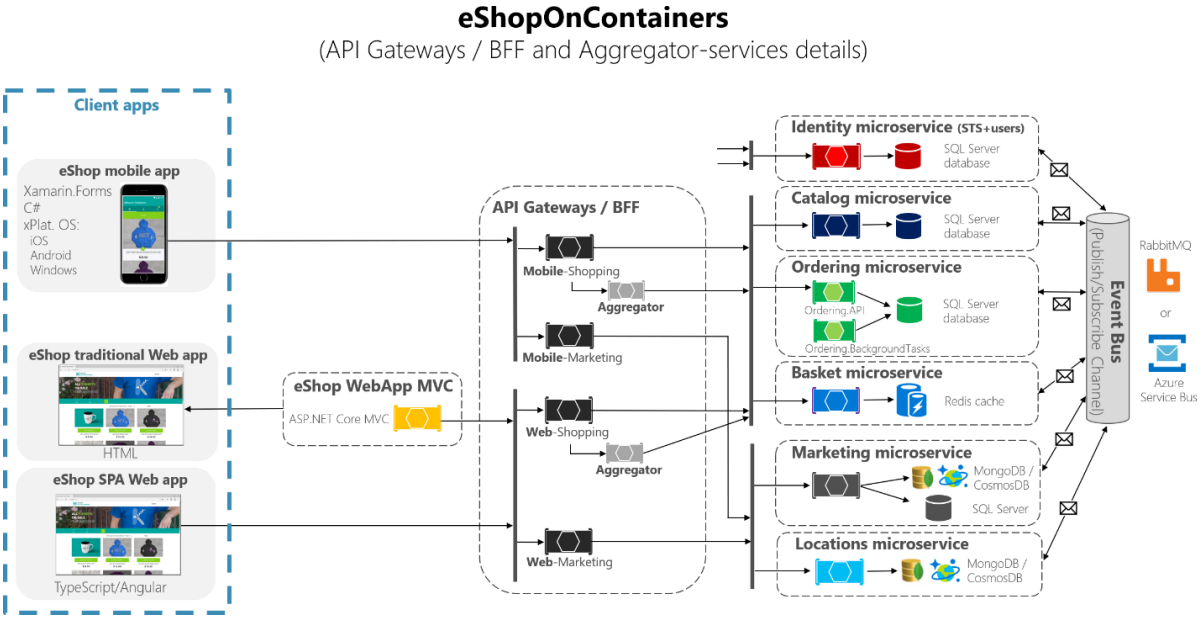


Figure 6-37. eShopOnContainers architecture with aggregator services

Zooming in further, on the "Shopping" business area in the following image, you can see that chattiness between the client apps and the microservices is reduced when using the aggregator services in the API Gateways.

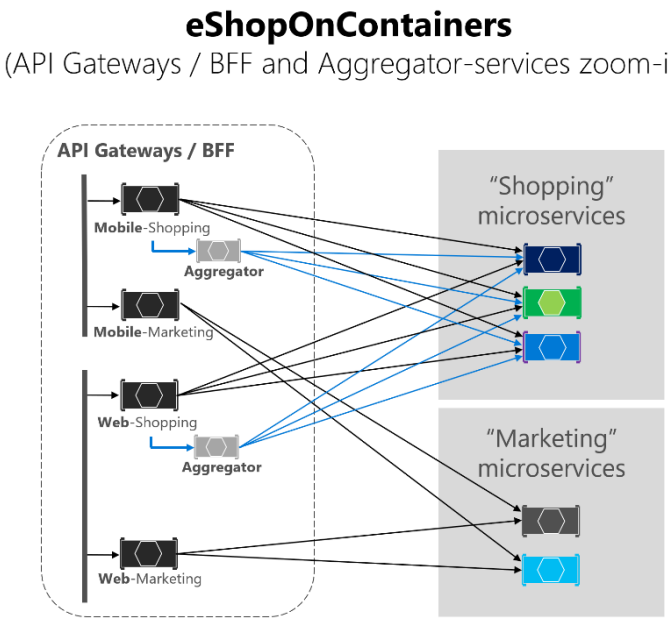


Figure 6-38. Zoom in vision of the Aggregator services

You can notice how when the diagram shows the possible requests coming from the API Gateways it can get complex. On the other hand, when you use the aggregator pattern, you can see how the arrows in blue would simplify the communication from a client app perspective. This pattern not only helps to reduce the chattiness and latency in the communication, it also improves the user experience significantly for the remote apps (mobile and SPA apps).

In the case of the "Marketing" business area and microservices, it is a simple use case so there was no need to use aggregators, but it could also be possible, if needed.

Authentication and authorization in Ocelot API Gateways

In an Ocelot API Gateway, you can sit the authentication service, such as an ASP.NET Core Web API service using IdentityServer providing the auth token, either out or inside the API Gateway.

Since eShopOnContainers is using multiple API Gateways with boundaries based on BFF and business areas, the Identity/Auth service is left out of the API Gateways, as highlighted in yellow in the following diagram.

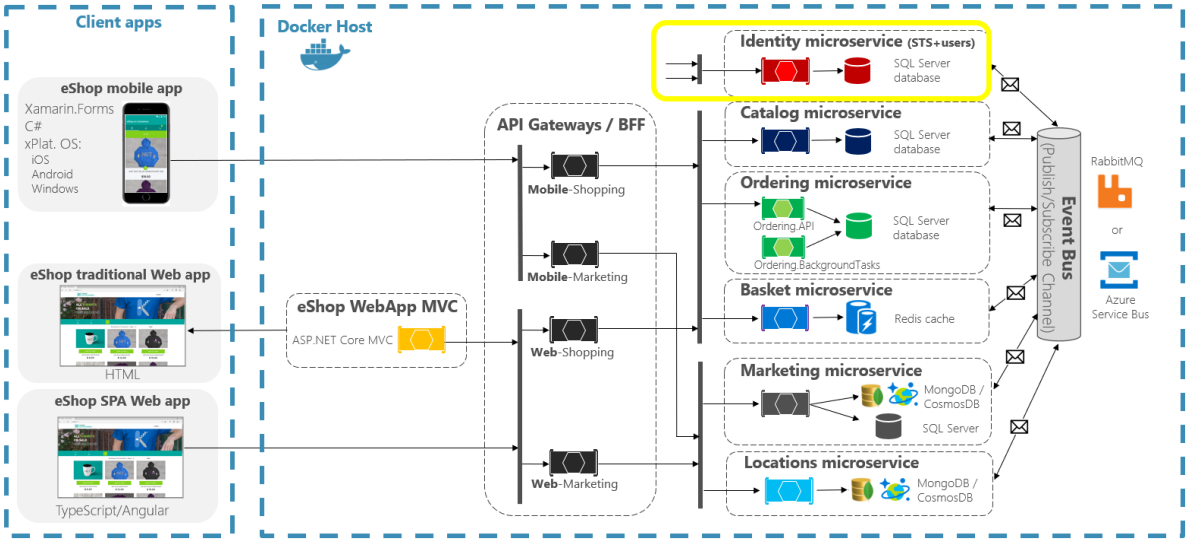


Figure 6-39. Position of the Identity service in eShopOnContainers

However, Ocelot also supports sitting the Identity/Auth microservice within the API Gateway boundary, as in this other diagram.

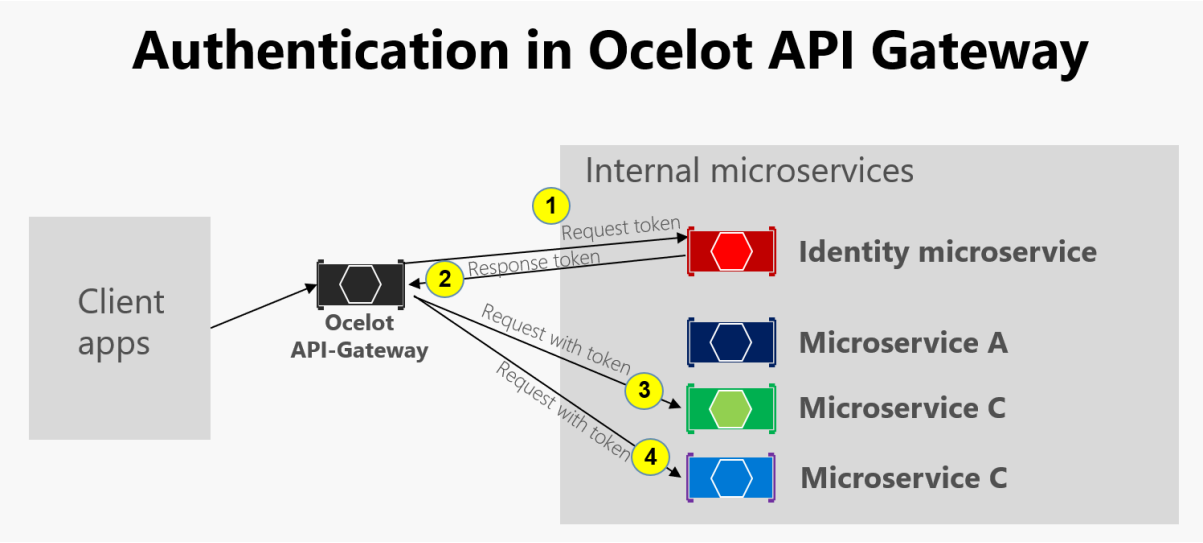


Figure 6-40. Authentication in Ocelot

As the previous diagram shows, when the Identity microservice is beneath the API gateway (AG): 1) AG requests an auth token from identity microservice, 2) The identity microservice returns token to AG, 3-4) AG requests from microservices using the auth token. Because eShopOnContainers application has split the API Gateway into multiple BFF (Backend for Frontend) and business areas API Gateways, another option would have been to create an additional API Gateway for cross-cutting concerns. That choice would be fair in a more complex microservice based architecture with multiple cross-cutting concerns microservices. Since there's only one cross-cutting concern in eShopOnContainers, it was decided to just handle the security service out of the API Gateway realm, for simplicity's sake.

In any case, if the app is secured at the API Gateway level, the authentication module of the Ocelot API Gateway is visited at first when trying to use any secured microservice. That redirects the HTTP request to visit the Identity or auth microservice to get the access token so you can visit the protected services with the access_token.

The way you secure with authentication any service at the API Gateway level is by setting the AuthenticationProviderKey in its related settings at the configuration.json.

```
    {
      "DownstreamPathTemplate": "/api/{version}/{everything}",
      "DownstreamScheme": "http",
      "DownstreamHostAndPorts": [
        {
          "Host": "basket-api",
          "Port": 80
        }
      ],
      "UpstreamPathTemplate": "/api/{version}/b/{everything}",
      "UpstreamHttpMethod": [],
      "AuthenticationOptions": {
        "AuthenticationProviderKey": "IdentityApiKey",
        "AllowedScopes": []
      }
    }
```

When Ocelot runs, it will look at the ReRoutes AuthenticationOptions.AuthenticationProviderKey and check that there is an Authentication Provider registered with the given key. If there isn't, then Ocelot will not start up. If there is, then the ReRoute will use that provider when it executes.

Because the Ocelot WebHost is configured with the authenticationProviderKey = "IdentityApiKey", that will require authentication whenever that service has any requests without any auth token.

```
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" }
                    };
                });
            //...
```

```
        }
    }
}
```

Then, you also need to set authorization with the [Authorize] attribute on any resource to be accessed like the microservices, such as in the following Basket microservice controller.

```
namespace Microsoft.eShopOnContainers.Services.Basket.API.Controllers
{
    [Route("api/v1/[controller]")]
    [Authorize]
    public class BasketController : Controller
    {
      //...
    }
}
```

The ValidAudiences such as "basket" are correlated with the audience defined in each microservice with AddJwtBearer() at the ConfigureServices() of the Startup class, such as in the code below.

```
// 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";
});
```

If you try to access any secured microservice, like the Basket microservice with a ReRoute URL based on the API Gateway like http://host.docker.internal:5202/api/v1/b/basket/1, then you'll get a 401 Unauthorized unless you provide a valid token. On the other hand, if a ReRoute URL is authenticated, Ocelot will invoke whatever downstream scheme is associated with it (the internal microservice URL).

Authorization at Ocelot's ReRoutes tier. Ocelot supports claims-based authorization evaluated after the authentication. You set the authorization at a route level by adding the following lines to the ReRoute configuration.

```
"RouteClaimsRequirement": {
    "UserType": "employee"
}
```

In that example, when the authorization middleware is called, Ocelot will find if the user has the claim type 'UserType' in the token and if the value of that claim is 'employee'. If it isn't, then the user will not be authorized and the response will be 403 forbidden.

Using Kubernetes Ingress plus Ocelot API Gateways

When using Kubernetes (like in an Azure Kubernetes Service cluster), you usually unify all the HTTP requests through the Kubernetes Ingress tier based on *Nginx*.

In Kubernetes, if you don't use any ingress approach, then your services and pods have IPs only routable by the cluster network.

But if you use an ingress approach, you'll have a middle tier between the Internet and your services (including your API Gateways), acting as a reverse proxy.

As a definition, an Ingress is a collection of rules that allow inbound connections to reach the cluster services. An ingress is configured to provide services externally reachable URLs, load balance traffic, SSL termination and more. Users request ingress by POSTing the Ingress resource to the API server.

In eShopOnContainers, when developing locally and using just your development machine as the Docker host, you are not using any ingress but only the multiple API Gateways.

However, when targeting a "production" environment based on Kubernetes, eShopOnContainers is using an ingress in front of the API gateways. That way, the clients still call the same base URL but the requests are routed to multiple API Gateways or BFF.

API Gateways are front-ends or façades surfacing only the services but not the web applications that are usually out of their scope. In addition, the API Gateways might hide certain internal microservices.

The ingress, however, is just redirecting HTTP requests but not trying to hide any microservice or web app.

Having an ingress Nginx tier in Kubernetes in front of the web applications plus the several Ocelot API Gateways / BFF is the ideal architecture, as shown in the following diagram.

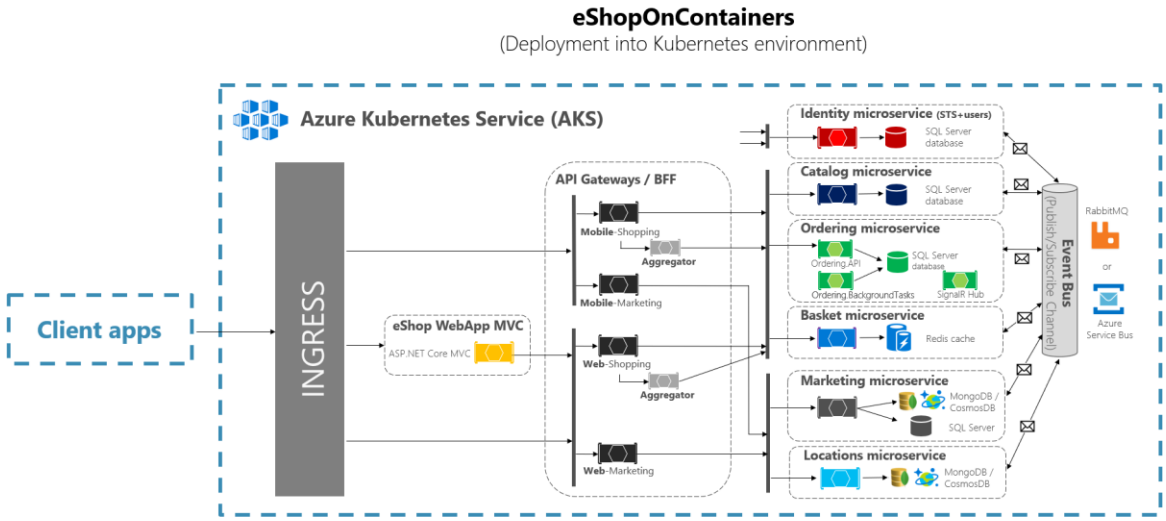


Figure 6-41. The ingress tier in eShopOnContainers when deployed into Kubernetes

A Kubernetes Ingress acts as a reverse proxy for all traffic to the app, including the web applications, that are out of the Api gateway scope. When you deploy eShopOnContainers into Kubernetes, it

exposes just a few services or endpoints via *ingress*, basically the following list of postfixes on the URLs:

- / for the client SPA web application
- /webmvc for the client MVC web application
- /webstatus for the client web app showing the status/healthchecks
- /webshoppingapigw for the web BFF and shopping business processes
- /webmarketingapigw for the web BFF and marketing business processes
- /mobileshoppingapigw for the mobile BFF and shopping business processes
- /mobilemarketingapigw for the mobile BFF and marketing business processes

When deploying to Kubernetes, each Ocelot API Gateway is using a different "configuration.json" file for each *pod* running the API Gateways. Those "configuration.json" files are provided by mounting (originally with the deploy.ps1 script) a volume created based on a Kubernetes *config map* named 'ocelot'. Each container mounts its related configuration file in the container's folder named /app/configuration.

In the source code files of eShopOnContainers, the original "configuration.json" files can be found within the k8s/ocelot/ folder. There's one file for each BFF/APIGateway.

Additional cross-cutting features in an Ocelot API Gateway

There are other important features to research and use, when using an Ocelot API Gateway, described in the following links.

- **Service discovery in the client side integrating Ocelot with Consul or Eureka**
  https://ocelot.readthedocs.io/en/latest/features/servicediscovery.html

- **Caching at the API Gateway tier**
  https://ocelot.readthedocs.io/en/latest/features/caching.html

- **Logging at the API Gateway tier**
  https://ocelot.readthedocs.io/en/latest/features/logging.html

- **Quality of Service (Retries and Circuit breakers) at the API Gateway tier**
  https://ocelot.readthedocs.io/en/latest/features/qualityofservice.html

- **Rate limiting**
  https://ocelot.readthedocs.io/en/latest/features/ratelimiting.html

- **Swagger for Ocelot**
  https://github.com/Burgyn/MMLib.SwaggerForOcelot

CHAPTER 6

Tackle Business Complexity in a Microservice with DDD and CQRS Patterns

Design a domain model for each microservice or Bounded Context that reflects understanding of the business domain.

This section focuses on more advanced microservices that you implement when you need to tackle complex subsystems, or microservices derived from the knowledge of domain experts with ever-changing business rules. The architecture patterns used in this section are based on domain-driven design (DDD) and Command and Query Responsibility Segregation (CQRS) approaches, as illustrated in Figure 7-1.

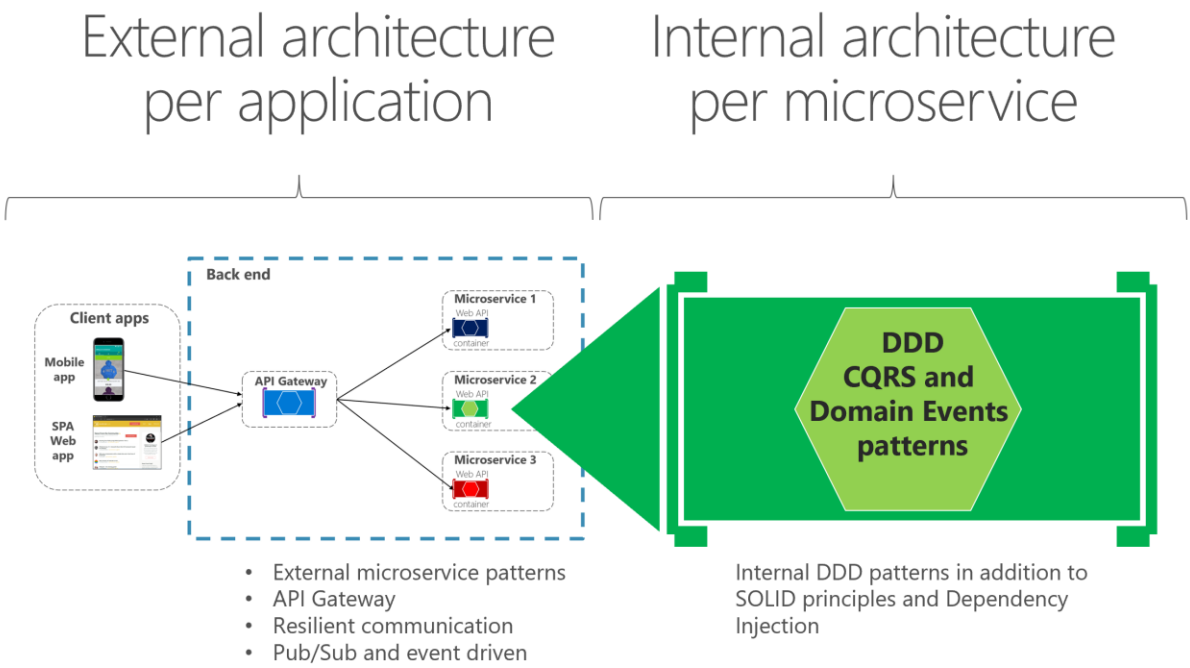


Figure 7-1. External microservice architecture versus internal architecture patterns for each microservice

However, most of the techniques for data driven microservices, such as how to implement an ASP.NET Core Web API service or how to expose Swagger metadata with Swashbuckle or NSwag, are also applicable to the more advanced microservices implemented internally with DDD patterns. This section is an extension of the previous sections, because most of the practices explained earlier also apply here or for any kind of microservice.

This section first provides details on the simplified CQRS patterns used in the eShopOnContainers reference application. Later, you will get an overview of the DDD techniques that enable you to find common patterns that you can reuse in your applications.

DDD is a large topic with a rich set of resources for learning. You can start with books like Domain-Driven Design by Eric Evans and additional materials from Vaughn Vernon, Jimmy Nilsson, Greg Young, Udi Dahan, Jimmy Bogard, and many other DDD/CQRS experts. But most of all you need to try to learn how to apply DDD techniques from the conversations, whiteboarding, and domain modeling sessions with the experts in your concrete business domain.

Additional resources

DDD (Domain-Driven Design)

- **Eric Evans. Domain Language**
  https://domainlanguage.com/
- **Martin Fowler. Domain-Driven Design**
  https://martinfowler.com/tags/domain%20driven%20design.html
- **Jimmy Bogard. Strengthening your domain: a primer**
  https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/

DDD books

- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software**
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/
- **Eric Evans. Domain-Driven Design Reference: Definitions and Pattern Summaries**
  https://www.amazon.com/Domain-Driven-Design-Reference-Definitions-2014-09-22/dp/B01N8YB4ZO/
- **Vaughn Vernon. Implementing Domain-Driven Design**
  https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/
- **Vaughn Vernon. Domain-Driven Design Distilled**
  https://www.amazon.com/Domain-Driven-Design-Distilled-Vaughn-Vernon/dp/0134434420/
- **Jimmy Nilsson. Applying Domain-Driven Design and Patterns**
  https://www.amazon.com/Applying-Domain-Driven-Design-Patterns-Examples/dp/0321268202/
- **Cesar de la Torre. N-Layered Domain-Oriented Architecture Guide with .NET**
  https://www.amazon.com/N-Layered-Domain-Oriented-Architecture-Guide-NET/dp/8493903612/
- **Abel Avram and Floyd Marinescu. Domain-Driven Design Quickly**
  https://www.amazon.com/Domain-Driven-Design-Quickly-Abel-Avram/dp/1411609255/
- **Scott Millett, Nick Tune - Patterns, Principles, and Practices of Domain-Driven Design**
  https://www.wiley.com/Patterns%2C+Principles%2C+and+Practices+of+Domain+Driven+Design-p-9781118714706

DDD training

- **Julie Lerman and Steve Smith. Domain-Driven Design Fundamentals**
  https://www.pluralsight.com/courses/fundamentals-domain-driven-design

Apply simplified CQRS and DDD patterns in a microservice

CQRS is an architectural pattern that separates the models for reading and writing data. The related term Command Query Separation (CQS) was originally defined by Bertrand Meyer in his book *Object-Oriented Software Construction*. The basic idea is that you can divide a system's operations into two sharply separated categories:

- Queries. These queries return a result and don't change the state of the system, and they're free of side effects.
- Commands. These commands change the state of a system.

CQS is a simple concept: it is about methods within the same object being either queries or commands. Each method either returns state or mutates state, but not both. Even a single repository pattern object can comply with CQS. CQS can be considered a foundational principle for CQRS.

Command and Query Responsibility Segregation (CQRS) was introduced by Greg Young and strongly promoted by Udi Dahan and others. It's based on the CQS principle, although it's more detailed. It can be considered a pattern based on commands and events plus optionally on asynchronous messages. In many cases, CQRS is related to more advanced scenarios, like having a different physical database for reads (queries) than for writes (updates). Moreover, a more evolved CQRS system might implement Event-Sourcing (ES) for your updates database, so you would only store events in the domain model instead of storing the current-state data. However, this approach is not used in this guide. This guide uses the simplest CQRS approach, which consists of just separating the queries from the commands.

The separation aspect of CQRS is achieved by grouping query operations in one layer and commands in another layer. Each layer has its own data model (note that we say model, not necessarily a different database) and is built using its own combination of patterns and technologies. More importantly, the two layers can be within the same tier or microservice, as in the example (ordering microservice) used for this guide. Or they could be implemented on different microservices or processes so they can be optimized and scaled out separately without affecting one another.

CQRS means having two objects for a read/write operation where in other contexts there's one. There are reasons to have a denormalized reads database, which you can learn about in more advanced CQRS literature. But we aren't using that approach here, where the goal is to have more flexibility in the queries instead of limiting the queries with constraints from DDD patterns like aggregates.

An example of this kind of service is the ordering microservice from the eShopOnContainers reference application. This service implements a microservice based on a simplified CQRS approach. It uses a single data source or database, but two logical models plus DDD patterns for the transactional domain, as shown in Figure 7-2.

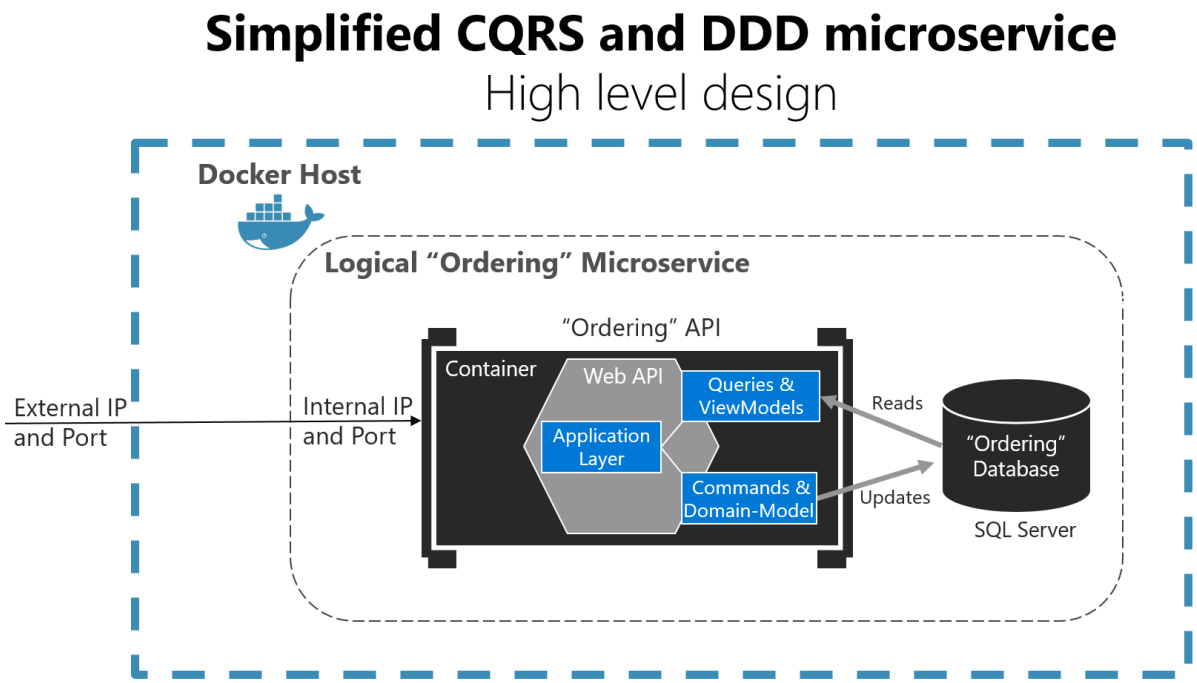


Figure 7-2. Simplified CQRS- and DDD-based microservice

The Logical "Ordering" Microservice includes its Ordering database, which can be, but doesn't have to be, the same Docker host. Having the database in the same Docker host is good for development, but not for production.

The application layer can be the Web API itself. The important design aspect here is that the microservice has split the queries and ViewModels (data models especially created for the client applications) from the commands, domain model, and transactions following the CQRS pattern. This approach keeps the queries independent from restrictions and constraints coming from DDD patterns that only make sense for transactions and updates, as explained in later sections.

Additional resources

- **Greg Young. Versioning in an Event Sourced System** (Free to read online e-book) https://leanpub.com/esversioning/read

Apply CQRS and CQS approaches in a DDD microservice in eShopOnContainers

The design of the ordering microservice at the eShopOnContainers reference application is based on CQRS principles. However, it uses the simplest approach, which is just separating the queries from the commands and using the same database for both actions.

The essence of those patterns, and the important point here, is that queries are idempotent: no matter how many times you query a system, the state of that system won't change. In other words, queries are side-effect free.

Therefore, you could use a different "reads" data model than the transactional logic "writes" domain model, even though the ordering microservices are using the same database. Hence, this is a simplified CQRS approach.

On the other hand, commands, which trigger transactions and data updates, change state in the system. With commands, you need to be careful when dealing with complexity and ever-changing business rules. This is where you want to apply DDD techniques to have a better modeled system.

The DDD patterns presented in this guide should not be applied universally. They introduce constraints on your design. Those constraints provide benefits such as higher quality over time, especially in commands and other code that modifies system state. However, those constraints add complexity with fewer benefits for reading and querying data.

One such pattern is the Aggregate pattern, which we examine more in later sections. Briefly, in the Aggregate pattern, you treat many domain objects as a single unit as a result of their relationship in the domain. You might not always gain advantages from this pattern in queries; it can increase the complexity of query logic. For read-only queries, you do not get the advantages of treating multiple objects as a single Aggregate. You only get the complexity.

As shown in Figure 7-2 in the previous section, this guide suggests using DDD patterns only in the transactional/updates area of your microservice (that is, as triggered by commands). Queries can follow a simpler approach and should be separated from commands, following a CQRS approach.

For implementing the "queries side", you can choose between many approaches, from your full-blown ORM like EF Core, AutoMapper projections, stored procedures, views, materialized views or a micro ORM.

In this guide and in eShopOnContainers (specifically the ordering microservice) we chose to implement straight queries using a micro ORM like Dapper. This guide lets you implement any query based on SQL statements to get the best performance, thanks to a light framework with little overhead.

When you use this approach, any updates to your model that impact how entities are persisted to a SQL database also need separate updates to SQL queries used by Dapper or any other separate (non-EF) approaches to querying.

CQRS and DDD patterns are not top-level architectures

It's important to understand that CQRS and most DDD patterns (like DDD layers or a domain model with aggregates) are not architectural styles, but only architecture patterns. Microservices, SOA, and event-driven architecture (EDA) are examples of architectural styles. They describe a system of many components, such as many microservices. CQRS and DDD patterns describe something inside a single system or component; in this case, something inside a microservice.

Different Bounded Contexts (BCs) will employ different patterns. They have different responsibilities, and that leads to different solutions. It is worth emphasizing that forcing the same pattern everywhere leads to failure. Do not use CQRS and DDD patterns everywhere. Many subsystems, BCs, or microservices are simpler and can be implemented more easily using simple CRUD services or using another approach.

There is only one application architecture: the architecture of the system or end-to-end application you are designing (for example, the microservices architecture). However, the design of each Bounded Context or microservice within that application reflects its own tradeoffs and internal design decisions at an architecture patterns level. Do not try to apply the same architectural patterns as CQRS or DDD everywhere.

Additional resources

- **Martin Fowler. CQRS**
  https://martinfowler.com/bliki/CQRS.html
- **Greg Young. CQRS Documents**
  https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf
- **Udi Dahan. Clarified CQRS**
  https://udidahan.com/2009/12/09/clarified-cqrs/

Implement reads/queries in a CQRS microservice

For reads/queries, the ordering microservice from the eShopOnContainers reference application implements the queries independently from the DDD model and transactional area. This implementation was done primarily because the demands for queries and for transactions are drastically different. Writes execute transactions that must be compliant with the domain logic. Queries, on the other hand, are idempotent and can be segregated from the domain rules.

The approach is simple, as shown in Figure 7-3. The API interface is implemented by the Web API controllers using any infrastructure, such as a micro Object Relational Mapper (ORM) like Dapper, and returning dynamic ViewModels depending on the needs of the UI applications.

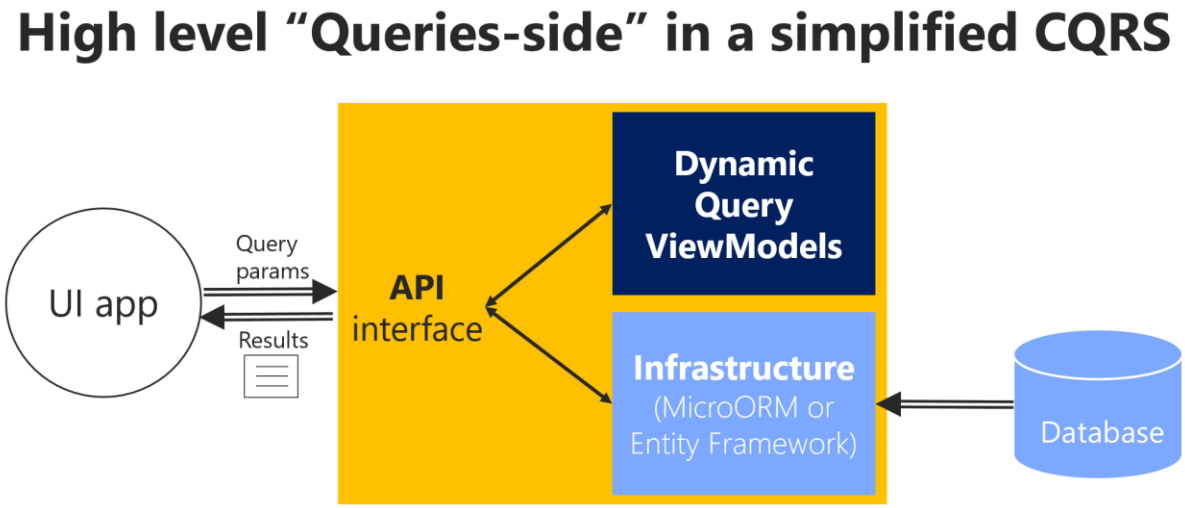


Figure 7-3. The simplest approach for queries in a CQRS microservice

The simplest approach for the queries-side in a simplified CQRS approach can be implemented by querying the database with a Micro-ORM like Dapper, returning dynamic ViewModels. The query

definitions query the database and return a dynamic ViewModel built on the fly for each query. Since the queries are idempotent, they won't change the data no matter how many times you run a query. Therefore, you don't need to be restricted by any DDD pattern used in the transactional side, like aggregates and other patterns, and that is why queries are separated from the transactional area. You query the database for the data that the UI needs and return a dynamic ViewModel that does not need to be statically defined anywhere (no classes for the ViewModels) except in the SQL statements themselves.

Since this approach is simple, the code required for the queries side (such as code using a micro ORM like Dapper) can be implemented within the same Web API project. Figure 7-4 shows this approach. The queries are defined in the **Ordering.API** microservice project within the eShopOnContainers solution.

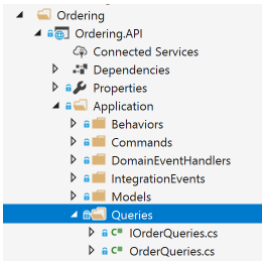


Figure 7-4. Queries in the Ordering microservice in eShopOnContainers

Use ViewModels specifically made for client apps, independent from domain model constraints

Since the queries are performed to obtain the data needed by the client applications, the returned type can be specifically made for the clients, based on the data returned by the queries. These models, or Data Transfer Objects (DTOs), are called ViewModels.

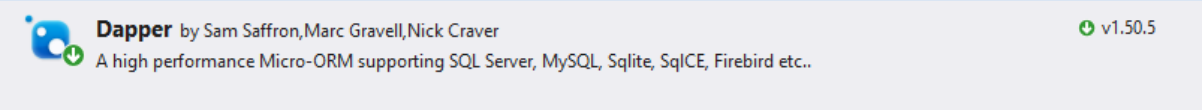
The returned data (ViewModel) can be the result of joining data from multiple entities or tables in the database, or even across multiple aggregates defined in the domain model for the transactional area. In this case, because you are creating queries independent of the domain model, the aggregates boundaries and constraints are ignored and you're free to query any table and column you might need. This approach provides great flexibility and productivity for the developers creating or updating the queries.

The ViewModels can be static types defined in classes (as is implemented in the ordering microservice). Or they can be created dynamically based on the queries performed, which is agile for developers.

Use Dapper as a micro ORM to perform queries

You can use any micro ORM, Entity Framework Core, or even plain ADO.NET for querying. In the sample application, Dapper was selected for the ordering microservice in eShopOnContainers as a good example of a popular micro ORM. It can run plain SQL queries with great performance, because it's a light framework. Using Dapper, you can write a SQL query that can access and join multiple tables.

Dapper is an open-source project (original created by Sam Saffron), and is part of the building blocks used in Stack Overflow. To use Dapper, you just need to install it through the Dapper NuGet package, as shown in the following figure:



You also need to add a using directive so your code has access to the Dapper extension methods.

When you use Dapper in your code, you directly use the SqlConnection class available in the Microsoft.Data.SqlClient namespace. Through the QueryAsync method and other extension methods that extend the SqlConnection class, you can run queries in a straightforward and performant way.

Dynamic versus static ViewModels

When returning ViewModels from the server-side to client apps, you can think about those ViewModels as DTOs (Data Transfer Objects) that can be different to the internal domain entities of your entity model because the ViewModels hold the data the way the client app needs. Therefore, in many cases, you can aggregate data coming from multiple domain entities and compose the ViewModels precisely according to how the client app needs that data.

Those ViewModels or DTOs can be defined explicitly (as data holder classes), like the OrderSummary class shown in a later code snippet. Or, you could just return dynamic ViewModels or dynamic DTOs based on the attributes returned by your queries as a dynamic type.

ViewModel as dynamic type

As shown in the following code, a ViewModel can be directly returned by the queries by just returning a *dynamic* type that internally is based on the attributes returned by a query. That means that the subset of attributes to be returned is based on the query itself. Therefore, if you add a new column to the query or join, that data is dynamically added to the returned ViewModel.

```
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]");
        }
    }
}
```

The important point is that by using a dynamic type, the returned collection of data is dynamically assembled as the ViewModel.

Pros: This approach reduces the need to modify static ViewModel classes whenever you update the SQL sentence of a query, making this design approach agile when coding, straightforward, and quick to evolve in regard to future changes.

Cons: In the long term, dynamic types can negatively impact the clarity and the compatibility of a service with client apps. In addition, middleware software like Swashbuckle cannot provide the same level of documentation on returned types if using dynamic types.

ViewModel as predefined DTO classes

Pros: Having static, predefined ViewModel classes, like "contracts" based on explicit DTO classes, is definitely better for public APIs but also for long-term microservices, even if they are only used by the same application.

If you want to specify response types for Swagger, you need to use explicit DTO classes as the return type. Therefore, predefined DTO classes allow you to offer richer information from Swagger. That improves the API documentation and compatibility when consuming an API.

Cons: As mentioned earlier, when updating the code, it takes some more steps to update the DTO classes.

Tip based on our experience: In the queries implemented at the Ordering microservice in eShopOnContainers, we started developing by using dynamic ViewModels as it was straightforward and agile on the early development stages. But, once the development was stabilized, we chose to refactor the APIs and use static or pre-defined DTOs for the ViewModels, because it is clearer for the microservice's consumers to know explicit DTO types, used as "contracts".

In the following example, you can see how the query is returning data by using an explicit ViewModel DTO class: the OrderSummary class.

```
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]");
        }
    }
}
```

Describe response types of Web APIs

Developers consuming web APIs and microservices are most concerned with what is returned—specifically response types and error codes (if not standard). The response types are handled in the XML comments and data annotations.

Without proper documentation in the Swagger UI, the consumer lacks knowledge of what types are being returned or what HTTP codes can be returned. That problem is fixed by adding the Microsoft.AspNetCore.Mvc.ProducesResponseTypeAttribute, so Swashbuckle can generate richer information about the API return model and values, as shown in the following code:

```
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);
        }
    }
}
```

However, the ProducesResponseType attribute cannot use dynamic as a type but requires to use explicit types, like the OrderSummary ViewModel DTO, shown in the following example:

```
public class OrderSummary
{
    public int ordernumber { get; set; }
    public DateTime date { get; set; }
    public string status { get; set; }
    public double total { get; set; }
}
// or using C# 8 record types:
public record OrderSummary(int ordernumber, DateTime date, string status, double total);
```

This is another reason why explicit returned types are better than dynamic types, in the long term. When using the ProducesResponseType attribute, you can also specify what is the expected outcome regarding possible HTTP errors/codes, like 200, 400, etc.

In the following image, you can see how Swagger UI shows the ResponseType information.

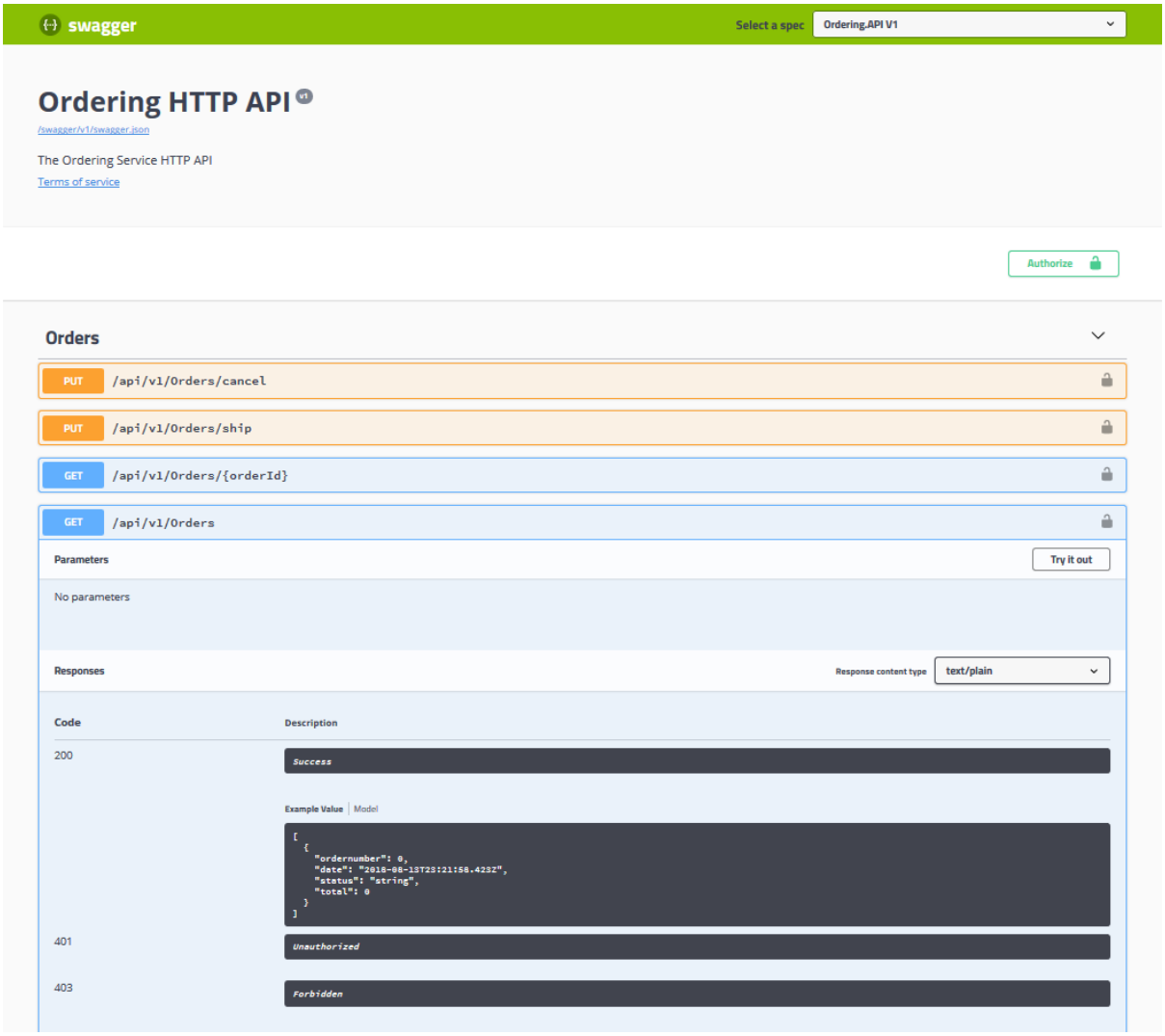


Figure 7-5. Swagger UI showing response types and possible HTTP status codes from a Web API

The image shows some example values based on the ViewModel types and the possible HTTP status codes that can be returned.

Additional resources

- **Dapper**
  https://github.com/StackExchange/dapper-dot-net
- **Julie Lerman. Data Points - Dapper, Entity Framework and Hybrid Apps (MSDN magazine article)**

https://learn.microsoft.com/archive/msdn-magazine/2016/may/data-points-dapper-entity-framework-and-hybrid-apps

- **ASP.NET Core Web API Help Pages using Swagger** https://learn.microsoft.com/aspnet/core/tutorials/web-api-help-pages-using-swagger?tabs=visual-studio
- **Create record types** https://learn.microsoft.com/dotnet/csharp/whats-new/tutorials/records

Design a DDD-oriented microservice

Domain-driven design (DDD) advocates modeling based on the reality of business as relevant to your use cases. In the context of building applications, DDD talks about problems as domains. It describes independent problem areas as Bounded Contexts (each Bounded Context correlates to a microservice), and emphasizes a common language to talk about these problems. It also suggests many technical concepts and patterns, like domain entities with rich models (no anemic-domain model), value objects, aggregates, and aggregate root (or root entity) rules to support the internal implementation. This section introduces the design and implementation of those internal patterns.

Sometimes these DDD technical rules and patterns are perceived as obstacles that have a steep learning curve for implementing DDD approaches. But the important part is not the patterns themselves, but organizing the code so it is aligned to the business problems, and using the same business terms (ubiquitous language). In addition, DDD approaches should be applied only if you are implementing complex microservices with significant business rules. Simpler responsibilities, like a CRUD service, can be managed with simpler approaches.

Where to draw the boundaries is the key task when designing and defining a microservice. DDD patterns help you understand the complexity in the domain. For the domain model for each Bounded Context, you identify and define the entities, value objects, and aggregates that model your domain. You build and refine a domain model that is contained within a boundary that defines your context. And that is explicit in the form of a microservice. The components within those boundaries end up being your microservices, although in some cases a BC or business microservices can be composed of several physical services. DDD is about boundaries and so are microservices.

Keep the microservice context boundaries relatively small

Determining where to place boundaries between Bounded Contexts balances two competing goals. First, you want to initially create the smallest possible microservices, although that should not be the main driver; you should create a boundary around things that need cohesion. Second, you want to avoid chatty communications between microservices. These goals can contradict one another. You should balance them by decomposing the system into as many small microservices as you can until you see communication boundaries growing quickly with each additional attempt to separate a new Bounded Context. Cohesion is key within a single bounded context.

It is similar to the Inappropriate Intimacy code smell when implementing classes. If two microservices need to collaborate a lot with each other, they should probably be the same microservice.

Another way to look at this aspect is autonomy. If a microservice must rely on another service to directly service a request, it is not truly autonomous.

Layers in DDD microservices

Most enterprise applications with significant business and technical complexity are defined by multiple layers. The layers are a logical artifact, and are not related to the deployment of the service. They exist to help developers manage the complexity in the code. Different layers (like the domain model layer versus the presentation layer, etc.) might have different types, which mandate translations between those types.

For example, an entity could be loaded from the database. Then part of that information, or an aggregation of information including additional data from other entities, can be sent to the client UI through a REST Web API. The point here is that the domain entity is contained within the domain model layer and should not be propagated to other areas that it does not belong to, like to the presentation layer.

Additionally, you need to have always-valid entities (see the Designing validations in the domain model layer section) controlled by aggregate roots (root entities). Therefore, entities should not be bound to client views, because at the UI level some data might still not be validated. This reason is what the ViewModel is for. The ViewModel is a data model exclusively for presentation layer needs. The domain entities do not belong directly to the ViewModel. Instead, you need to translate between ViewModels and domain entities and vice versa.

When tackling complexity, it is important to have a domain model controlled by aggregate roots that make sure that all the invariants and rules related to that group of entities (aggregate) are performed through a single entry-point or gate, the aggregate root.

Figure 7-5 shows how a layered design is implemented in the eShopOnContainers application.

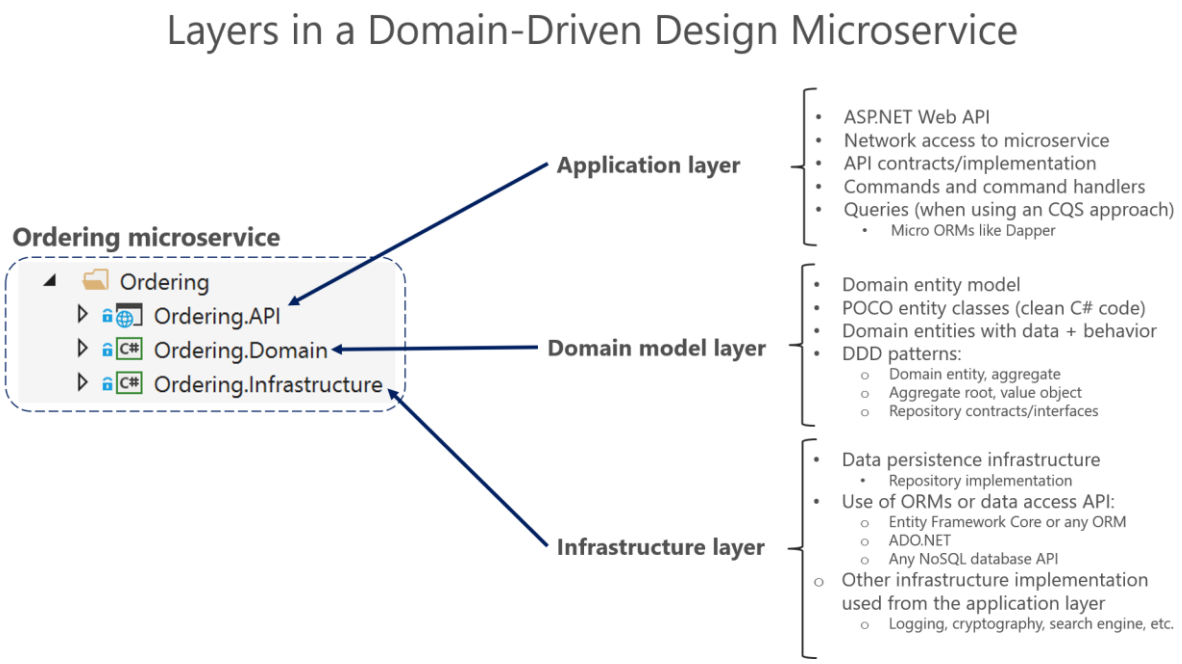


Figure 7-5. DDD layers in the ordering microservice in eShopOnContainers

The three layers in a DDD microservice like Ordering. Each layer is a VS project: Application layer is Ordering.API, Domain layer is Ordering.Domain and the Infrastructure layer is Ordering.Infrastructure. You want to design the system so that each layer communicates only with certain other layers. That approach may be easier to enforce if layers are implemented as different class libraries, because you can clearly identify what dependencies are set between libraries. For instance, the domain model layer should not take a dependency on any other layer (the domain model classes should be Plain Old Class Objects, or POCO, classes). As shown in Figure 7-6, the **Ordering.Domain** layer library has dependencies only on the .NET libraries or NuGet packages, but not on any other custom library, such as data library or persistence library.

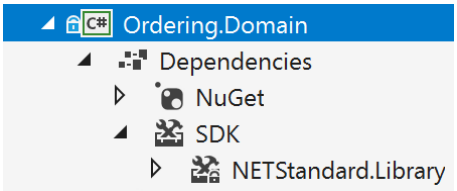


Figure 7-6. Layers implemented as libraries allow better control of dependencies between layers

The domain model layer

Eric Evans's excellent book Domain Driven Design says the following about the domain model layer and the application layer.

Domain Model Layer: Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.

The domain model layer is where the business is expressed. When you implement a microservice domain model layer in .NET, that layer is coded as a class library with the domain entities that capture data plus behavior (methods with logic).

Following the Persistence Ignorance and the Infrastructure Ignorance principles, this layer must completely ignore data persistence details. These persistence tasks should be performed by the infrastructure layer. Therefore, this layer should not take direct dependencies on the infrastructure, which means that an important rule is that your domain model entity classes should be POCOs.

Domain entities should not have any direct dependency (like deriving from a base class) on any data access infrastructure framework like Entity Framework or NHibernate. Ideally, your domain entities should not derive from or implement any type defined in any infrastructure framework.

Most modern ORM frameworks like Entity Framework Core allow this approach, so that your domain model classes are not coupled to the infrastructure. However, having POCO entities is not always possible when using certain NoSQL databases and frameworks, like Actors and Reliable Collections in Azure Service Fabric.

Even when it is important to follow the Persistence Ignorance principle for your Domain model, you should not ignore persistence concerns. It is still important to understand the physical data model and how it maps to your entity object model. Otherwise you can create impossible designs.

Also, this aspect does not mean you can take a model designed for a relational database and directly move it to a NoSQL or document-oriented database. In some entity models, the model might fit, but usually it does not. There are still constraints that your entity model must adhere to, based both on the storage technology and ORM technology.

The application layer

Moving on to the application layer, we can again cite Eric Evans's book Domain Driven Design:

Application Layer: Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.

A microservice's application layer in .NET is commonly coded as an ASP.NET Core Web API project. The project implements the microservice's interaction, remote network access, and the external Web APIs used from the UI or client apps. It includes queries if using a CQRS approach, commands accepted by the microservice, and even the event-driven communication between microservices (integration events). The ASP.NET Core Web API that represents the application layer must not contain business rules or domain knowledge (especially domain rules for transactions or updates); these should be owned by the domain model class library. The application layer must only coordinate tasks and must not hold or define any domain state (domain model). It delegates the execution of business rules to the domain model classes themselves (aggregate roots and domain entities), which will ultimately update the data within those domain entities.

Basically, the application logic is where you implement all use cases that depend on a given front end. For example, the implementation related to a Web API service.

The goal is that the domain logic in the domain model layer, its invariants, the data model, and related business rules must be completely independent from the presentation and application layers. Most of all, the domain model layer must not directly depend on any infrastructure framework.

The infrastructure layer

The infrastructure layer is how the data that is initially held in domain entities (in memory) is persisted in databases or another persistent store. An example is using Entity Framework Core code to implement the Repository pattern classes that use a DBContext to persist data in a relational database.

In accordance with the previously mentioned Persistence Ignorance and Infrastructure Ignorance principles, the infrastructure layer must not "contaminate" the domain model layer. You must keep the domain model entity classes agnostic from the infrastructure that you use to persist data (EF or any other framework) by not taking hard dependencies on frameworks. Your domain model layer class library should have only your domain code, just POCO entity classes implementing the heart of your software and completely decoupled from infrastructure technologies.

Thus, your layers or class libraries and projects should ultimately depend on your domain model layer (library), not vice versa, as shown in Figure 7-7.

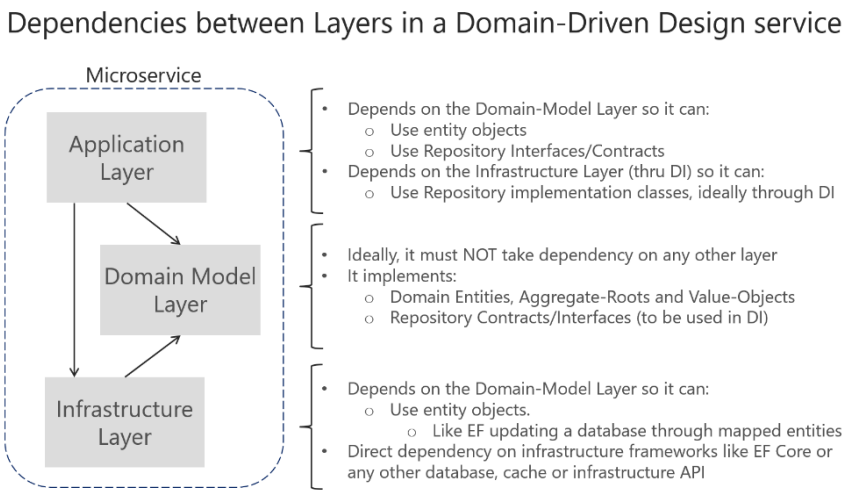


Figure 7-7. Dependencies between layers in DDD

Dependencies in a DDD Service, the Application layer depends on Domain and Infrastructure, and Infrastructure depends on Domain, but Domain doesn't depend on any layer. This layer design should be independent for each microservice. As noted earlier, you can implement the most complex microservices following DDD patterns, while implementing simpler data-driven microservices (simple CRUD in a single layer) in a simpler way.

Additional resources

- **DevIQ. Persistence Ignorance principle**
  https://deviq.com/persistence-ignorance/
- **Oren Eini. Infrastructure Ignorance**
  https://ayende.com/blog/3137/infrastructure-ignorance
- **Angel Lopez. Layered Architecture In Domain-Driven Design**
  https://ajlopez.wordpress.com/2008/09/12/layered-architecture-in-domain-driven-design/

Design a microservice domain model

Define one rich domain model for each business microservice or Bounded Context.

Your goal is to create a single cohesive domain model for each business microservice or Bounded Context (BC). Keep in mind, however, that a BC or business microservice could sometimes be composed of several physical services that share a single domain model. The domain model must capture the rules, behavior, business language, and constraints of the single Bounded Context or business microservice that it represents.

The Domain Entity pattern

Entities represent domain objects and are primarily defined by their identity, continuity, and persistence over time, and not only by the attributes that comprise them. As Eric Evans says, "an object primarily defined by its identity is called an Entity." Entities are very important in the domain model, since they are the base for a model. Therefore, you should identify and design them carefully.

An entity's identity can cross multiple microservices or Bounded Contexts.

The same identity (that is, the same Id value, although perhaps not the same domain entity) can be modeled across multiple Bounded Contexts or microservices. However, that does not imply that the same entity, with the same attributes and logic would be implemented in multiple Bounded Contexts. Instead, entities in each Bounded Context limit their attributes and behaviors to those required in that Bounded Context's domain.

For instance, the buyer entity might have most of a person's attributes that are defined in the user entity in the profile or identity microservice, including the identity. But the buyer entity in the ordering microservice might have fewer attributes, because only certain buyer data is related to the order process. The context of each microservice or Bounded Context impacts its domain model.

Domain entities must implement behavior in addition to implementing data attributes.

A domain entity in DDD must implement the domain logic or behavior related to the entity data (the object accessed in memory). For example, as part of an order entity class you must have business logic and operations implemented as methods for tasks such as adding an order item, data validation, and total calculation. The entity's methods take care of the invariants and rules of the entity instead of having those rules spread across the application layer.

Figure 7-8 shows a domain entity that implements not only data attributes but operations or methods with related domain logic.

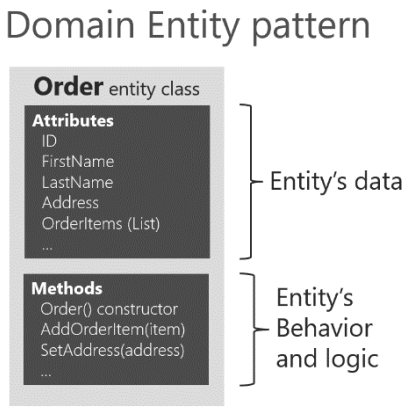


Figure 7-8. Example of a domain entity design implementing data plus behavior

A domain model entity implements behaviors through methods, that is, it's not an "anemic" model. Of course, sometimes you can have entities that do not implement any logic as part of the entity class. This can happen in child entities within an aggregate if the child entity does not have any special logic because most of the logic is defined in the aggregate root. If you have a complex microservice that has logic implemented in the service classes instead of in the domain entities, you could be falling into the anemic domain model, explained in the following section.

Rich domain model versus anemic domain model

In his post AnemicDomainModel, Martin Fowler describes an anemic domain model this way:

The basic symptom of an Anemic Domain Model is that at first blush it looks like the real thing. There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters.

Of course, when you use an anemic domain model, those data models will be used from a set of service objects (traditionally named the *business layer*) which capture all the domain or business logic. The business layer sits on top of the data model and uses the data model just as data.

The anemic domain model is just a procedural style design. Anemic entity objects are not real objects because they lack behavior (methods). They only hold data properties and thus it is not object-oriented design. By putting all the behavior out into service objects (the business layer), you essentially end up with spaghetti code or transaction scripts, and therefore you lose the advantages that a domain model provides.

Regardless, if your microservice or Bounded Context is very simple (a CRUD service), the anemic domain model in the form of entity objects with just data properties might be good enough, and it might not be worth implementing more complex DDD patterns. In that case, it will be simply a persistence model, because you have intentionally created an entity with only data for CRUD purposes.

That is why microservices architectures are perfect for a multi-architectural approach depending on each Bounded Context. For instance, in eShopOnContainers, the ordering microservice implements DDD patterns, but the catalog microservice, which is a simple CRUD service, does not.

Some people say that the anemic domain model is an anti-pattern. It really depends on what you are implementing. If the microservice you are creating is simple enough (for example, a CRUD service), following the anemic domain model it is not an anti-pattern. However, if you need to tackle the complexity of a microservice's domain that has a lot of ever-changing business rules, the anemic domain model might be an anti-pattern for that microservice or Bounded Context. In that case, designing it as a rich model with entities containing data plus behavior as well as implementing additional DDD patterns (aggregates, value objects, etc.) might have huge benefits for the long-term success of such a microservice.

Additional resources

- **DevIQ. Domain Entity**
  https://deviq.com/entity/
- **Martin Fowler. The Domain Model**
  https://martinfowler.com/eaaCatalog/domainModel.html
- **Martin Fowler. The Anemic Domain Model**
  https://martinfowler.com/bliki/AnemicDomainModel.html

The Value Object pattern

As Eric Evans has noted, "Many objects do not have conceptual identity. These objects describe certain characteristics of a thing."

An entity requires an identity, but there are many objects in a system that do not, like the Value Object pattern. A value object is an object with no conceptual identity that describes a domain aspect. These are objects that you instantiate to represent design elements that only concern you temporarily. You care about *what* they are, not *who* they are. Examples include numbers and strings, but can also be higher-level concepts like groups of attributes.

Something that is an entity in a microservice might not be an entity in another microservice, because in the second case, the Bounded Context might have a different meaning. For example, an address in an e-commerce application might not have an identity at all, since it might only represent a group of attributes of the customer's profile for a person or company. In this case, the address should be classified as a value object. However, in an application for an electric power utility company, the customer address could be important for the business domain. Therefore, the address must have an identity so the billing system can be directly linked to the address. In that case, an address should be classified as a domain entity.

A person with a name and surname is usually an entity because a person has identity, even if the name and surname coincide with another set of values, such as if those names also refer to a different person.

Value objects are hard to manage in relational databases and ORMs like Entity Framework (EF), whereas in document-oriented databases they are easier to implement and use.

EF Core 2.0 and later versions include the Owned Entities feature that makes it easier to handle value objects, as we'll see in detail later on.

Additional resources

- **Martin Fowler. Value Object pattern**
  https://martinfowler.com/bliki/ValueObject.html
- **Value Object**
  https://deviq.com/value-object/
- **Value Objects in Test-Driven Development**
  https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects
- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software.** (Book; includes a discussion of value objects)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/

The Aggregate pattern

A domain model contains clusters of different data entities and processes that can control a significant area of functionality, such as order fulfillment or inventory. A more fine-grained DDD unit is the aggregate, which describes a cluster or group of entities and behaviors that can be treated as a cohesive unit.

You usually define an aggregate based on the transactions that you need. A classic example is an order that also contains a list of order items. An order item will usually be an entity. But it will be a child entity within the order aggregate, which will also contain the order entity as its root entity, typically called an aggregate root.

Identifying aggregates can be hard. An aggregate is a group of objects that must be consistent together, but you cannot just pick a group of objects and label them an aggregate. You must start with a domain concept and think about the entities that are used in the most common transactions related to that concept. Those entities that need to be transactionally consistent are what forms an aggregate. Thinking about transaction operations is probably the best way to identify aggregates.

The Aggregate Root or Root Entity pattern

An aggregate is composed of at least one entity: the aggregate root, also called root entity or primary entity. Additionally, it can have multiple child entities and value objects, with all entities and objects working together to implement required behavior and transactions.

The purpose of an aggregate root is to ensure the consistency of the aggregate; it should be the only entry point for updates to the aggregate through methods or operations in the aggregate root class. You should make changes to entities within the aggregate only via the aggregate root. It is the aggregate's consistency guardian, considering all the invariants and consistency rules you might need to comply with in your aggregate. If you change a child entity or value object independently, the

aggregate root cannot ensure that the aggregate is in a valid state. It would be like a table with a loose leg. Maintaining consistency is the main purpose of the aggregate root.

In Figure 7-9, you can see sample aggregates like the buyer aggregate, which contains a single entity (the aggregate root Buyer). The order aggregate contains multiple entities and a value object.

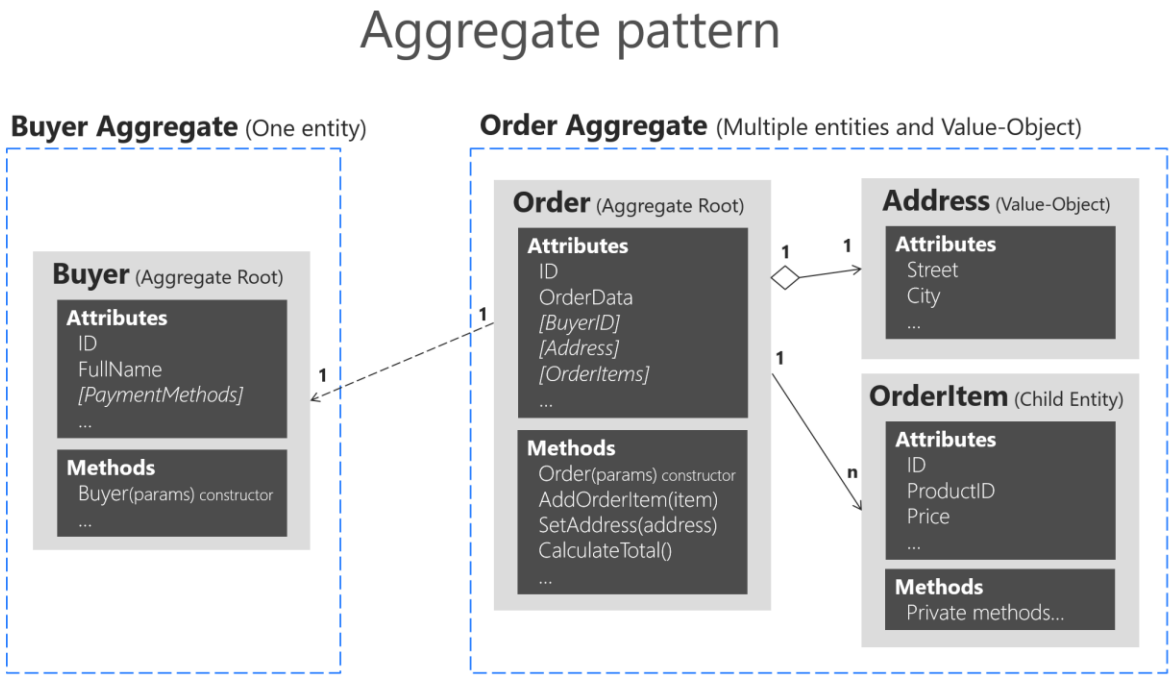


Figure 7-9. Example of aggregates with multiple or single entities

A DDD domain model is composed from aggregates, an aggregate can have just one entity or more, and can include value objects as well. Note that the Buyer aggregate could have additional child entities, depending on your domain, as it does in the ordering microservice in the eShopOnContainers reference application. Figure 7-9 just illustrates a case in which the buyer has a single entity, as an example of an aggregate that contains only an aggregate root.

In order to maintain separation of aggregates and keep clear boundaries between them, it is a good practice in a DDD domain model to disallow direct navigation between aggregates and only having the foreign key (FK) field, as implemented in the Ordering microservice domain model in eShopOnContainers. The Order entity only has a foreign key field for the buyer, but not an EF Core navigation property, as shown in the following code:

```
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
}
```

Identifying and working with aggregates requires research and experience. For more information, see the following Additional resources list.

Additional resources

- **Vaughn Vernon. Effective Aggregate Design - Part I: Modeling a Single Aggregate** (from https://dddcommunity.org/) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_1.pdf
- **Vaughn Vernon. Effective Aggregate Design - Part II: Making Aggregates Work Together** (from https://dddcommunity.org/) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Vaughn Vernon. Effective Aggregate Design - Part III: Gaining Insight Through Discovery** (from https://dddcommunity.org/) https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_3.pdf
- **Sergey Grybniak. DDD Tactical Design Patterns** https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part
- **Chris Richardson. Developing Transactional Microservices Using Aggregates** https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson
- **DevIQ. The Aggregate pattern** https://deviq.com/aggregate-pattern/

Implement a microservice domain model with .NET

In the previous section, the fundamental design principles and patterns for designing a domain model were explained. Now it's time to explore possible ways to implement the domain model by using .NET (plain C# code) and EF Core. Your domain model will be composed simply of your code. It will have just the EF Core model requirements, but not real dependencies on EF. You shouldn't have hard dependencies or references to EF Core or any other ORM in your domain model.

Domain model structure in a custom .NET Standard Library

The folder organization used for the eShopOnContainers reference application demonstrates the DDD model for the application. You might find that a different folder organization more clearly communicates the design choices made for your application. As you can see in Figure 7-10, in the ordering domain model there are two aggregates, the order aggregate and the buyer aggregate. Each aggregate is a group of domain entities and value objects, although you could have an aggregate composed of a single domain entity (the aggregate root or root entity) as well.

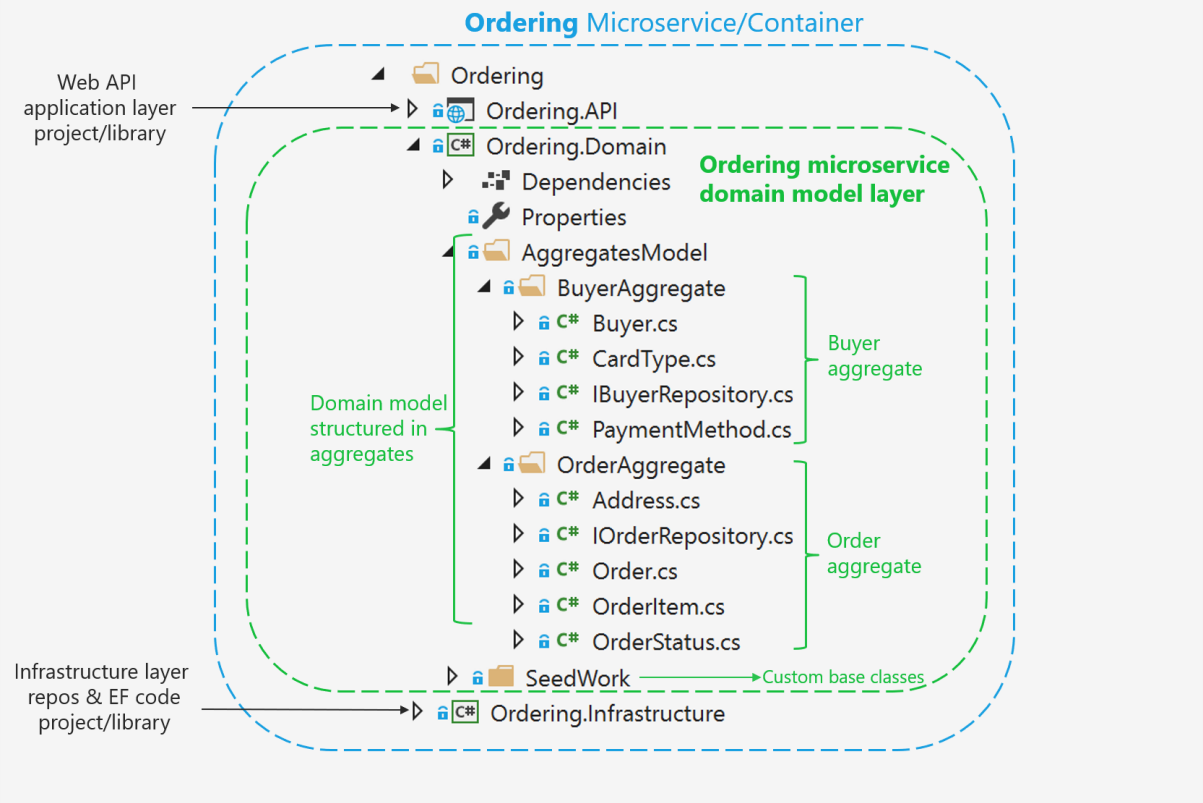


Figure 7-10. Domain model structure for the ordering microservice in eShopOnContainers

Additionally, the domain model layer includes the repository contracts (interfaces) that are the infrastructure requirements of your domain model. In other words, these interfaces express what repositories and the methods the infrastructure layer must implement. It's critical that the implementation of the repositories be placed outside of the domain model layer, in the infrastructure layer library, so the domain model layer isn't "contaminated" by API or classes from infrastructure technologies, like Entity Framework.

You can also see a SeedWork folder that contains custom base classes that you can use as a base for your domain entities and value objects, so you don't have redundant code in each domain's object class.

Structure aggregates in a custom .NET Standard library

An aggregate refers to a cluster of domain objects grouped together to match transactional consistency. Those objects could be instances of entities (one of which is the aggregate root or root entity) plus any additional value objects.

Transactional consistency means that an aggregate is guaranteed to be consistent and up to date at the end of a business action. For example, the order aggregate from the eShopOnContainers ordering microservice domain model is composed as shown in Figure 7-11.

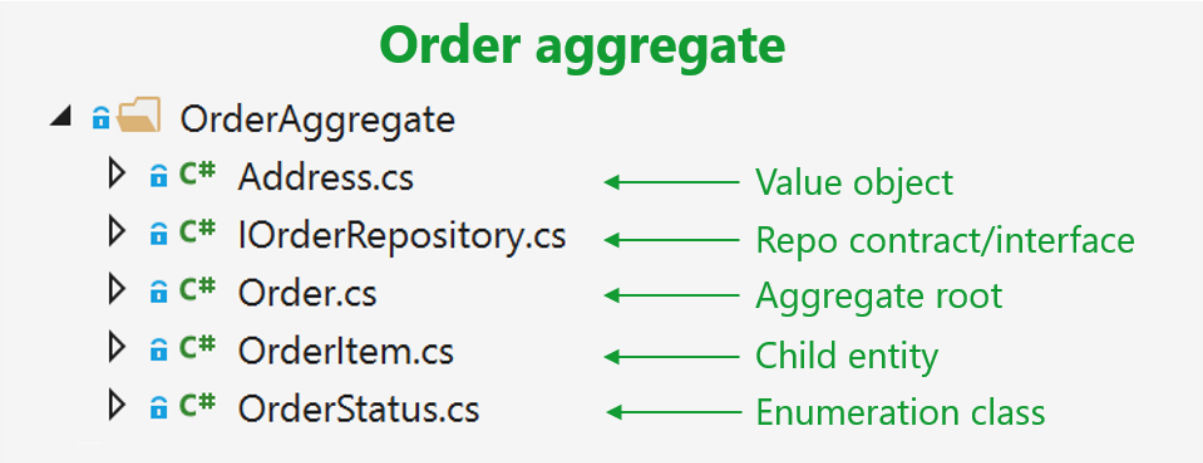


Figure 7-11. The order aggregate in Visual Studio solution

If you open any of the files in an aggregate folder, you can see how it's marked as either a custom base class or interface, like entity or value object, as implemented in the SeedWork folder.

Implement domain entities as POCO classes

You implement a domain model in .NET by creating POCO classes that implement your domain entities. In the following example, the Order class is defined as an entity and also as an aggregate root. Because the Order class derives from the Entity base class, it can reuse common code related to entities. Bear in mind that these base classes and interfaces are defined by you in the domain model project, so it is your code, not infrastructure code from an ORM like EF.

```
// COMPATIBLE WITH ENTITY FRAMEWORK CORE 5.0
// 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
    // ...
}
```

It's important to note that this is a domain entity implemented as a POCO class. It doesn't have any direct dependency on Entity Framework Core or any other infrastructure framework. This implementation is as it should be in DDD, just C# code implementing a domain model.

In addition, the class is decorated with an interface named IAggregateRoot. That interface is an empty interface, sometimes called a *marker interface*, that's used just to indicate that this entity class is also an aggregate root.

A marker interface is sometimes considered as an anti-pattern; however, it's also a clean way to mark a class, especially when that interface might be evolving. An attribute could be the other choice for the marker, but it's quicker to see the base class (Entity) next to the IAggregate interface instead of putting an Aggregate attribute marker above the class. It's a matter of preferences, in any case.

Having an aggregate root means that most of the code related to consistency and business rules of the aggregate's entities should be implemented as methods in the Order aggregate root class (for example, AddOrderItem when adding an OrderItem object to the aggregate). You should not create or update OrderItems objects independently or directly; the AggregateRoot class must keep control and consistency of any update operation against its child entities.

Encapsulate data in the Domain Entities

A common problem in entity models is that they expose collection navigation properties as publicly accessible list types. This allows any collaborator developer to manipulate the contents of these collection types, which may bypass important business rules related to the collection, possibly leaving the object in an invalid state. The solution to this is to expose read-only access to related collections and explicitly provide methods that define ways in which clients can manipulate them.

In the previous code, note that many attributes are read-only or private and are only updatable by the class methods, so any update considers business domain invariants and logic specified within the class methods.

For example, following DDD patterns, **you should *not* do the following** from any command handler method or application layer class (actually, it should be impossible for you to do so):

```
// 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);
//...
```

In this case, the Add method is purely an operation to add data, with direct access to the OrderItems collection. Therefore, most of the domain logic, rules, or validations related to that operation with the child entities will be spread across the application layer (command handlers and Web API controllers).

If you go around the aggregate root, the aggregate root cannot guarantee its invariants, its validity, or its consistency. Eventually you'll have spaghetti code or transactional script code.

To follow DDD patterns, entities must not have public setters in any entity property. Changes in an entity should be driven by explicit methods with explicit ubiquitous language about the change they're performing in the entity.

Furthermore, collections within the entity (like the order items) should be read-only properties (the AsReadOnly method explained later). You should be able to update it only from within the aggregate root class methods or the child entity methods.

As you can see in the code for the Order aggregate root, all setters should be private or at least read-only externally, so that any operation against the entity's data or its child entities has to be performed through methods in the entity class. This maintains consistency in a controlled and object-oriented way instead of implementing transactional script code.

The following code snippet shows the proper way to code the task of adding an OrderItem object to the Order aggregate.

```
// 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.

//...
```

In this snippet, most of the validations or logic related to the creation of an OrderItem object will be under the control of the Order aggregate root—in the AddOrderItem method—especially validations and logic related to other elements in the aggregate. For instance, you might get the same product item as the result of multiple calls to AddOrderItem. In that method, you could examine the product items and consolidate the same product items into a single OrderItem object with several units.

Additionally, if there are different discount amounts but the product ID is the same, you would likely apply the higher discount. This principle applies to any other domain logic for the OrderItem object.

In addition, the new OrderItem(params) operation will also be controlled and performed by the AddOrderItem method from the Order aggregate root. Therefore, most of the logic or validations related to that operation (especially anything that impacts the consistency between other child entities) will be in a single place within the aggregate root. That is the ultimate purpose of the aggregate root pattern.

When you use Entity Framework Core 1.1 or later, a DDD entity can be better expressed because it allows mapping to fields in addition to properties. This is useful when protecting collections of child entities or value objects. With this enhancement, you can use simple private fields instead of properties and you can implement any update to the field collection in public methods and provide read-only access through the AsReadOnly method.

In DDD, you want to update the entity only through methods in the entity (or the constructor) in order to control any invariant and the consistency of the data, so properties are defined only with a get accessor. The properties are backed by private fields. Private members can only be accessed from within the class. However, there is one exception: EF Core needs to set these fields as well (so it can return the object with the proper values).

Map properties with only get accessors to the fields in the database table

Mapping properties to database table columns is not a domain responsibility but part of the infrastructure and persistence layer. We mention this here just so you're aware of the new capabilities in EF Core 1.1 or later related to how you can model entities. Additional details on this topic are explained in the infrastructure and persistence section.

When you use EF Core 1.0 or later, within the DbContext you need to map the properties that are defined only with getters to the actual fields in the database table. This is done with the HasField method of the PropertyBuilder class.

Map fields without properties

With the feature in EF Core 1.1 or later to map columns to fields, it's also possible to not use properties. Instead, you can just map columns from a table to fields. A common use case for this is private fields for an internal state that doesn't need to be accessed from outside the entity.

For example, in the preceding OrderAggregate code example, there are several private fields, like the _paymentMethodId field, that have no related property for either a setter or getter. That field could also be calculated within the order's business logic and used from the order's methods, but it needs to be persisted in the database as well. So in EF Core (since v1.1), there's a way to map a field without a related property to a column in the database. This is also explained in the Infrastructure layer section of this guide.

Additional resources

- **Vaughn Vernon. Modeling Aggregates with DDD and Entity Framework.** Note that this is *not* Entity Framework Core. https://kalele.io/blog-posts/modeling-aggregates-with-ddd-and-entity-framework/
- **Julie Lerman. Data Points - Coding for Domain-Driven Design: Tips for Data-Focused Devs** https://learn.microsoft.com/archive/msdn-magazine/2013/august/data-points-coding-for-domain-driven-design-tips-for-data-focused-devs
- **Udi Dahan. How to create fully encapsulated Domain Models** https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/
- **Steve Smith. What is the difference between a DTO and a POCO?** https://ardalis.com/dto-or-poco/

Seedwork (reusable base classes and interfaces for your domain model)

The solution folder contains a *SeedWork* folder. This folder contains custom base classes that you can use as a base for your domain entities and value objects. Use these base classes so you don't have redundant code in each domain's object class. The folder for these types of classes is called *SeedWork* and not something like *Framework*. It's called *SeedWork* because the folder contains just a small subset of reusable classes that cannot really be considered a framework. *Seedwork* is a term introduced by Michael Feathers and popularized by Martin Fowler but you could also name that folder Common, SharedKernel, or similar.

Figure 7-12 shows the classes that form the seedwork of the domain model in the ordering microservice. It has a few custom base classes like Entity, ValueObject, and Enumeration, plus a few interfaces. These interfaces (IRepository and IUnitOfWork) inform the infrastructure layer about what needs to be implemented. Those interfaces are also used through Dependency Injection from the application layer.

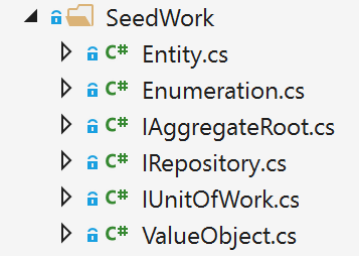


Figure 7-12. A sample set of domain model "seedwork" base classes and interfaces

This is the type of copy and paste reuse that many developers share between projects, not a formal framework. You can have seedworks in any layer or library. However, if the set of classes and interfaces gets large enough, you might want to create a single class library.

The custom Entity base class

The following code is an example of an Entity base class where you can place code that can be used the same way by any domain entity, such as the entity ID, equality operators, a domain event list per entity, 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;
        }
        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://learn.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);
    }
}
```

The previous code using a domain event list per entity will be explained in the next sections when focusing on domain events.

Repository contracts (interfaces) in the domain model layer

Repository contracts are simply .NET interfaces that express the contract requirements of the repositories to be used for each aggregate.

The repositories themselves, with EF Core code or any other infrastructure dependencies and code (Linq, SQL, etc.), must not be implemented within the domain model; the repositories should only implement the interfaces you define in the domain model.

A pattern related to this practice (placing the repository interfaces in the domain model layer) is the Separated Interface pattern. As explained by Martin Fowler, "Use Separated Interface to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation."

Following the Separated Interface pattern enables the application layer (in this case, the Web API project for the microservice) to have a dependency on the requirements defined in the domain model, but not a direct dependency to the infrastructure/persistence layer. In addition, you can use Dependency Injection to isolate the implementation, which is implemented in the infrastructure/persistence layer using repositories.

For example, the following example with the IOrderRepository interface defines what operations the OrderRepository class will need to implement at the infrastructure layer. In the current implementation of the application, the code just needs to add or update orders to the database, since queries are split following the simplified CQRS approach.

```
// 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; }
}
```

Additional resources

- **Martin Fowler. Separated Interface.** https://www.martinfowler.com/eaaCatalog/separatedInterface.html

Implement value objects

As discussed in earlier sections about entities and aggregates, identity is fundamental for entities. However, there are many objects and data items in a system that do not require an identity and identity tracking, such as value objects.

A value object can reference other entities. For example, in an application that generates a route that describes how to get from one point to another, that route would be a value object. It would be a snapshot of points on a specific route, but this suggested route would not have an identity, even though internally it might refer to entities like City, Road, etc.

Figure 7-13 shows the Address value object within the Order aggregate.

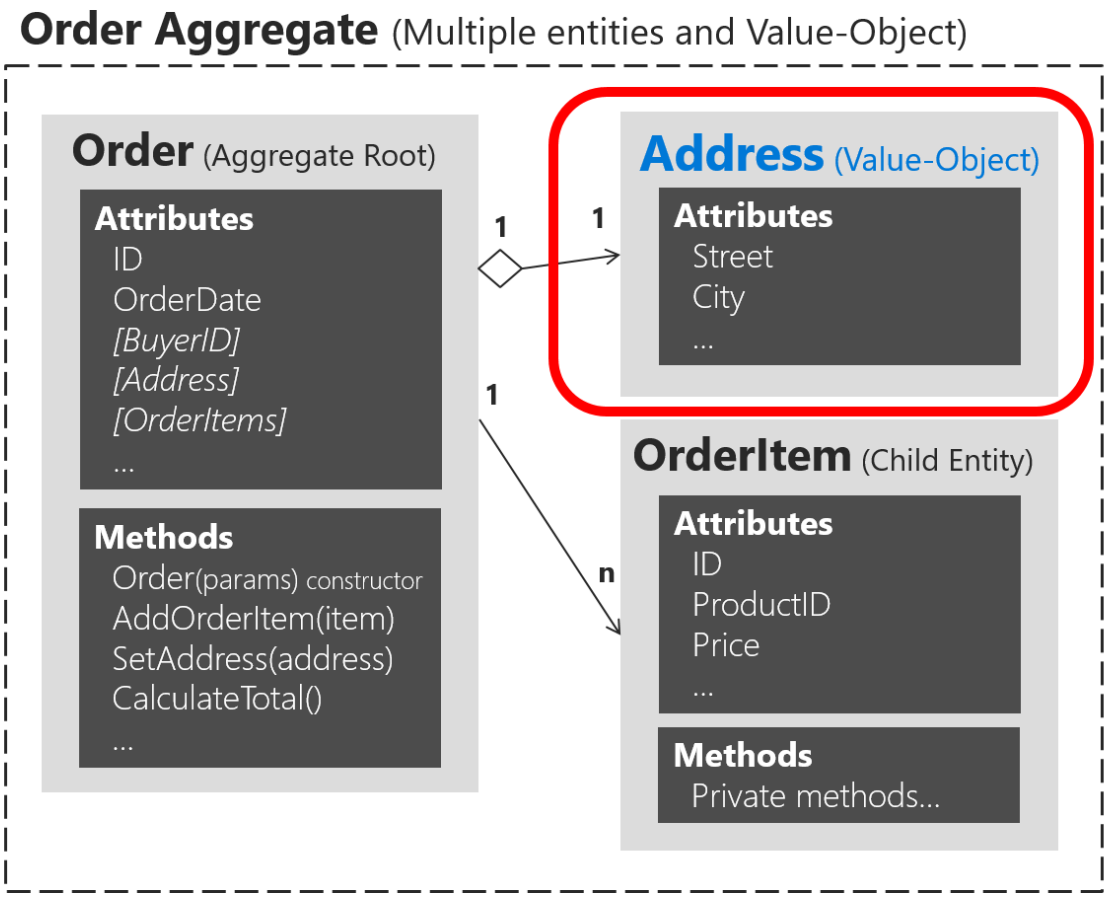


Figure 7-13. Address value object within the Order aggregate

As shown in Figure 7-13, an entity is usually composed of multiple attributes. For example, the Order entity can be modeled as an entity with an identity and composed internally of a set of attributes such as OrderId, OrderDate, OrderItems, etc. But the address, which is simply a complex-value composed of country/region, street, city, etc., and has no identity in this domain, must be modeled and treated as a value object.

Important characteristics of value objects

There are two main characteristics for value objects:

- They have no identity.
- They are immutable.

The first characteristic was already discussed. Immutability is an important requirement. The values of a value object must be immutable once the object is created. Therefore, when the object is

constructed, you must provide the required values, but you must not allow them to change during the object's lifetime.

Value objects allow you to perform certain tricks for performance, thanks to their immutable nature. This is especially true in systems where there may be thousands of value object instances, many of which have the same values. Their immutable nature allows them to be reused; they can be interchangeable objects, since their values are the same and they have no identity. This type of optimization can sometimes make a difference between software that runs slowly and software with good performance. Of course, all these cases depend on the application environment and deployment context.

Value object implementation in C#

In terms of implementation, you can have a value object base class that has basic utility methods like equality based on the comparison between all the attributes (since a value object must not be based on identity) and other fundamental characteristics. The following example shows a value object base class used in the ordering microservice from 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, right) || 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
}
```

The ValueObject is an abstract class type, but in this example, it doesn't overload the == and != operators. You could choose to do so, making comparisons delegate to the Equals override. For example, consider the following operator overloads to the ValueObject type:

```
public static bool operator ==(ValueObject one, ValueObject two)
{
    return EqualOperator(one, two);
}

public static bool operator !=(ValueObject one, ValueObject two)
{
    return NotEqualOperator(one, two);
}
You can use this class when implementing your actual value object, as with the Address
value object shown in the following example:
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;
    }
}
```

This value object implementation of Address has no identity, and therefore no ID field is defined for it, either in the Address class definition or the ValueObject class definition.

Having no ID field in a class to be used by Entity Framework (EF) was not possible until EF Core 2.0, which greatly helps to implement better value objects with no ID. That is precisely the explanation of the next section.

It could be argued that value objects, being immutable, should be read-only (that is, have get-only properties), and that's indeed true. However, value objects are usually serialized and deserialized to go

through message queues, and being read-only stops the deserializer from assigning values, so you just leave them as private set, which is read-only enough to be practical.

Value object comparison semantics

Two instances of the Address type can be compared using all the following methods:

```
var one = new Address("1 Microsoft Way", "Redmond", "WA", "US", "98052");
var two = new Address("1 Microsoft Way", "Redmond", "WA", "US", "98052");

Console.WriteLine(EqualityComparer<Address>.Default.Equals(one, two)); // True
Console.WriteLine(object.Equals(one, two)); // True
Console.WriteLine(one.Equals(two)); // True
Console.WriteLine(one == two); // True
```

When all the values are the same, the comparisons are correctly evaluated as true. If you didn't choose to overload the == and != operators, then the last comparison of one == two would evaluate as false. For more information, see Overload ValueObject equality operators.

How to persist value objects in the database with EF Core 2.0 and later

You just saw how to define a value object in your domain model. But how can you actually persist it into the database using Entity Framework Core since it usually targets entities with identity?

Background and older approaches using EF Core 1.1

As background, a limitation when using EF Core 1.0 and 1.1 was that you could not use complex types as defined in EF 6.x in the traditional .NET Framework. Therefore, if using EF Core 1.0 or 1.1, you needed to store your value object as an EF entity with an ID field. Then, so it looked more like a value object with no identity, you could hide its ID so you make clear that the identity of a value object is not important in the domain model. You could hide that ID by using the ID as a shadow property. Since that configuration for hiding the ID in the model is set up in the EF infrastructure level, it would be kind of transparent for your domain model.

In the initial version of eShopOnContainers (.NET Core 1.1), the hidden ID needed by EF Core infrastructure was implemented in the following way in the DbContext level, using Fluent API at the infrastructure project. Therefore, the ID was hidden from the domain model point of view, but still present in the infrastructure.

```
// 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
}
```

However, the persistence of that value object into the database was performed like a regular entity in a different table.

With EF Core 2.0 and later, there are new and better ways to persist value objects.

Persist value objects as owned entity types in EF Core 2.0 and later

Even with some gaps between the canonical value object pattern in DDD and the owned entity type in EF Core, it's currently the best way to persist value objects with EF Core 2.0 and later. You can see limitations at the end of this section.

The owned entity type feature was added to EF Core since version 2.0.

An owned entity type allows you to map types that do not have their own identity explicitly defined in the domain model and are used as properties, such as a value object, within any of your entities. An owned entity type shares the same CLR type with another entity type (that is, it's just a regular class). The entity containing the defining navigation is the owner entity. When querying the owner, the owned types are included by default.

Just by looking at the domain model, an owned type looks like it doesn't have any identity. However, under the covers, owned types do have the identity, but the owner navigation property is part of this identity.

The identity of instances of owned types is not completely their own. It consists of three components:

- The identity of the owner
- The navigation property pointing to them
- In the case of collections of owned types, an independent component (supported in EF Core 2.2 and later).

For example, in the Ordering domain model at eShopOnContainers, as part of the Order entity, the Address value object is implemented as an owned entity type within the owner entity, which is the Order entity. Address is a type with no identity property defined in the domain model. It is used as a property of the Order type to specify the shipping address for a particular order.

By convention, a shadow primary key is created for the owned type and it will be mapped to the same table as the owner by using table splitting. This allows to use owned types similarly to how complex types are used in EF6 in the traditional .NET Framework.

It is important to note that owned types are never discovered by convention in EF Core, so you have to declare them explicitly.

In eShopOnContainers, in the OrderingContext.cs file, within the OnModelCreating() method, multiple infrastructure configurations are applied. One of them is related to the Order entity.

```
// 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
}
```

In the following code, the persistence infrastructure is defined for the Order entity:

```
// 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...
    //...
}
```

In the previous code, the orderConfiguration.OwnsOne(o => o.Address) method specifies that the Address property is an owned entity of the Order type.

By default, EF Core conventions name the database columns for the properties of the owned entity type as EntityProperty_OwnedEntityProperty. Therefore, the internal properties of Address will appear in the Orders table with the names Address_Street, Address_City (and so on for State, Country, and ZipCode).

You can append the Property().HasColumnName() fluent method to rename those columns. In the case where Address is a public property, the mappings would be like the following:

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

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

It's possible to chain the OwnsOne method in a fluent mapping. In the following hypothetical example, OrderDetails owns BillingAddress and ShippingAddress, which are both Address types. Then OrderDetails is owned by the Order type.

```
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; }
}
```

Additional details on owned entity types

- Owned types are defined when you configure a navigation property to a particular type using the OwnsOne fluent API.
- The definition of an owned type in our metadata model is a composite of: the owner type, the navigation property, and the CLR type of the owned type.
- The identity (key) of an owned type instance in our stack is a composite of the identity of the owner type and the definition of the owned type.

Owned entities capabilities

- Owned types can reference other entities, either owned (nested owned types) or non-owned (regular reference navigation properties to other entities).
- You can map the same CLR type as different owned types in the same owner entity through separate navigation properties.
- Table splitting is set up by convention, but you can opt out by mapping the owned type to a different table using ToTable.
- Eager loading is performed automatically on owned types, that is, there's no need to call .Include() on the query.
- Can be configured with attribute [Owned], using EF Core 2.1 and later.
- Can handle collections of owned types (using version 2.2 and later).

Owned entities limitations

- You can't create a DbSet<T> of an owned type (by design).
- You can't call ModelBuilder.Entity<T>() on owned types (currently by design).
- No support for optional (that is, nullable) owned types that are mapped with the owner in the same table (that is, using table splitting). This is because mapping is done for each property, there is no separate sentinel for the null complex value as a whole.
- No inheritance-mapping support for owned types, but you should be able to map two leaf types of the same inheritance hierarchies as different owned types. EF Core will not reason about the fact that they are part of the same hierarchy.

Main differences with EF6's complex types

- Table splitting is optional, that is, they can optionally be mapped to a separate table and still be owned types.

Additional resources

- **Martin Fowler. ValueObject pattern**
  https://martinfowler.com/bliki/ValueObject.html
- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software.** (Book; includes a discussion of value objects)
  https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/
- **Vaughn Vernon. Implementing Domain-Driven Design.** (Book; includes a discussion of value objects)
  https://www.amazon.com/Implementing-Domain-Driven-Design-Vaughn-Vernon/dp/0321834577/
- **Owned Entity Types**
  https://learn.microsoft.com/ef/core/modeling/owned-entities
- **Shadow Properties**
  https://learn.microsoft.com/ef/core/modeling/shadow-properties
- **Complex types and/or value objects**. Discussion in the EF Core GitHub repo (Issues tab)
  https://github.com/dotnet/efcore/issues/246
- **ValueObject.cs.** Base value object class in eShopOnContainers.
  https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs
- **ValueObject.cs.** Base value object class in CSharpFunctionalExtensions.
  https://github.com/vkhorikov/CSharpFunctionalExtensions/blob/master/CSharpFunctionalExtensions/ValueObject/ValueObject.cs
- **Address class.** Sample value object class in eShopOnContainers.
  https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Address.cs

```
Use enumeration classes instead of enum types
```

Enumerations (or *enum types* for short) are a thin language wrapper around an integral type. You might want to limit their use to when you are storing one value from a closed set of values. Classification based on sizes (small, medium, large) is a good example. Using enums for control flow or more robust abstractions can be a code smell. This type of usage leads to fragile code with many control flow statements checking values of the enum.

Instead, you can create Enumeration classes that enable all the rich features of an object-oriented language.

However, this isn't a critical topic and in many cases, for simplicity, you can still use regular enum types if that's your preference. The use of enumeration classes is more related to business-related concepts.

Implement an Enumeration base class

The ordering microservice in eShopOnContainers provides a sample Enumeration base class implementation, as shown in the following example:

```
public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration(int id, string name) => (Id, Name) = (id, name);

    public override string ToString() => Name;

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

    public override bool Equals(object obj)
    {
        if (obj is not Enumeration otherValue)
        {
            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 ...
}
You can use this class as a type in any entity or value object, as for the following
CardType : Enumeration class:
public class CardType
    : Enumeration
{
    public static CardType Amex = new(1, nameof(Amex));
    public static CardType Visa = new(2, nameof(Visa));
    public static CardType MasterCard = new(3, nameof(MasterCard));

    public CardType(int id, string name)
        : base(id, name)
    {
```

```
    }
}
```

Additional resources

- **Jimmy Bogard. Enumeration classes**
  https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/
- **Steve Smith. Enum Alternatives in C#**
  https://ardalis.com/enum-alternatives-in-c
- **Enumeration.cs.** Base Enumeration class in eShopOnContainers
  https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/SeedWork/Enumeration.cs
- **CardType.cs**. Sample Enumeration class in eShopOnContainers.
  https://github.com/dotnet-architecture/eShopOnContainers/blob/dev/src/Services/Ordering/Ordering.Domain/AggregatesModel/BuyerAggregate/CardType.cs
- **SmartEnum**. Ardalis - Classes to help produce strongly typed smarter enums in .NET.
  https://www.nuget.org/packages/Ardalis.SmartEnum/

Design validations in the domain model layer

In DDD, validation rules can be thought as invariants. The main responsibility of an aggregate is to enforce invariants across state changes for all the entities within that aggregate.

Domain entities should always be valid entities. There are a certain number of invariants for an object that should always be true. For example, an order item object always has to have a quantity that must be a positive integer, plus an article name and price. Therefore, invariants enforcement is the responsibility of the domain entities (especially of the aggregate root) and an entity object should not be able to exist without being valid. Invariant rules are simply expressed as contracts, and exceptions or notifications are raised when they are violated.

The reasoning behind this is that many bugs occur because objects are in a state they should never have been in.

Let's propose we now have a SendUserCreationEmailService that takes a UserProfile … how can we rationalize in that service that Name is not null? Do we check it again? Or more likely … you just don't bother to check and "hope for the best"—you hope that someone bothered to validate it before sending it to you. Of course, using TDD one of the first tests we should be writing is that if I send a customer with a null name that it should raise an error. But once we start writing these kinds of tests over and over again we realize … "what if we never allowed name to become null? we wouldn't have all of these tests!".

Implement validations in the domain model layer

Validations are usually implemented in domain entity constructors or in methods that can update the entity. There are multiple ways to implement validations, such as verifying data and raising exceptions if the validation fails. There are also more advanced patterns such as using the Specification pattern for validations, and the Notification pattern to return a collection of errors instead of returning an exception for each validation as it occurs.

Validate conditions and throw exceptions

The following code example shows the simplest approach to validation in a domain entity by raising an exception. In the references table at the end of this section you can see links to more advanced implementations based on the patterns we have discussed previously.

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
A better example would demonstrate the need to ensure that either the internal state did
not change, or that all the mutations for a method occurred. For example, the following
implementation would leave the object in an invalid state:
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 …);
}
```

If the value of the state is invalid, the first address line and the city have already been changed. That might make the address invalid.

A similar approach can be used in the entity's constructor, raising an exception to make sure that the entity is valid once it is created.

Use validation attributes in the model based on data annotations

Data annotations, like the Required or MaxLength attributes, can be used to configure EF Core database field properties, as explained in detail in the Table mapping section, but they no longer work for entity validation in EF Core (neither does the IValidatableObject.Validate method), as they have done since EF 4.x in .NET Framework.

Data annotations and the IValidatableObject interface can still be used for model validation during model binding, prior to the controller's actions invocation as usual, but that model is meant to be a ViewModel or DTO and that's an MVC or API concern not a domain model concern.

Having made the conceptual difference clear, you can still use data annotations and IValidatableObject in the entity class for validation, if your actions receive an entity class object parameter, which is not recommended. In that case, validation will occur upon model binding, just before invoking the action and you can check the controller's ModelState.IsValid property to check

the result, but then again, it happens in the controller, not before persisting the entity object in the DbContext, as it had done since EF 4.x.

You can still implement custom validation in the entity class using data annotations and the IValidatableObject.Validate method, by overriding the DbContext's SaveChanges method.

You can see a sample implementation for validating IValidatableObject entities in this comment on GitHub. That sample doesn't do attribute-based validations, but they should be easy to implement using reflection in the same override.

However, from a DDD point of view, the domain model is best kept lean with the use of exceptions in your entity's behavior methods, or by implementing the Specification and Notification patterns to enforce validation rules.

It can make sense to use data annotations at the application layer in ViewModel classes (instead of domain entities) that will accept input, to allow for model validation within the UI layer. However, this should not be done at the exclusion of validation within the domain model.

Validate entities by implementing the Specification pattern and the Notification pattern

Finally, a more elaborate approach to implementing validations in the domain model is by implementing the Specification pattern in conjunction with the Notification pattern, as explained in some of the additional resources listed later.

It is worth mentioning that you can also use just one of those patterns—for example, validating manually with control statements, but using the Notification pattern to stack and return a list of validation errors.

Use deferred validation in the domain

There are various approaches to deal with deferred validations in the domain. In his book Implementing Domain-Driven Design, Vaughn Vernon discusses these in the section on validation.

Two-step validation

Also consider two-step validation. Use field-level validation on your command Data Transfer Objects (DTOs) and domain-level validation inside your entities. You can do this by returning a result object instead of exceptions in order to make it easier to deal with the validation errors.

Using field validation with data annotations, for example, you do not duplicate the validation definition. The execution, though, can be both server-side and client-side in the case of DTOs (commands and ViewModels, for instance).

Additional resources

- **Rachel Appel. Introduction to model validation in ASP.NET Core MVC**
  https://learn.microsoft.com/aspnet/core/mvc/models/validation
- **Rick Anderson. Adding validation**
  https://learn.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation

- **Martin Fowler. Replacing Throwing Exceptions with Notification in Validations**
  https://martinfowler.com/articles/replaceThrowWithNotification.html
- **Specification and Notification Patterns**
  https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns
- **Lev Gorodinski. Validation in Domain-Driven Design (DDD)**
  http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/
- **Colin Jack. Domain Model Validation**
  https://colinjack.blogspot.com/2008/03/domain-model-validation.html
- **Jimmy Bogard. Validation in a DDD world**
  https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/

Client-side validation (validation in the presentation layers)

Even when the source of truth is the domain model and ultimately you must have validation at the domain model level, validation can still be handled at both the domain model level (server side) and the UI (client side).

Client-side validation is a great convenience for users. It saves time they would otherwise spend waiting for a round trip to the server that might return validation errors. In business terms, even a few fractions of seconds multiplied hundreds of times each day adds up to a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

Just as the view model and the domain model are different, view model validation and domain model validation might be similar but serve a different purpose. If you are concerned about DRY (the Don't Repeat Yourself principle), consider that in this case code reuse might also mean coupling, and in enterprise applications it is more important not to couple the server side to the client side than to follow the DRY principle.

Even when using client-side validation, you should always validate your commands or input DTOs in server code, because the server APIs are a possible attack vector. Usually, doing both is your best bet because if you have a client application, from a UX perspective, it is best to be proactive and not allow the user to enter invalid information.

Therefore, in client-side code you typically validate the ViewModels. You could also validate the client output DTOs or commands before you send them to the services.

The implementation of client-side validation depends on what kind of client application you are building. It will be different if you are validating data in a web MVC web application with most of the code in .NET, a SPA web application with that validation being coded in JavaScript or TypeScript, or a mobile app coded with Xamarin and C#.

Additional resources

Validation in Xamarin mobile apps

- **Validate Text Input and Show Errors**
  https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/
- **Validation Callback**
  https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/

Validation in ASP.NET Core apps

- **Rick Anderson. Adding validation**
  https://learn.microsoft.com/aspnet/core/tutorials/first-mvc-app/validation

Validation in SPA Web apps (Angular 2, TypeScript, JavaScript, Blazor WebAssembly)

- **Form Validation**
  https://angular.io/guide/form-validation
- **Validation.** Breeze documentation.
  https://breeze.github.io/doc-js/validation.html
- **ASP.NET Core Blazor forms and input components**

In summary, these are the most important concepts in regards to validation:

- Entities and aggregates should enforce their own consistency and be "always valid". Aggregate roots are responsible for multi-entity consistency within the same aggregate.
- If you think that an entity needs to enter an invalid state, consider using a different object model—for example, using a temporary DTO until you create the final domain entity.
- If you need to create several related objects, such as an aggregate, and they are only valid once all of them have been created, consider using the Factory pattern.
- In most of the cases, having redundant validation in the client side is good, because the application can be proactive.

Domain events: Design and implementation

Use domain events to explicitly implement side effects of changes within your domain. In other words, and using DDD terminology, use domain events to explicitly implement side effects across multiple aggregates. Optionally, for better scalability and less impact in database locks, use eventual consistency between aggregates within the same domain.

What is a domain event?

An event is something that has happened in the past. A domain event is, something that happened in the domain that you want other parts of the same domain (in-process) to be aware of. The notified parts usually react somehow to the events.

An important benefit of domain events is that side effects can be expressed explicitly.

For example, if you're just using Entity Framework and there has to be a reaction to some event, you would probably code whatever you need close to what triggers the event. So the rule gets coupled, implicitly, to the code, and you have to look into the code to, hopefully, realize the rule is implemented there.

On the other hand, using domain events makes the concept explicit, because there's a DomainEvent and at least one DomainEventHandler involved.

For example, in the eShopOnContainers application, when an order is created, the user becomes a buyer, so an OrderStartedDomainEvent is raised and handled in the ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler, so the underlying concept is evident.

In short, domain events help you to express, explicitly, the domain rules, based in the ubiquitous language provided by the domain experts. Domain events also enable a better separation of concerns among classes within the same domain.

It's important to ensure that, just like a database transaction, either all the operations related to a domain event finish successfully or none of them do.

Domain events are similar to messaging-style events, with one important difference. With real messaging, message queuing, message brokers, or a service bus using AMQP, a message is always sent asynchronously and communicated across processes and machines. This is useful for integrating multiple Bounded Contexts, microservices, or even different applications. However, with domain events, you want to raise an event from the domain operation you're currently running, but you want any side effects to occur within the same domain.

The domain events and their side effects (the actions triggered afterwards that are managed by event handlers) should occur almost immediately, usually in-process, and within the same domain. Thus, domain events could be synchronous or asynchronous. Integration events, however, should always be asynchronous.

Domain events versus integration events

Semantically, domain and integration events are the same thing: notifications about something that just happened. However, their implementation must be different. Domain events are just messages pushed to a domain event dispatcher, which could be implemented as an in-memory mediator based on an IoC container or any other method.

On the other hand, the purpose of integration events is to propagate committed transactions and updates to additional subsystems, whether they are other microservices, Bounded Contexts or even

external applications. Hence, they should occur only if the entity is successfully persisted, otherwise it's as if the entire operation never happened.

As mentioned before, integration events must be based on asynchronous communication between multiple microservices (other Bounded Contexts) or even external systems/applications.

Thus, the event bus interface needs some infrastructure that allows inter-process and distributed communication between potentially remote services. It can be based on a commercial service bus, queues, a shared database used as a mailbox, or any other distributed and ideally push based messaging system.

Domain events as a preferred way to trigger side effects across multiple aggregates within the same domain

If executing a command related to one aggregate instance requires additional domain rules to be run on one or more additional aggregates, you should design and implement those side effects to be triggered by domain events. As shown in Figure 7-14, and as one of the most important use cases, a domain event should be used to propagate state changes across multiple aggregates within the same domain model.

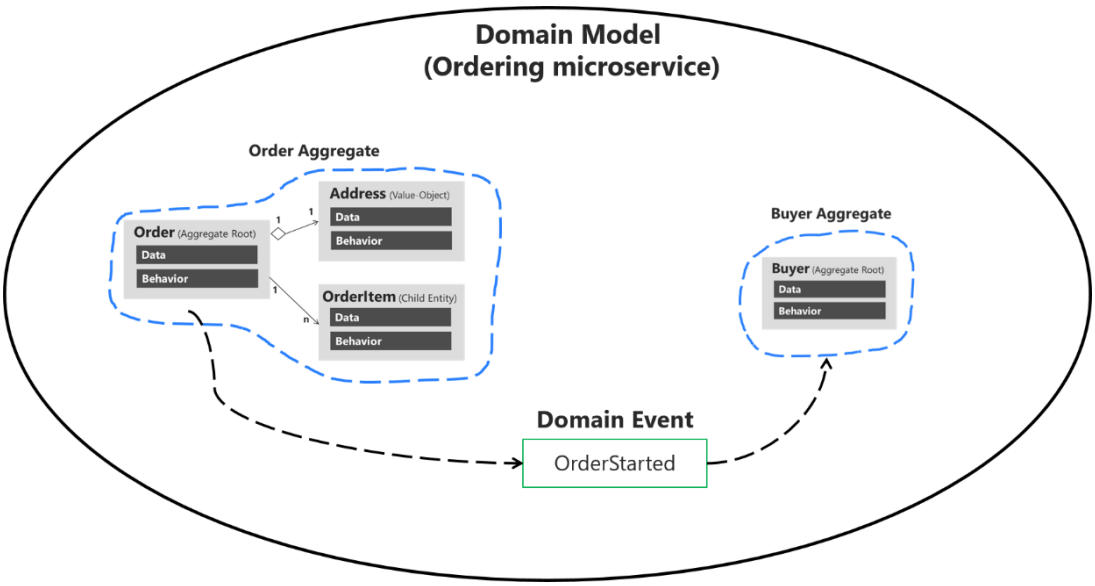


Figure 7-14. Domain events to enforce consistency between multiple aggregates within the same domain

Figure 7-14 shows how consistency between aggregates is achieved by domain events. When the user initiates an order, the Order Aggregate sends an OrderStarted domain event. The OrderStarted domain event is handled by the Buyer Aggregate to create a Buyer object in the ordering microservice, based on the original user info from the identity microservice (with information provided in the CreateOrder command).

Alternately, you can have the aggregate root subscribed for events raised by members of its aggregates (child entities). For instance, each OrderItem child entity can raise an event when the item price is higher than a specific amount, or when the product item amount is too high. The aggregate root can then receive those events and perform a global calculation or aggregation.

It's important to understand that this event-based communication is not implemented directly within the aggregates; you need to implement domain event handlers.

Handling the domain events is an application concern. The domain model layer should only focus on the domain logic—things that a domain expert would understand, not application infrastructure like handlers and side-effect persistence actions using repositories. Therefore, the application layer level is where you should have domain event handlers triggering actions when a domain event is raised.

Domain events can also be used to trigger any number of application actions, and what is more important, must be open to increase that number in the future in a decoupled way. For instance, when the order is started, you might want to publish a domain event to propagate that info to other aggregates or even to raise application actions like notifications.

The key point is the open number of actions to be executed when a domain event occurs. Eventually, the actions and rules in the domain and application will grow. The complexity or number of side-effect actions when something happens will grow, but if your code were coupled with "glue" (that is, creating specific objects with new), then every time you needed to add a new action you would also need to change working and tested code.

This change could result in new bugs and this approach also goes against the Open/Closed principle from SOLID. Not only that, the original class that was orchestrating the operations would grow and grow, which goes against the Single Responsibility Principle (SRP).

On the other hand, if you use domain events, you can create a fine-grained and decoupled implementation by segregating responsibilities using this approach:

1. Send a command (for example, CreateOrder).
2. Receive the command in a command handler.
   - Execute a single aggregate's transaction.
   - (Optional) Raise domain events for side effects (for example, OrderStartedDomainEvent).
3. Handle domain events (within the current process) that will execute an open number of side effects in multiple aggregates or application actions. For example:
   - Verify or create buyer and payment method.
   - Create and send a related integration event to the event bus to propagate states across microservices or trigger external actions like sending an email to the buyer.
   - Handle other side effects.

As shown in Figure 7-15, starting from the same domain event, you can handle multiple actions related to other aggregates in the domain or additional application actions you need to perform across microservices connecting with integration events and the event bus.

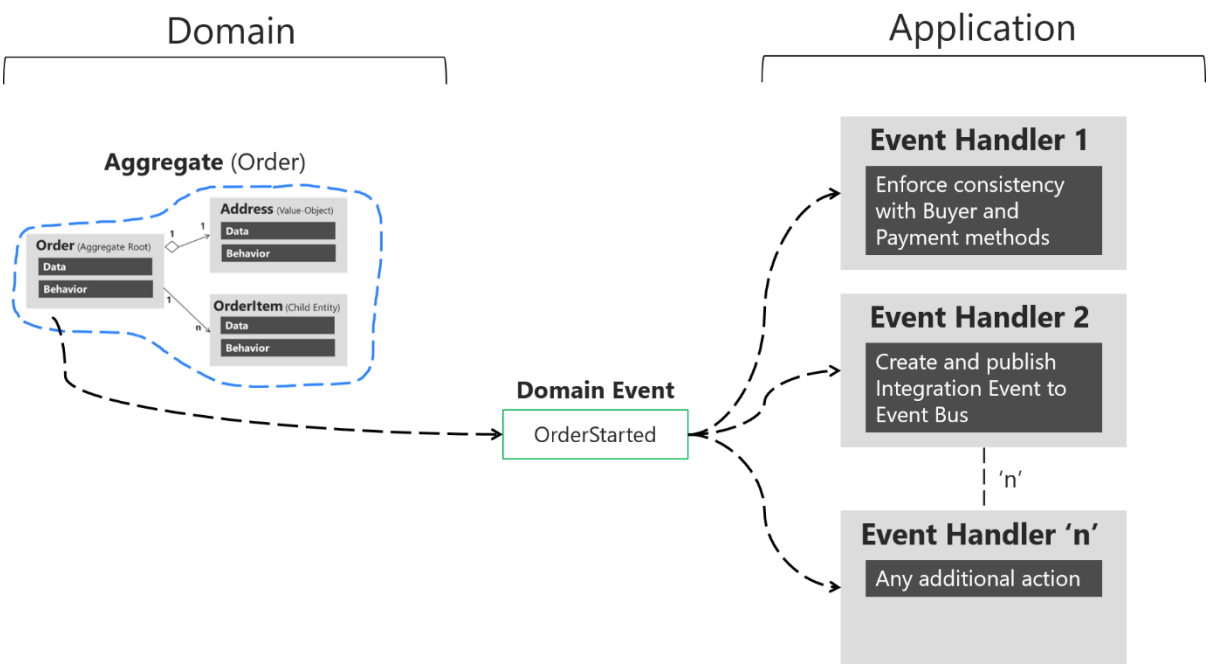


Figure 7-15. Handling multiple actions per domain

There can be several handlers for the same domain event in the Application Layer, one handler can solve consistency between aggregates and another handler can publish an integration event, so other microservices can do something with it. The event handlers are typically in the application layer, because you'll use infrastructure objects like repositories or an application API for the microservice's behavior. In that sense, event handlers are similar to command handlers, so both are part of the application layer. The important difference is that a command should be processed only once. A domain event could be processed zero or *n* times, because it can be received by multiple receivers or event handlers with a different purpose for each handler.

Having an open number of handlers per domain event allows you to add as many domain rules as needed, without affecting current code. For instance, implementing the following business rule might be as easy as adding a few event handlers (or even just one):

When the total amount purchased by a customer in the store, across any number of orders, exceeds $6,000, apply a 10% off discount to every new order and notify the customer with an email about that discount for future orders.

Implement domain events

In C#, a domain event is simply a data-holding structure or class, like a DTO, with all the information related to what just happened in the domain, as shown in the following example:

```
public class OrderStartedDomainEvent : INotification
{
    public string UserId { get; }
    public string UserName { 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, string userId, string userName,
                                   int cardTypeId, string cardNumber,
                                   string cardSecurityNumber, string cardHolderName,
                                   DateTime cardExpiration)
    {
        Order = order;
        UserId = userId;
        UserName = userName;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}
```

This is essentially a class that holds all the data related to the OrderStarted event.

In terms of the ubiquitous language of the domain, since an event is something that happened in the past, the class name of the event should be represented as a past-tense verb, like OrderStartedDomainEvent or OrderShippedDomainEvent. That's how the domain event is implemented in the ordering microservice in eShopOnContainers.

As noted earlier, an important characteristic of events is that since an event is something that happened in the past, it shouldn't change. Therefore, it must be an immutable class. You can see in the previous code that the properties are read-only. There's no way to update the object, you can only set values when you create it.

It's important to highlight here that if domain events were to be handled asynchronously, using a queue that required serializing and deserializing the event objects, the properties would have to be "private set" instead of read-only, so the deserializer would be able to assign the values upon dequeuing. This is not an issue in the Ordering microservice, as the domain event pub/sub is implemented synchronously using MediatR.

Raise domain events

The next question is how to raise a domain event so it reaches its related event handlers. You can use multiple approaches.

Udi Dahan originally proposed (for example, in several related posts, such as Domain Events – Take 2) using a static class for managing and raising the events. This might include a static class named DomainEvents that would raise domain events immediately when it's called, using syntax like DomainEvents.Raise(Event myEvent). Jimmy Bogard wrote a blog post (Strengthening your domain: Domain Events) that recommends a similar approach.

However, when the domain events class is static, it also dispatches to handlers immediately. This makes testing and debugging more difficult, because the event handlers with side-effects logic are executed immediately after the event is raised. When you're testing and debugging, you just want to focus on what is happening in the current aggregate classes; you don't want to suddenly be

redirected to other event handlers for side effects related to other aggregates or application logic. This is why other approaches have evolved, as explained in the next section.

The deferred approach to raise and dispatch events

Instead of dispatching to a domain event handler immediately, a better approach is to add the domain events to a collection and then to dispatch those domain events *right before* or *right after* committing the transaction (as with SaveChanges in EF). (This approach was described by Jimmy Bogard in this post A better domain events pattern.)

Deciding if you send the domain events right before or right after committing the transaction is important, since it determines whether you will include the side effects as part of the same transaction or in different transactions. In the latter case, you need to deal with eventual consistency across multiple aggregates. This topic is discussed in the next section.

The deferred approach is what eShopOnContainers uses. First, you add the events happening in your entities into a collection or list of events per entity. That list should be part of the entity object, or even better, part of your base entity class, as shown in the following example of the Entity base class:

```
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
}
```

When you want to raise an event, you just add it to the event collection from code at any method of the aggregate-root entity.

The following code, part of the Order aggregate-root at eShopOnContainers, shows an example:

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

Notice that the only thing that the AddDomainEvent method is doing is adding an event to the list. No event is dispatched yet, and no event handler is invoked yet.

You actually want to dispatch the events later on, when you commit the transaction to the database. If you are using Entity Framework Core, that means in the SaveChanges method of your EF DbContext, as in the following code:

```
// 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();
    }
}
```

With this code, you dispatch the entity events to their respective event handlers.

The overall result is that you've decoupled the raising of a domain event (a simple add into a list in memory) from dispatching it to an event handler. In addition, depending on what kind of dispatcher you are using, you could dispatch the events synchronously or asynchronously.

Be aware that transactional boundaries come into significant play here. If your unit of work and transaction can span more than one aggregate (as when using EF Core and a relational database), this can work well. But if the transaction cannot span aggregates, you have to implement additional steps to achieve consistency. This is another reason why persistence ignorance is not universal; it depends on the storage system you use.

Single transaction across aggregates versus eventual consistency across aggregates

The question of whether to perform a single transaction across aggregates versus relying on eventual consistency across those aggregates is a controversial one. Many DDD authors like Eric Evans and Vaughn Vernon advocate the rule that one transaction = one aggregate and therefore argue for eventual consistency across aggregates. For example, in his book *Domain-Driven Design*, Eric Evans says this:

Any rule that spans Aggregates will not be expected to be up-to-date at all times. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specific time. (page 128)

Vaughn Vernon says the following in [Effective Aggregate Design. Part II: Making Aggregates Work Together]:

Thus, if executing a command on one aggregate instance requires that additional business rules execute on one or more aggregates, use eventual consistency [...] There is a practical way to support eventual consistency in a DDD model. An aggregate method publishes a domain event that is in time delivered to one or more asynchronous subscribers.

This rationale is based on embracing fine-grained transactions instead of transactions spanning many aggregates or entities. The idea is that in the second case, the number of database locks will be substantial in large-scale applications with high scalability needs. Embracing the fact that highly scalable applications need not have instant transactional consistency between multiple aggregates helps with accepting the concept of eventual consistency. Atomic changes are often not needed by the business, and it is in any case the responsibility of the domain experts to say whether particular operations need atomic transactions or not. If an operation always needs an atomic transaction between multiple aggregates, you might ask whether your aggregate should be larger or wasn't correctly designed.

However, other developers and architects like Jimmy Bogard are okay with spanning a single transaction across several aggregates—but only when those additional aggregates are related to side effects for the same original command. For instance, in A better domain events pattern, Bogard says this:

Typically, I want the side effects of a domain event to occur within the same logical transaction, but not necessarily in the same scope of raising the domain event [...] Just before we commit our transaction, we dispatch our events to their respective handlers.

If you dispatch the domain events right *before* committing the original transaction, it is because you want the side effects of those events to be included in the same transaction. For example, if the EF DbContext SaveChanges method fails, the transaction will roll back all changes, including the result of any side effect operations implemented by the related domain event handlers. This is because the DbContext life scope is by default defined as "scoped." Therefore, the DbContext object is shared across multiple repository objects being instantiated within the same scope or object graph. This coincides with the HttpRequest scope when developing Web API or MVC apps.

Actually, both approaches (single atomic transaction and eventual consistency) can be right. It really depends on your domain or business requirements and what the domain experts tell you. It also depends on how scalable you need the service to be (more granular transactions have less impact with regard to database locks). And it depends on how much investment you're willing to make in your code, since eventual consistency requires more complex code in order to detect possible inconsistencies across aggregates and the need to implement compensatory actions. Consider that if you commit changes to the original aggregate and afterwards, when the events are being dispatched, if there's an issue and the event handlers cannot commit their side effects, you'll have inconsistencies between aggregates.

A way to allow compensatory actions would be to store the domain events in additional database tables so they can be part of the original transaction. Afterwards, you could have a batch process that detects inconsistencies and runs compensatory actions by comparing the list of events with the current state of the aggregates. The compensatory actions are part of a complex topic that will require deep analysis from your side, which includes discussing it with the business user and domain experts.

In any case, you can choose the approach you need. But the initial deferred approach—raising the events before committing, so you use a single transaction—is the simplest approach when using EF Core and a relational database. It's easier to implement and valid in many business cases. It's also the approach used in the ordering microservice in eShopOnContainers.

But how do you actually dispatch those events to their respective event handlers? What's the _mediator object you see in the previous example? It has to do with the techniques and artifacts you use to map between events and their event handlers.

The domain event dispatcher: mapping from events to event handlers

Once you're able to dispatch or publish the events, you need some kind of artifact that will publish the event, so that every related handler can get it and process side effects based on that event.

One approach is a real messaging system or even an event bus, possibly based on a service bus as opposed to in-memory events. However, for the first case, real messaging would be overkill for processing domain events, since you just need to process those events within the same process (that is, within the same domain and application layer).

How to subscribe to domain events

When you use MediatR, each event handler must use an event type that is provided on the generic parameter of the INotificationHandler interface, as you can see in the following code:

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
  : INotificationHandler<OrderStartedDomainEvent>
```

Based on the relationship between event and event handler, which can be considered the subscription, the MediatR artifact can discover all the event handlers for each event and trigger each one of those event handlers.

How to handle domain events

Finally, the event handler usually implements application layer code that uses infrastructure repositories to obtain the required additional aggregates and to execute side-effect domain logic. The following domain event handler code at eShopOnContainers, shows an implementation example.

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : INotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILogger _logger;
    private readonly IBuyerRepository _buyerRepository;
    private readonly IOrderingIntegrationEventService _orderingIntegrationEventService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(
        ILogger<ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler> logger,
        IBuyerRepository buyerRepository,
        IOrderingIntegrationEventService orderingIntegrationEventService)
    {
        _buyerRepository = buyerRepository ?? throw new
ArgumentNullException(nameof(buyerRepository));
        _orderingIntegrationEventService = orderingIntegrationEventService ?? throw new
ArgumentNullException(nameof(orderingIntegrationEventService));
```

```
        _logger = logger ?? throw new ArgumentNullException(nameof(logger));
    }

    public async Task Handle(
        OrderStartedDomainEvent domainEvent, CancellationToken cancellationToken)
    {
        var cardTypeId = domainEvent.CardTypeId != 0 ? domainEvent.CardTypeId : 1;
        var buyer = await _buyerRepository.FindAsync(domainEvent.UserId);
        var buyerExisted = buyer is not null;

        if (!buyerExisted)
        {
            buyer = new Buyer(domainEvent.UserId, domainEvent.UserName);
        }

        buyer.VerifyOrAddPaymentMethod(
            cardTypeId,
            $"Payment Method on {DateTime.UtcNow}",
            domainEvent.CardNumber,
            domainEvent.CardSecurityNumber,
            domainEvent.CardHolderName,
            domainEvent.CardExpiration,
            domainEvent.Order.Id);

        var buyerUpdated = buyerExisted ?
            _buyerRepository.Update(buyer) :
            _buyerRepository.Add(buyer);

        await _buyerRepository.UnitOfWork
            .SaveEntitiesAsync(cancellationToken);

        var integrationEvent = new OrderStatusChangedToSubmittedIntegrationEvent(
            domainEvent.Order.Id, domainEvent.Order.OrderStatus.Name, buyer.Name);
        await _orderingIntegrationEventService.AddAndSaveEventAsync(integrationEvent);

        OrderingApiTrace.LogOrderBuyerAndPaymentValidatedOrUpdated(
            _logger, buyerUpdated.Id, domainEvent.Order.Id);
    }
}
```

The previous domain event handler code is considered application layer code because it uses infrastructure repositories, as explained in the next section on the infrastructure-persistence layer. Event handlers could also use other infrastructure components.

Domain events can generate integration events to be published outside of the microservice boundaries

Finally, it's important to mention that you might sometimes want to propagate events across multiple microservices. That propagation is an integration event, and it could be published through an event bus from any specific domain event handler.

Conclusions on domain events

As stated, use domain events to explicitly implement side effects of changes within your domain. To use DDD terminology, use domain events to explicitly implement side effects across one or multiple aggregates. Additionally, and for better scalability and less impact on database locks, use eventual consistency between aggregates within the same domain.

The reference app uses MediatR to propagate domain events synchronously across aggregates, within a single transaction. However, you could also use some AMQP implementation like RabbitMQ or Azure Service Bus to propagate domain events asynchronously, using eventual consistency but, as mentioned above, you have to consider the need for compensatory actions in case of failures.

Additional resources

- **Greg Young. What is a Domain Event?**
  https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf#page=25
- **Jan Stenberg. Domain Events and Eventual Consistency**
  https://www.infoq.com/news/2015/09/domain-events-consistency
- **Jimmy Bogard. A better domain events pattern**
  https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/
- **Vaughn Vernon. Effective Aggregate Design Part II: Making Aggregates Work Together**
  https://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
- **Jimmy Bogard. Strengthening your domain: Domain Events**
  https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/
- **Udi Dahan. How to create fully encapsulated Domain Models**
  https://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/
- **Udi Dahan. Domain Events – Take 2**
  https://udidahan.com/2008/08/25/domain-events-take-2/
- **Udi Dahan. Domain Events – Salvation**
  https://udidahan.com/2009/06/14/domain-events-salvation/
- **Cesar de la Torre. Domain Events vs. Integration Events in DDD and microservices architectures**
  https://devblogs.microsoft.com/cesardelatorre/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/

Design the infrastructure persistence layer

Data persistence components provide access to the data hosted within the boundaries of a microservice (that is, a microservice's database). They contain the actual implementation of components such as repositories and Unit of Work classes, like custom Entity Framework (EF) DbContext objects. EF DbContext implements both the Repository and the Unit of Work patterns.

The Repository pattern

The Repository pattern is a Domain-Driven Design pattern intended to keep persistence concerns outside of the system's domain model. One or more persistence abstractions - interfaces - are defined

in the domain model, and these abstractions have implementations in the form of persistence-specific adapters defined elsewhere in the application.

Repository implementations are classes that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the domain model. If you use an Object-Relational Mapper (ORM) like Entity Framework, the code that must be implemented is simplified, thanks to LINQ and strong typing. This lets you focus on the data persistence logic rather than on data access plumbing.

The Repository pattern is a well-documented way of working with a data source. In the book Patterns of Enterprise Application Architecture, Martin Fowler describes a repository as follows:

A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting in a similar way to a set of domain objects in memory. Client objects declaratively build queries and send them to the repositories for answers. Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way that is closer to the persistence layer. Repositories, also, support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping.

Define one repository per aggregate

For each aggregate or aggregate root, you should create one repository class. You may be able to leverage C# Generics to reduce the total number concrete classes you need to maintain (as demonstrated later in this chapter). In a microservice based on Domain-Driven Design (DDD) patterns, the only channel you should use to update the database should be the repositories. This is because they have a one-to-one relationship with the aggregate root, which controls the aggregate's invariants and transactional consistency. It's okay to query the database through other channels (as you can do following a CQRS approach), because queries don't change the state of the database. However, the transactional area (that is, the updates) must always be controlled by the repositories and the aggregate roots.

Basically, a repository allows you to populate data in memory that comes from the database in the form of the domain entities. Once the entities are in memory, they can be changed and then persisted back to the database through transactions.

As noted earlier, if you're using the CQS/CQRS architectural pattern, the initial queries are performed by side queries out of the domain model, performed by simple SQL statements using Dapper. This approach is much more flexible than repositories because you can query and join any tables you need, and these queries aren't restricted by rules from the aggregates. That data goes to the presentation layer or client app.

If the user makes changes, the data to be updated comes from the client app or presentation layer to the application layer (such as a Web API service). When you receive a command in a command handler, you use repositories to get the data you want to update from the database. You update it in memory with the data passed with the commands, and you then add or update the data (domain entities) in the database through a transaction.

It's important to emphasize again that you should only define one repository for each aggregate root, as shown in Figure 7-17. To achieve the goal of the aggregate root to maintain transactional consistency between all the objects within the aggregate, you should never create a repository for each table in the database.

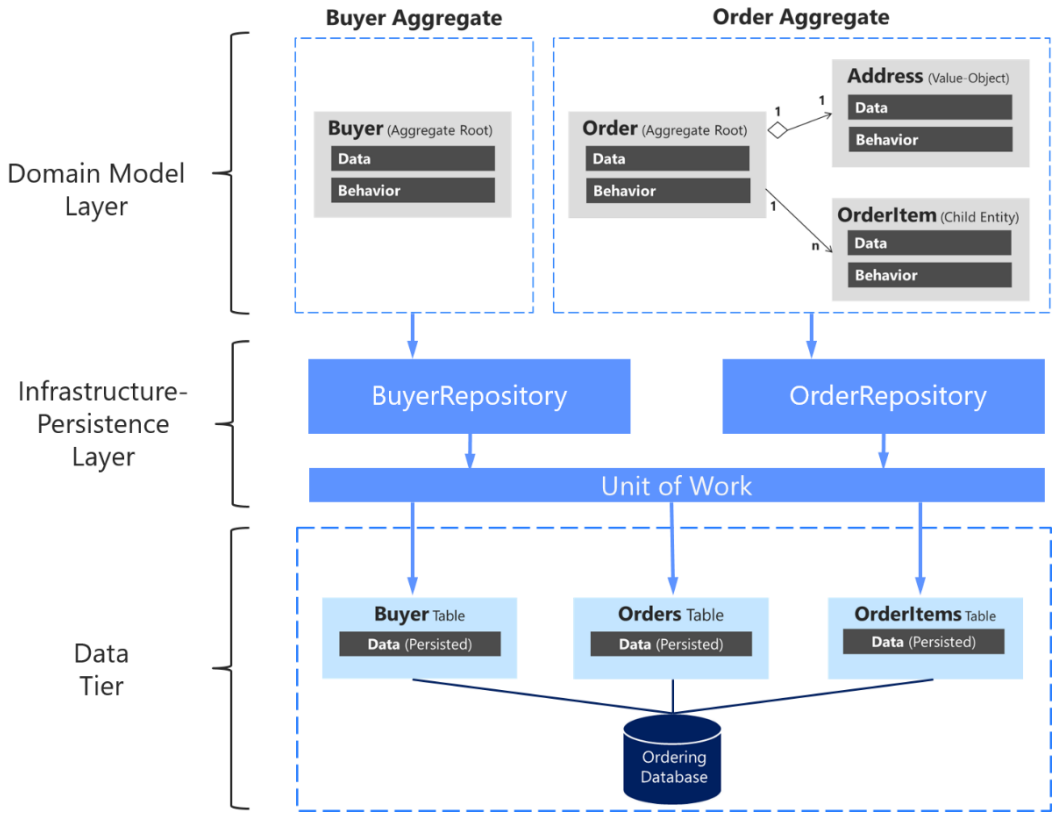


Figure 7-17. The relationship between repositories, aggregates, and database tables

The above diagram shows the relationships between Domain and Infrastructure layers: Buyer Aggregate depends on the IBuyerRepository and Order Aggregate depends on the IOrderRepository interfaces, these interfaces are implemented in the Infrastructure layer by the corresponding repositories that depend on UnitOfWork, also implemented there, that accesses the tables in the Data tier.

Enforce one aggregate root per repository

It can be valuable to implement your repository design in such a way that it enforces the rule that only aggregate roots should have repositories. You can create a generic or base repository type that constrains the type of entities it works with to ensure they have the IAggregateRoot marker interface.

Thus, each repository class implemented at the infrastructure layer implements its own contract or interface, as shown in the following code:

```
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
      // ...
```

```
    }
}
```

Each specific repository interface implements the generic IRepository interface:

```
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);
    // ...
}
```

However, a better way to have the code enforce the convention that each repository is related to a single aggregate is to implement a generic repository type. That way, it's explicit that you're using a repository to target a specific aggregate. That can be easily done by implementing a generic IRepository base interface, as in the following code:

```
public interface IRepository<T> where T : IAggregateRoot
{
    //....
}
```

The Repository pattern makes it easier to test your application logic

The Repository pattern allows you to easily test your application with unit tests. Remember that unit tests only test your code, not infrastructure, so the repository abstractions make it easier to achieve that goal.

As noted in an earlier section, it's recommended that you define and place the repository interfaces in the domain model layer so the application layer, such as your Web API microservice, doesn't depend directly on the infrastructure layer where you've implemented the actual repository classes. By doing this and using Dependency Injection in the controllers of your Web API, you can implement mock repositories that return fake data instead of data from the database. This decoupled approach allows you to create and run unit tests that focus the logic of your application without requiring connectivity to the database.

Connections to databases can fail and, more importantly, running hundreds of tests against a database is bad for two reasons. First, it can take a long time because of the large number of tests. Second, the database records might change and impact the results of your tests, especially if your tests are running in parallel, so that they might not be consistent. Unit tests typically can run in parallel; integration tests may not support parallel execution depending on their implementation. Testing against the database isn't a unit test but an integration test. You should have many unit tests running fast, but fewer integration tests against the databases.

In terms of separation of concerns for unit tests, your logic operates on domain entities in memory. It assumes the repository class has delivered those. Once your logic modifies the domain entities, it assumes the repository class will store them correctly. The important point here is to create unit tests against your domain model and its domain logic. Aggregate roots are the main consistency boundaries in DDD.

The repositories implemented in eShopOnContainers rely on EF Core's DbContext implementation of the Repository and Unit of Work patterns using its change tracker, so they don't duplicate this functionality.

The difference between the Repository pattern and the legacy Data Access class (DAL class) pattern

A typical DAL object directly performs data access and persistence operations against storage, often at the level of a single table and row. Simple CRUD operations implemented with a set of DAL classes frequently do not support transactions (though this is not always the case). Most DAL class approaches make minimal use of abstractions, resulting in tight coupling between application or Business Logic Layer (BLL) classes that call the DAL objects.

When using repository, the implementation details of persistence are encapsulated away from the domain model. The use of an abstraction provides ease of extending behavior through patterns like Decorators or Proxies. For instance, cross-cutting concerns like caching, logging, and error handling can all be applied using these patterns rather than hard-coded in the data access code itself. It's also trivial to support multiple repository adapters which may be used in different environments, from local development to shared staging environments to production.

Implementing Unit of Work

A unit of work refers to a single transaction that involves multiple insert, update, or delete operations. In simple terms, it means that for a specific user action, such as a registration on a website, all the insert, update, and delete operations are handled in a single transaction. This is more efficient than handling multiple database operations in a chattier way.

These multiple persistence operations are performed later in a single action when your code from the application layer commands it. The decision about applying the in-memory changes to the actual database storage is typically based on the Unit of Work pattern. In EF, the Unit of Work pattern is implemented by a DbContext and is executed when a call is made to SaveChanges.

In many cases, this pattern or way of applying operations against the storage can increase application performance and reduce the possibility of inconsistencies. It also reduces transaction blocking in the database tables, because all the intended operations are committed as part of one transaction. This is more efficient in comparison to executing many isolated operations against the database. Therefore, the selected ORM can optimize the execution against the database by grouping several update actions within the same transaction, as opposed to many small and separate transaction executions.

The Unit of Work pattern can be implemented with or without using the Repository pattern.

Repositories shouldn't be mandatory

Custom repositories are useful for the reasons cited earlier, and that is the approach for the ordering microservice in eShopOnContainers. However, it isn't an essential pattern to implement in a DDD design or even in general .NET development.

For instance, Jimmy Bogard, when providing direct feedback for this guide, said the following:

This'll probably be my biggest feedback. I'm really not a fan of repositories, mainly because they hide the important details of the underlying persistence mechanism. It's why I go for MediatR for commands, too. I can use the full power of the persistence layer, and push all that domain behavior into my aggregate roots. I don't usually want to mock my repositories – I still need to have that

integration test with the real thing. Going CQRS meant that we didn't really have a need for repositories any more.

Repositories might be useful, but they are not critical for your DDD design in the way that the Aggregate pattern and a rich domain model are. Therefore, use the Repository pattern or not, as you see fit.

Additional resources

Repository pattern

- **Edward Hieatt and Rob Mee. Repository pattern.** https://martinfowler.com/eaaCatalog/repository.html
- **The Repository pattern** https://learn.microsoft.com/previous-versions/msp-n-p/ff649690(v=pandp.10)
- **Eric Evans. Domain-Driven Design: Tackling Complexity in the Heart of Software.** (Book; includes a discussion of the Repository pattern) https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215/

Unit of Work pattern

- **Martin Fowler. Unit of Work pattern.** https://martinfowler.com/eaaCatalog/unitOfWork.html
- **Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application** https://learn.microsoft.com/aspnet/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

Implement the infrastructure persistence layer with Entity Framework Core

When you use relational databases such as SQL Server, Oracle, or PostgreSQL, a recommended approach is to implement the persistence layer based on Entity Framework (EF). EF supports LINQ and provides strongly typed objects for your model, as well as simplified persistence into your database.

Entity Framework has a long history as part of the .NET Framework. When you use .NET, you should also use Entity Framework Core, which runs on Windows or Linux in the same way as .NET. EF Core is a complete rewrite of Entity Framework that's implemented with a much smaller footprint and important improvements in performance.

Introduction to Entity Framework Core

Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology. It was introduced with .NET Core in mid-2016.

Since an introduction to EF Core is already available in Microsoft documentation, here we simply provide links to that information.

Additional resources

- **Entity Framework Core**
  https://learn.microsoft.com/ef/core/
- **Getting started with ASP.NET Core and Entity Framework Core using Visual Studio**
  https://learn.microsoft.com/aspnet/core/data/ef-mvc/
- **DbContext Class**
  https://learn.microsoft.com/dotnet/api/microsoft.entityframeworkcore.dbcontext
- **Compare EF Core & EF6.x**
  https://learn.microsoft.com/ef/efcore-and-ef6/index

Infrastructure in Entity Framework Core from a DDD perspective

From a DDD point of view, an important capability of EF is the ability to use POCO domain entities, also known in EF terminology as POCO *code-first entities*. If you use POCO domain entities, your domain model classes are persistence-ignorant, following the Persistence Ignorance and the Infrastructure Ignorance principles.

Per DDD patterns, you should encapsulate domain behavior and rules within the entity class itself, so it can control invariants, validations, and rules when accessing any collection. Therefore, it is not a good practice in DDD to allow public access to collections of child entities or value objects. Instead, you want to expose methods that control how and when your fields and property collections can be updated, and what behavior and actions should occur when that happens.

Since EF Core 1.1, to satisfy those DDD requirements, you can have plain fields in your entities instead of public properties. If you do not want an entity field to be externally accessible, you can just create the attribute or field instead of a property. You can also use private property setters.

In a similar way, you can now have read-only access to collections by using a public property typed as IReadOnlyCollection<T>, which is backed by a private field member for the collection (like a List<T>) in your entity that relies on EF for persistence. Previous versions of Entity Framework required collection properties to support ICollection<T>, which meant that any developer using the parent entity class could add or remove items through its property collections. That possibility would be against the recommended patterns in DDD.

You can use a private collection while exposing a read-only IReadOnlyCollection<T> object, as shown in the following code example:

```
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);
    }
}
```

The OrderItems property can only be accessed as read-only using IReadOnlyCollection<OrderItem>. This type is read-only so it is protected against regular external updates.

EF Core provides a way to map the domain model to the physical database without "contaminating" the domain model. It is pure .NET POCO code, because the mapping action is implemented in the persistence layer. In that mapping action, you need to configure the fields-to-database mapping. In the following example of the OnModelCreating method from OrderingContext and the OrderEntityTypeConfiguration class, the call to SetPropertyAccessMode tells EF Core to access the OrderItems property through its field.

```
// 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
    }
}
```

When you use fields instead of properties, the OrderItem entity is persisted as if it had a List<OrderItem> property. However, it exposes a single accessor, the AddOrderItem method, for adding new items to the order. As a result, behavior and data are tied together and will be consistent throughout any application code that uses the domain model.

Implement custom repositories with Entity Framework Core

At the implementation level, a repository is simply a class with data persistence code coordinated by a unit of work (DBContext in EF Core) when performing updates, as shown in the following class:

```
// 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;
        }
    }
}
```

The IBuyerRepository interface comes from the domain model layer as a contract. However, the repository implementation is done at the persistence and infrastructure layer.

The EF DbContext comes through the constructor through Dependency Injection. It is shared between multiple repositories within the same HTTP request scope, thanks to its default lifetime (ServiceLifetime.Scoped) in the IoC container (which can also be explicitly set with services.AddDbContext<>).

Methods to implement in a repository (updates or transactions versus queries)

Within each repository class, you should put the persistence methods that update the state of entities contained by its related aggregate. Remember there is one-to-one relationship between an aggregate and its related repository. Consider that an aggregate root entity object might have embedded child entities within its EF graph. For example, a buyer might have multiple payment methods as related child entities.

Since the approach for the ordering microservice in eShopOnContainers is also based on CQS/CQRS, most of the queries are not implemented in custom repositories. Developers have the freedom to create the queries and joins they need for the presentation layer without the restrictions imposed by aggregates, custom repositories per aggregate, and DDD in general. Most of the custom repositories suggested by this guide have several update or transactional methods but just the query methods needed to get data to be updated. For example, the BuyerRepository repository implements a FindAsync method, because the application needs to know whether a particular buyer exists before creating a new buyer related to the order.

However, the real query methods to get data to send to the presentation layer or client apps are implemented, as mentioned, in the CQRS queries based on flexible queries using Dapper.

Using a custom repository versus using EF DbContext directly

The Entity Framework DbContext class is based on the Unit of Work and Repository patterns and can be used directly from your code, such as from an ASP.NET Core MVC controller. The Unit of Work and Repository patterns result in the simplest code, as in the CRUD catalog microservice in eShopOnContainers. In cases where you want the simplest code possible, you might want to directly use the DbContext class, as many developers do.

However, implementing custom repositories provides several benefits when implementing more complex microservices or applications. The Unit of Work and Repository patterns are intended to encapsulate the infrastructure persistence layer so it is decoupled from the application and domain-model layers. Implementing these patterns can facilitate the use of mock repositories simulating access to the database.

In Figure 7-18, you can see the differences between not using repositories (directly using the EF DbContext) versus using repositories, which makes it easier to mock those repositories.

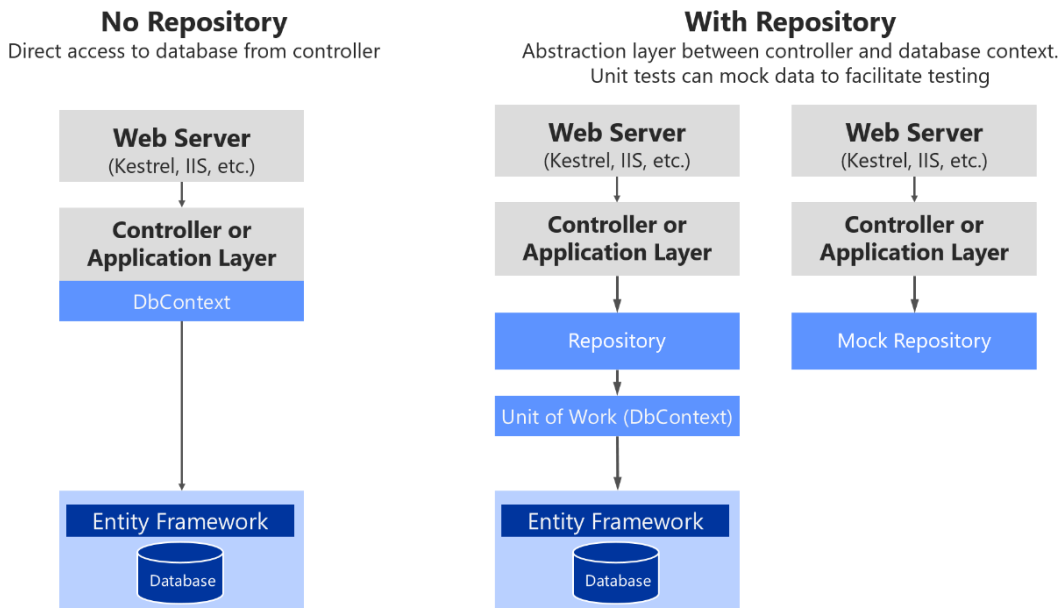


Figure 7-18. Using custom repositories versus a plain DbContext

Figure 7-18 shows that using a custom repository adds an abstraction layer that can be used to ease testing by mocking the repository. There are multiple alternatives when mocking. You could mock just repositories or you could mock a whole unit of work. Usually mocking just the repositories is enough, and the complexity to abstract and mock a whole unit of work is usually not needed.

Later, when we focus on the application layer, you will see how Dependency Injection works in ASP.NET Core and how it is implemented when using repositories.

In short, custom repositories allow you to test code more easily with unit tests that are not impacted by the data tier state. If you run tests that also access the actual database through the Entity Framework, they are not unit tests but integration tests, which are a lot slower.

If you were using DbContext directly, you would have to mock it or to run unit tests by using an in-memory SQL Server with predictable data for unit tests. But mocking the DbContext or controlling fake data requires more work than mocking at the repository level. Of course, you could always test the MVC controllers.

EF DbContext and IUnitOfWork instance lifetime in your IoC container

The DbContext object (exposed as an IUnitOfWork object) should be shared among multiple repositories within the same HTTP request scope. For example, this is true when the operation being executed must deal with multiple aggregates, or simply because you are using multiple repository instances. It is also important to mention that the IUnitOfWork interface is part of your domain layer, not an EF Core type.

In order to do that, the instance of the DbContext object has to have its service lifetime set to ServiceLifetime.Scoped. This is the default lifetime when registering a DbContext with builder.Services.AddDbContext in your IoC container from the *Program.cs* file in your ASP.NET Core Web API project. The following code illustrates this.

```
// Add framework services.
builder.Services.AddMvc(options =>
{
    options.Filters.Add(typeof(HttpGlobalExceptionFilter));
}).AddControllersAsServices();

builder.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.
    );
```

The DbContext instantiation mode should not be configured as ServiceLifetime.Transient or ServiceLifetime.Singleton.

The repository instance lifetime in your IoC container

In a similar way, repository's lifetime should usually be set as scoped (InstancePerLifetimeScope in Autofac). It could also be transient (InstancePerDependency in Autofac), but your service will be more efficient in regards to memory when using the scoped lifetime.

```
// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
    .As<IOrderRepository>()
    .InstancePerLifetimeScope();
```

Using the singleton lifetime for the repository could cause you serious concurrency problems when your DbContext is set to scoped (InstancePerLifetimeScope) lifetime (the default lifetimes for a DBContext). As long as your service lifetimes for your repositories and your DbContext are both Scoped, you'll avoid these issues.

Additional resources

- **Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application**
  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. Implementation Strategies for the Repository Pattern with Entity Framework, Dapper, and Chain**
  https://www.infoq.com/articles/repository-implementation-strategies
- **Cesar de la Torre. Comparing ASP.NET Core IoC container service lifetimes with Autofac IoC container instance scopes**
  https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Table mapping

Table mapping identifies the table data to be queried from and saved to the database. Previously you saw how domain entities (for example, a product or order domain) can be used to generate a related database schema. EF is strongly designed around the concept of *conventions*. Conventions address questions like "What will the name of a table be?" or "What property is the primary key?" Conventions are typically based on conventional names. For example, it is typical for the primary key to be a property that ends with Id.

By convention, each entity will be set up to map to a table with the same name as the DbSet<TEntity> property that exposes the entity on the derived context. If no DbSet<TEntity> value is provided for the given entity, the class name is used.

Data Annotations versus Fluent API

There are many additional EF Core conventions, and most of them can be changed by using either data annotations or Fluent API, implemented within the OnModelCreating method.

Data annotations must be used on the entity model classes themselves, which is a more intrusive way from a DDD point of view. This is because you are contaminating your model with data annotations related to the infrastructure database. On the other hand, Fluent API is a convenient way to change most conventions and mappings within your data persistence infrastructure layer, so the entity model will be clean and decoupled from the persistence infrastructure.

Fluent API and the OnModelCreating method

As mentioned, in order to change conventions and mappings, you can use the OnModelCreating method in the DbContext class.

The ordering microservice in eShopOnContainers implements explicit mapping and configuration, when needed, as shown in the following code.

```
// 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");
    }
}
```

You could set all the Fluent API mappings within the same OnModelCreating method, but it's advisable to partition that code and have multiple configuration classes, one per entity, as shown in

the example. Especially for large models, it is advisable to have separate configuration classes for configuring different entity types.

The code in the example shows a few explicit declarations and mapping. However, EF Core conventions do many of those mappings automatically, so the actual code you would need in your case might be smaller.

The Hi/Lo algorithm in EF Core

An interesting aspect of code in the preceding example is that it uses the Hi/Lo algorithm as the key generation strategy.

The Hi/Lo algorithm is useful when you need unique keys before committing changes. As a summary, the Hi-Lo algorithm assigns unique identifiers to table rows while not depending on storing the row in the database immediately. This lets you start using the identifiers right away, as happens with regular sequential database IDs.

The Hi/Lo algorithm describes a mechanism for getting a batch of unique IDs from a related database sequence. These IDs are safe to use because the database guarantees the uniqueness, so there will be no collisions between users. This algorithm is interesting for these reasons:

- It does not break the Unit of Work pattern.
- It gets sequence IDs in batches, to minimize round trips to the database.
- It generates a human readable identifier, unlike techniques that use GUIDs.

EF Core supports HiLo with the UseHiLo method, as shown in the preceding example.

Map fields instead of properties

With this feature, available since EF Core 1.1, you can directly map columns to fields. It is possible to not use properties in the entity class, and just to map columns from a table to fields. A common use for that would be private fields for any internal state that do not need to be accessed from outside the entity.

You can do this with single fields or also with collections, like a List<> field. This point was mentioned earlier when we discussed modeling the domain model classes, but here you can see how that mapping is performed with the PropertyAccessMode.Field configuration highlighted in the previous code.

Use shadow properties in EF Core, hidden at the infrastructure level

Shadow properties in EF Core are properties that do not exist in your entity class model. The values and states of these properties are maintained purely in the ChangeTracker class at the infrastructure level.

Implement the Query Specification pattern

As introduced earlier in the design section, the Query Specification pattern is a Domain-Driven Design pattern designed as the place where you can put the definition of a query with optional sorting and paging logic.

The Query Specification pattern defines a query in an object. For example, in order to encapsulate a paged query that searches for some products you can create a PagedProduct specification that takes the necessary input parameters (pageNumber, pageSize, filter, etc.). Then, within any Repository method (usually a List() overload) it would accept an ISpecification and run the expected query based on that specification.

An example of a generic Specification interface is the following code, which is similar to code used in the eShopOnWeb reference application.

```
// 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; }
}
```

Then, the implementation of a generic specification base class is the following.

```
// 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);
    }
}
```

The following specification loads a single basket entity given either the basket's ID or the ID of the buyer to whom the basket belongs. It will eagerly load the basket's Items collection.

```
// 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);
    }
}
```

And finally, you can see below how a generic EF Repository can use such a specification to filter and eager-load data related to a given entity type 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();
}
```

In addition to encapsulating filtering logic, the specification can specify the shape of the data to be returned, including which properties to populate.

Although we don't recommend returning IQueryable from a repository, it's perfectly fine to use them within the repository to build up a set of results. You can see this approach used in the List method above, which uses intermediate IQueryable expressions to build up the query's list of includes before executing the query with the specification's criteria on the last line.

Learn how the specification pattern is applied in the eShopOnWeb sample.

Additional resources

- **Table Mapping**
  https://learn.microsoft.com/ef/core/modeling/relational/tables
- **Use HiLo to generate keys with Entity Framework Core**
  https://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/
- **Backing Fields**
  https://learn.microsoft.com/ef/core/modeling/backing-field
- **Steve Smith. Encapsulated Collections in Entity Framework Core**
  https://ardalis.com/encapsulated-collections-in-entity-framework-core
- **Shadow Properties**
  https://learn.microsoft.com/ef/core/modeling/shadow-properties
- **The Specification pattern**
  https://deviq.com/specification-pattern/

  Ardalis.Specification NuGet Package Used by eShopOnWeb. https://www.nuget.org/packages/Ardalis.Specification

Use NoSQL databases as a persistence infrastructure

When you use NoSQL databases for your infrastructure data tier, you typically do not use an ORM like Entity Framework Core. Instead you use the API provided by the NoSQL engine, such as Azure Cosmos DB, MongoDB, Cassandra, RavenDB, CouchDB, or Azure Storage Tables.

However, when you use a NoSQL database, especially a document-oriented database like Azure Cosmos DB, CouchDB, or RavenDB, the way you design your model with DDD aggregates is partially similar to how you can do it in EF Core, in regards to the identification of aggregate roots, child entity classes, and value object classes. But, ultimately, the database selection will impact in your design.

When you use a document-oriented database, you implement an aggregate as a single document, serialized in JSON or another format. However, the use of the database is transparent from a domain model code point of view. When using a NoSQL database, you still are using entity classes and aggregate root classes, but with more flexibility than when using EF Core because the persistence is not relational.

The difference is in how you persist that model. If you implemented your domain model based on POCO entity classes, agnostic to the infrastructure persistence, it might look like you could move to a different persistence infrastructure, even from relational to NoSQL. However, that should not be your goal. There are always constraints and trade-offs in the different database technologies, so you will not be able to have the same model for relational or NoSQL databases. Changing persistence models is not a trivial task, because transactions and persistence operations will be very different.

For example, in a document-oriented database, it is okay for an aggregate root to have multiple child collection properties. In a relational database, querying multiple child collection properties is not

easily optimized, because you get a UNION ALL SQL statement back from EF. Having the same domain model for relational databases or NoSQL databases is not simple, and you should not try to do it. You really have to design your model with an understanding of how the data is going to be used in each particular database.

A benefit when using NoSQL databases is that the entities are more denormalized, so you do not set a table mapping. Your domain model can be more flexible than when using a relational database.

When you design your domain model based on aggregates, moving to NoSQL and document-oriented databases might be even easier than using a relational database, because the aggregates you design are similar to serialized documents in a document-oriented database. Then you can include in those "bags" all the information you might need for that aggregate.

For instance, the following JSON code is a sample implementation of an order aggregate when using a document-oriented database. It is similar to the order aggregate we implemented in the eShopOnContainers sample, but without using EF Core underneath.

```
{
    "id": "2024001",
    "orderDate": "2/25/2024",
    "buyerId": "1234567",
    "address": [
        {
        "street": "100 One Microsoft Way",
        "city": "Redmond",
        "state": "WA",
        "zip": "98052",
        "country": "U.S."
        }
    ],
    "orderItems": [
        {"id": 20240011, "productId": "123456", "productName": ".NET T-Shirt",
        "unitPrice": 25, "units": 2, "discount": 0},
        {"id": 20240012, "productId": "123457", "productName": ".NET Mug",
        "unitPrice": 15, "units": 1, "discount": 0}
    ]
}
```

Introduction to Azure Cosmos DB and the native Cosmos DB API

Azure Cosmos DB is Microsoft's globally distributed database service for mission-critical applications. Azure Cosmos DB provides turn-key global distribution, elastic scaling of throughput and storage worldwide, single-digit millisecond latencies at the 99th percentile, five well-defined consistency levels, and guaranteed high availability, all backed by industry-leading SLAs. Azure Cosmos DB automatically indexes data without requiring you to deal with schema and index management. It is multi-model and supports document, key-value, graph, and columnar data models.

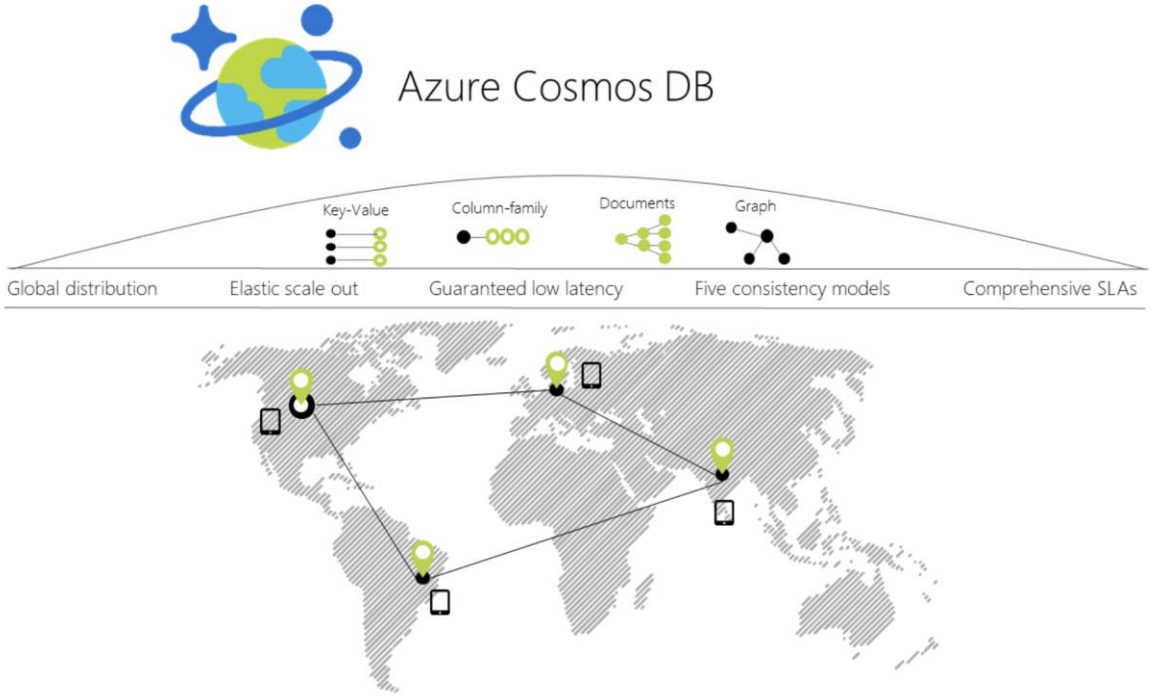


Figure 7-19. Azure Cosmos DB global distribution

When you use a C# model to implement the aggregate to be used by the Azure Cosmos DB API, the aggregate can be similar to the C# POCO classes used with EF Core. The difference is in the way to use them from the application and infrastructure layers, as in the following code:

```
// 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 = "2024001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "PO18009186470"
}

Address address = new Address
{
    Street = "100 One Microsoft Way",
    City = "Redmond",
    State = "WA",
    Zip = "98052",
    Country = "U.S."
}

orderAggregate.UpdateAddress(address);

OrderItem orderItem1 = new OrderItem
{
```

```
    Id = 20240011,
    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);
```

You can see that the way you work with your domain model can be similar to the way you use it in your domain model layer when the infrastructure is EF. You still use the same aggregate root methods to ensure consistency, invariants, and validations within the aggregate.

However, when you persist your model into the NoSQL database, the code and API change dramatically compared to EF Core code or any other code related to relational databases.

Implement .NET code targeting MongoDB and Azure Cosmos DB

Use Azure Cosmos DB from .NET containers

You can access Azure Cosmos DB databases from .NET code running in containers, like from any other .NET application. For instance, the Locations.API and Marketing.API microservices in eShopOnContainers are implemented so they can consume Azure Cosmos DB databases.

However, there's a limitation in Azure Cosmos DB from a Docker development environment point of view. Even though there's an on-premises Azure Cosmos DB Emulator that can run in a local development machine, it only supports Windows. Linux and macOS aren't supported.

There's also the possibility to run this emulator on Docker, but just on Windows Containers, not with Linux Containers. That's an initial handicap for the development environment if your application is deployed as Linux containers, since, currently, you can't deploy Linux and Windows Containers on Docker for Windows at the same time. Either all containers being deployed have to be for Linux or for Windows.

The ideal and more straightforward deployment for a dev/test solution is to be able to deploy your database systems as containers along with your custom containers so your dev/test environments are always consistent.

Use MongoDB API for local dev/test Linux/Windows containers plus Azure Cosmos DB

Cosmos DB databases support MongoDB API for .NET as well as the native MongoDB wire protocol. This means that by using existing drivers, your application written for MongoDB can now communicate with Cosmos DB and use Cosmos DB databases instead of MongoDB databases, as shown in Figure 7-20.

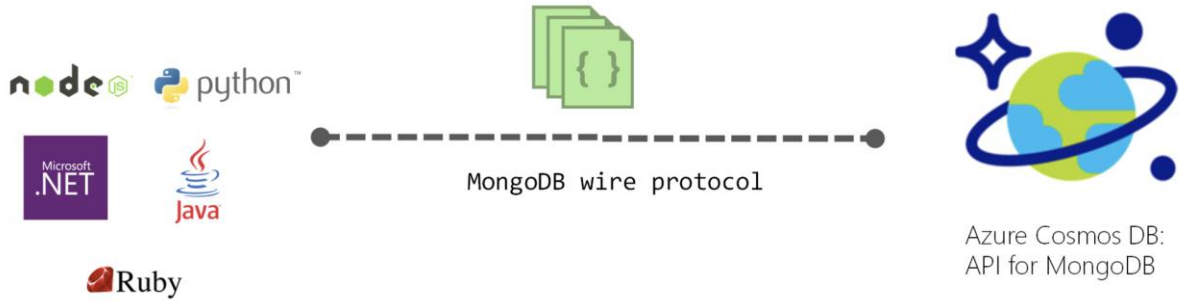


Figure 7-20. Using MongoDB API and protocol to access Azure Cosmos DB

This is a very convenient approach for proof of concepts in Docker environments with Linux containers because the MongoDB Docker image is a multi-arch image that supports Docker Linux containers and Docker Windows containers.

As shown in the following image, by using the MongoDB API, eShopOnContainers supports MongoDB Linux and Windows containers for the local development environment but then, you can move to a scalable, PaaS cloud solution as Azure Cosmos DB by simply changing the MongoDB connection string to point to Azure Cosmos DB.

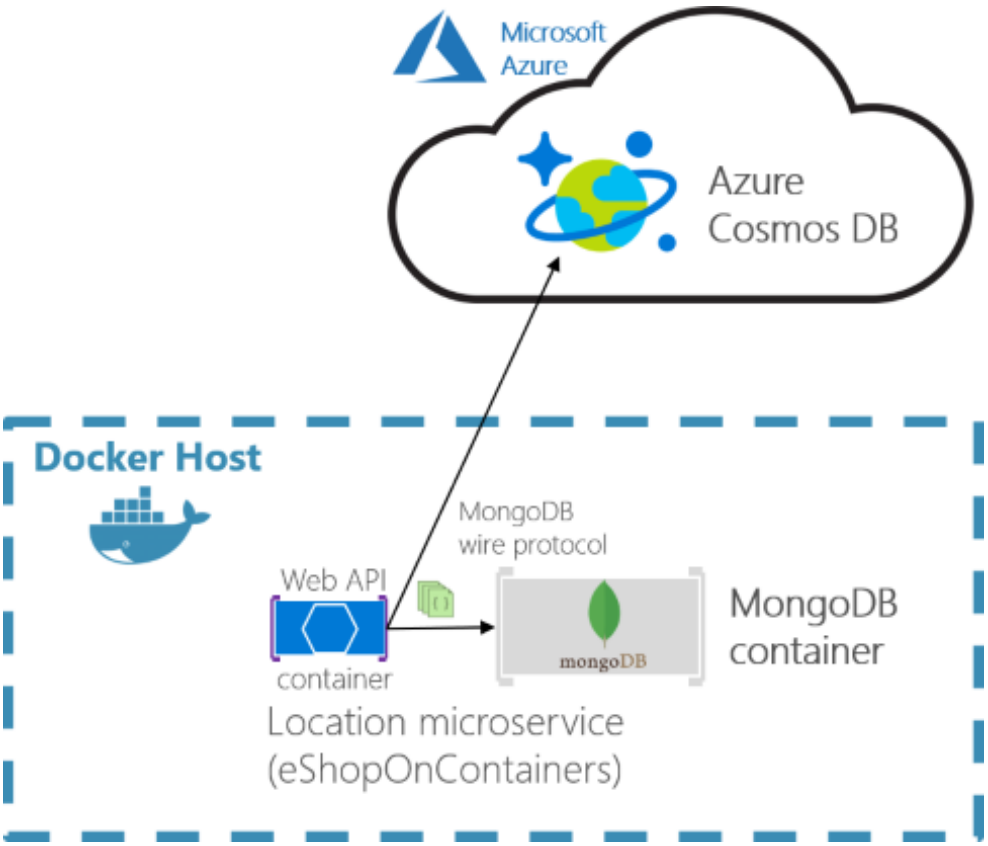


Figure 7-21. eShopOnContainers using MongoDB containers for dev-env or Azure Cosmos DB for production

The production Azure Cosmos DB would be running in Azure's cloud as a PaaS and scalable service.

Your custom .NET containers can run on a local development Docker host (that is using Docker for Windows in a Windows 10 machine) or be deployed into a production environment, like Kubernetes in Azure AKS or Azure Service Fabric. In this second environment, you would deploy only the .NET custom containers but not the MongoDB container since you'd be using Azure Cosmos DB in the cloud for handling the data in production.

A clear benefit of using the MongoDB API is that your solution could run in both database engines, MongoDB or Azure Cosmos DB, so migrations to different environments should be easy. However, sometimes it is worthwhile to use a native API (that is the native Cosmos DB API) in order to take full advantage of the capabilities of a specific database engine.

For further comparison between simply using MongoDB versus Cosmos DB in the cloud, see the Benefits of using Azure Cosmos DB in this page.

Analyze your approach for production applications: MongoDB API vs. Cosmos DB API

In eShopOnContainers, we're using MongoDB API because our priority was fundamentally to have a consistent dev/test environment using a NoSQL database that could also work with Azure Cosmos DB.

However, if you are planning to use MongoDB API to access Azure Cosmos DB in Azure for production applications, you should analyze the differences in capabilities and performance when using MongoDB API to access Azure Cosmos DB databases compared to using the native Azure Cosmos DB API. If it is similar you can use MongoDB API and you get the benefit of supporting two NoSQL database engines at the same time.

You could also use MongoDB clusters as the production database in Azure's cloud, too, with MongoDB Azure Service. But that is not a PaaS service provided by Microsoft. In this case, Azure is just hosting that solution coming from MongoDB.

Basically, this is just a disclaimer stating that you shouldn't always use MongoDB API against Azure Cosmos DB, as we did in eShopOnContainers because it was a convenient choice for Linux containers. The decision should be based on the specific needs and tests you need to do for your production application.

The code: Use MongoDB API in .NET applications

MongoDB API for .NET is based on NuGet packages that you need to add to your projects, like in the Locations.API project shown in the following figure.

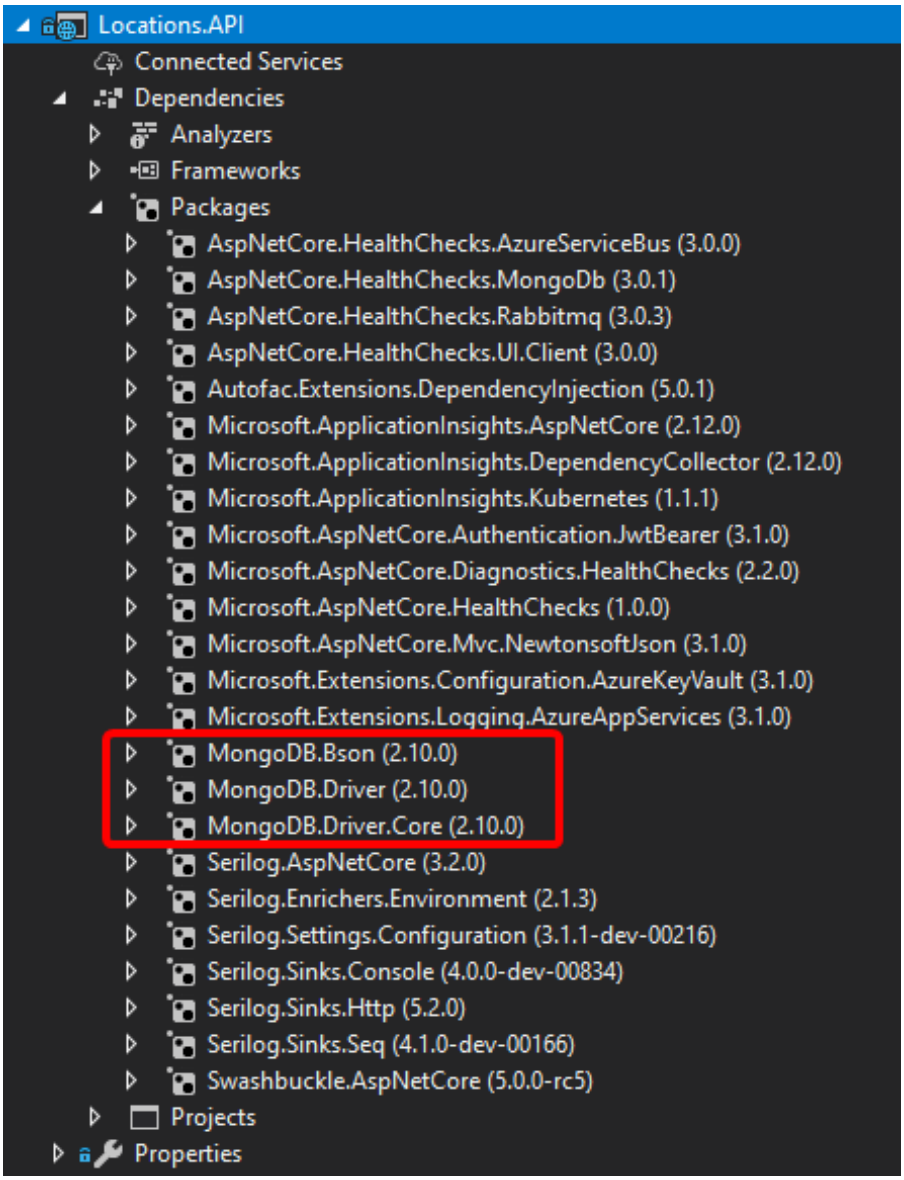


Figure 7-22. MongoDB API NuGet packages references in a .NET project

Let's investigate the code in the following sections.

A Model used by MongoDB API

First, you need to define a model that will hold the data coming from the database in your application's memory space. Here's an example of the model used for Locations at 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)));
    }
}
```

You can see there are a few attributes and types coming from the MongoDB NuGet packages.

NoSQL databases are usually very well suited for working with non-relational hierarchical data. In this example, we are using MongoDB types especially made for geo-locations, like GeoJson2DGeographicCoordinates.

Retrieve the database and the collection

In eShopOnContainers, we have created a custom database context where we implement the code to retrieve the database and the MongoCollections, as in the following code.

```
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");
        }
    }
}
```

Retrieve the data

In C# code, like Web API controllers or custom Repositories implementation, you can write similar code to the following when querying through the MongoDB API. Note that the _context object is an instance of the previous LocationsContext class.

```
public async Task<Locations> GetAsync(int locationId)
{
    var filter = Builders<Locations>.Filter.Eq("LocationId", locationId);
    return await _context.Locations
                            .Find(filter)
                            .FirstOrDefaultAsync();
}
```

Use an env-var in the docker-compose.override.yml file for the MongoDB connection string

When creating a MongoClient object, it needs a fundamental parameter which is precisely the ConnectionString parameter pointing to the right database. In the case of eShopOnContainers, the connection string can point to a local MongoDB Docker container or to a "production" Azure Cosmos DB database. That connection string comes from the environment variables defined in the docker-compose.override.yml files used when deploying with docker-compose or Visual Studio, as in the following yml code.

```
# docker-compose.override.yml
version: '3.4'
services:
  # Other services
  locations-api:
    environment:
      # Other settings
      - ConnectionString=${ESHOP_AZURE_COSMOSDB:-mongodb://nosqldata}
```

The ConnectionString environment variable is resolved this way: If the ESHOP_AZURE_COSMOSDB global variable is defined in the .env file with the Azure Cosmos DB connection string, it will use it to access the Azure Cosmos DB database in the cloud. If it's not defined, it will take the mongodb://nosqldata value and use the development MongoDB container.

The following code shows the .env file with the Azure Cosmos DB connection string global environment variable, as implemented in eShopOnContainers:

```
# .env file, in eShopOnContainers root folder
# Other Docker environment variables

ESHOP_EXTERNAL_DNS_NAME_OR_IP=host.docker.internal
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>
```

Uncomment the ESHOP_AZURE_COSMOSDB line and update it with your Azure Cosmos DB connection string obtained from the Azure portal as explained in Connect a MongoDB application to Azure Cosmos DB.

If the ESHOP_AZURE_COSMOSDB global variable is empty, meaning it's commented out in the .env file, then the container uses a default MongoDB connection string. This connection string points to the local MongoDB container deployed in eShopOnContainers that is named nosqldata and was defined at the docker-compose file, as shown in the following .yml code:

```yml
# docker-compose.yml
version: '3.4'
services:
  # ...Other services...
  nosqldata:
    image: mongo
```

Additional resources

- **Modeling document data for NoSQL databases**
  https://learn.microsoft.com/azure/cosmos-db/modeling-data
- **Vaughn Vernon. The Ideal Domain-Driven Design Aggregate Store?**
  https://kalele.io/blog-posts/the-ideal-domain-driven-design-aggregate-store/
- **Introduction to Azure Cosmos DB: API for MongoDB**
  https://learn.microsoft.com/azure/cosmos-db/mongodb-introduction
- **Azure Cosmos DB: Build a MongoDB API web app with .NET and the Azure portal**
  https://learn.microsoft.com/azure/cosmos-db/create-mongodb-dotnet
- **Use the Azure Cosmos DB Emulator for local development and testing**
  https://learn.microsoft.com/azure/cosmos-db/local-emulator
- **Connect a MongoDB application to Azure Cosmos DB**
  https://learn.microsoft.com/azure/cosmos-db/connect-mongodb-account
- **The Cosmos DB Emulator Docker image (Windows Container)**
  https://hub.docker.com/r/microsoft/azure-cosmosdb-emulator/
- **The MongoDB Docker image (Linux and Windows Container)**
  https://hub.docker.com/_/mongo/
- **Use MongoChef (Studio 3T) with an Azure Cosmos DB: API for MongoDB account**
  https://learn.microsoft.com/azure/cosmos-db/mongodb-mongochef

Design the microservice application layer and Web API

Use SOLID principles and Dependency Injection

SOLID principles are critical techniques to be used in any modern and mission-critical application, such as developing a microservice with DDD patterns. SOLID is an acronym that groups five fundamental principles:

- Single Responsibility principle
- Open/closed principle
- Liskov substitution principle
- Interface Segregation principle
- Dependency Inversion principle

SOLID is more about how you design your application or microservice internal layers and about decoupling dependencies between them. It is not related to the domain, but to the application's technical design. The final principle, the Dependency Inversion principle, allows you to decouple the infrastructure layer from the rest of the layers, which allows a better decoupled implementation of the DDD layers.

Dependency Injection (DI) is one way to implement the Dependency Inversion principle. It is a technique for achieving loose coupling between objects and their dependencies. Rather than directly instantiating collaborators, or using static references (that is, using new...), the objects that a class needs in order to perform its actions are provided to (or "injected into") the class. Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies principle. Dependency Injection is usually based on specific Inversion of Control (IoC) containers. ASP.NET Core provides a simple built-in IoC container, but you can also use your favorite IoC container, like Autofac or Ninject.

By following the SOLID principles, your classes will tend naturally to be small, well-factored, and easily tested. But how can you know if too many dependencies are being injected into your classes? If you use DI through the constructor, it will be easy to detect that by just looking at the number of parameters for your constructor. If there are too many dependencies, this is generally a sign (a code smell) that your class is trying to do too much, and is probably violating the Single Responsibility principle.

It would take another guide to cover SOLID in detail. Therefore, this guide requires you to have only a minimum knowledge of these topics.

Additional resources

- **SOLID: Fundamental OOP Principles**
  https://deviq.com/solid/

- **Inversion of Control Containers and the Dependency Injection pattern**
  https://martinfowler.com/articles/injection.html
- **Steve Smith. New is Glue**
  https://ardalis.com/new-is-glue

Implement the microservice application layer using the Web API

Use Dependency Injection to inject infrastructure objects into your application layer

As mentioned previously, the application layer can be implemented as part of the artifact (assembly) you are building, such as within a Web API project or an MVC web app project. In the case of a microservice built with ASP.NET Core, the application layer will usually be your Web API library. If you want to separate what is coming from ASP.NET Core (its infrastructure plus your controllers) from your custom application layer code, you could also place your application layer in a separate class library, but that is optional.

For instance, the application layer code of the ordering microservice is directly implemented as part of the **Ordering.API** project (an ASP.NET Core Web API project), as shown in Figure 7-23.

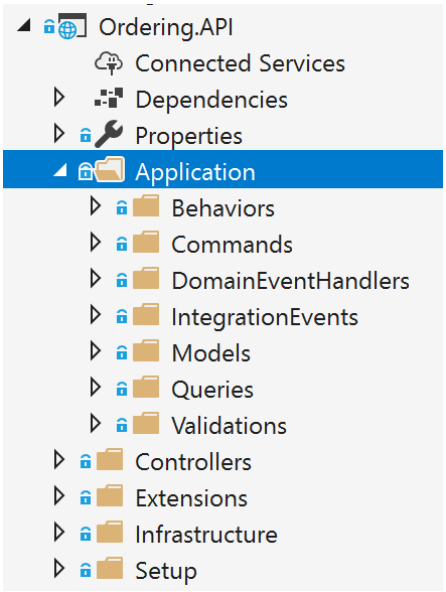


Figure 7-23. The application layer in the Ordering.API ASP.NET Core Web API project

ASP.NET Core includes a simple built-in IoC container (represented by the IServiceProvider interface) that supports constructor injection by default, and ASP.NET makes certain services available through DI. ASP.NET Core uses the term *service* for any of the types you register that will be injected through DI. You configure the built-in container's services in your application's *Program.cs* file. Your

dependencies are implemented in the services that a type needs and that you register in the IoC container.

Typically, you want to inject dependencies that implement infrastructure objects. A typical dependency to inject is a repository. But you could inject any other infrastructure dependency that you may have. For simpler implementations, you could directly inject your Unit of Work pattern object (the EF DbContext object), because the DBContext is also the implementation of your infrastructure persistence objects.

In the following example, you can see how .NET is injecting the required repository objects through the constructor. The class is a command handler, which will get covered in the next section.

```
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);
    }
}
```

The class uses the injected repositories to execute the transaction and persist the state changes. It does not matter whether that class is a command handler, an ASP.NET Core Web API controller method, or a DDD Application Service. It is ultimately a simple class that uses repositories, domain entities, and other application coordination in a fashion similar to a command handler. Dependency Injection works the same way for all the mentioned classes, as in the example using DI based on the constructor.

Register the dependency implementation types and interfaces or abstractions

Before you use the objects injected through constructors, you need to know where to register the interfaces and classes that produce the objects injected into your application classes through DI. (Like DI based on the constructor, as shown previously.)

Use the built-in IoC container provided by ASP.NET Core

When you use the built-in IoC container provided by ASP.NET Core, you register the types you want to inject in the *Program.cs* file, as in the following code:

```
// Register out-of-the-box framework services.
builder.Services.AddDbContext<CatalogContext>(c =>
    c.UseSqlServer(Configuration["ConnectionString"]),
    ServiceLifetime.Scoped);

builder.Services.AddMvc();
// Register custom application dependencies.
builder.Services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();
```

The most common pattern when registering types in an IoC container is to register a pair of types—an interface and its related implementation class. Then when you request an object from the IoC container through any constructor, you request an object of a certain type of interface. For instance, in the previous example, the last line states that when any of your constructors have a dependency on IMyCustomRepository (interface or abstraction), the IoC container will inject an instance of the MyCustomSQLServerRepository implementation class.

Use the Scrutor library for automatic types registration

When using DI in .NET, you might want to be able to scan an assembly and automatically register its types by convention. This feature is not currently available in ASP.NET Core. However, you can use the

Scrutor library for that. This approach is convenient when you have dozens of types that need to be registered in your IoC container.

Additional resources

- **Matthew King. Registering services with Scrutor**
  https://www.mking.net/blog/registering-services-with-scrutor
- **Kristian Hellang. Scrutor.** GitHub repo.
  https://github.com/khellang/Scrutor

Use Autofac as an IoC container

You can also use additional IoC containers and plug them into the ASP.NET Core pipeline, as in the ordering microservice in eShopOnContainers, which uses Autofac. When using Autofac you typically register the types via modules, which allow you to split the registration types between multiple files depending on where your types are, just as you could have the application types distributed across multiple class libraries.

For example, the following is the Autofac application module for the Ordering.API Web API project with the types you will want to inject.

```
public class ApplicationModule : Autofac.Module
{
    public string QueriesConnectionString { get; }
    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }

    protected override void Load(ContainerBuilder builder)
    {
        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>()
            .InstancePerLifetimeScope();
        builder.RegisterType<BuyerRepository>()
            .As<IBuyerRepository>()
            .InstancePerLifetimeScope();
        builder.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();
        builder.RegisterType<RequestManager>()
            .As<IRequestManager>()
            .InstancePerLifetimeScope();
   }
}
```

Autofac also has a feature to scan assemblies and register types by name conventions.

The registration process and concepts are very similar to the way you can register types with the built-in ASP.NET Core IoC container, but the syntax when using Autofac is a bit different.

In the example code, the abstraction IOrderRepository is registered along with the implementation class OrderRepository. This means that whenever a constructor is declaring a dependency through the

IOrderRepository abstraction or interface, the IoC container will inject an instance of the OrderRepository class.

The instance scope type determines how an instance is shared between requests for the same service or dependency. When a request is made for a dependency, the IoC container can return the following:

- A single instance per lifetime scope (referred to in the ASP.NET Core IoC container as *scoped*).
- A new instance per dependency (referred to in the ASP.NET Core IoC container as *transient*).
- A single instance shared across all objects using the IoC container (referred to in the ASP.NET Core IoC container as *singleton*).

Additional resources

- **Introduction to Dependency Injection in ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/fundamentals/dependency-injection
- **Autofac.** Official documentation.
  https://docs.autofac.org/en/latest/
- **Comparing ASP.NET Core IoC container service lifetimes with Autofac IoC container instance scopes - Cesar de la Torre.**
  https://devblogs.microsoft.com/cesardelatorre/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Implement the Command and Command Handler patterns

In the DI-through-constructor example shown in the previous section, the IoC container was injecting repositories through a constructor in a class. But exactly where were they injected? In a simple Web API (for example, the catalog microservice in eShopOnContainers), you inject them at the MVC controllers' level, in a controller constructor, as part of the request pipeline of ASP.NET Core. However, in the initial code of this section (the CreateOrderCommandHandler class from the Ordering.API service in eShopOnContainers), the injection of dependencies is done through the constructor of a particular command handler. Let us explain what a command handler is and why you would want to use it.

The Command pattern is intrinsically related to the CQRS pattern that was introduced earlier in this guide. CQRS has two sides. The first area is queries, using simplified queries with the Dapper micro ORM, which was explained previously. The second area is commands, which are the starting point for transactions, and the input channel from outside the service.

As shown in Figure 7-24, the pattern is based on accepting commands from the client-side, processing them based on the domain model rules, and finally persisting the states with transactions.

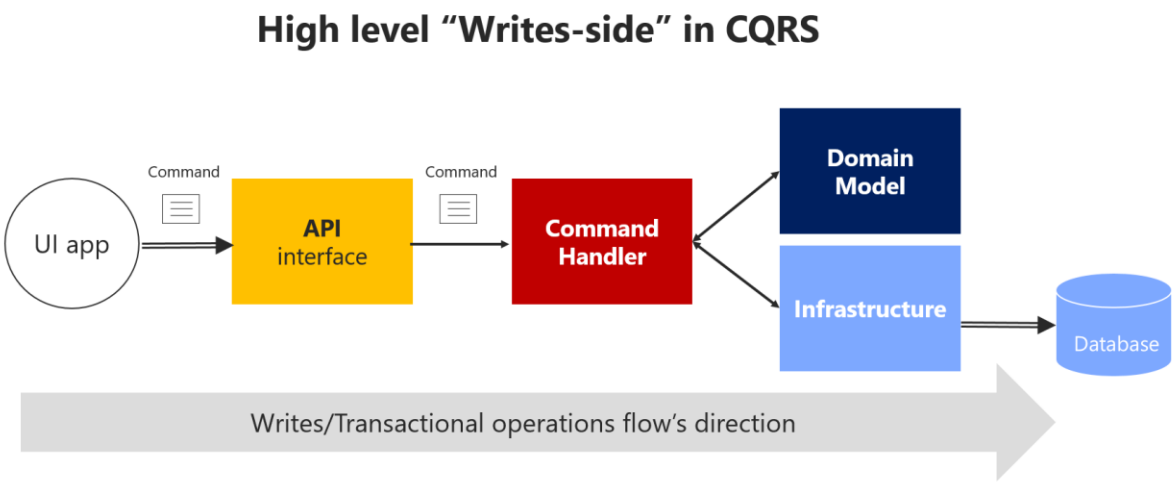


Figure 7-24. High-level view of the commands or "transactional side" in a CQRS pattern

Figure 7-24 shows that the UI app sends a command through the API that gets to a CommandHandler, that depends on the Domain model and the Infrastructure, to update the database.

The command class

A command is a request for the system to perform an action that changes the state of the system. Commands are imperative, and should be processed just once.

Since commands are imperatives, they are typically named with a verb in the imperative mood (for example, "create" or "update"), and they might include the aggregate type, such as CreateOrderCommand. Unlike an event, a command is not a fact from the past; it is only a request, and thus may be refused.

Commands can originate from the UI as a result of a user initiating a request, or from a process manager when the process manager is directing an aggregate to perform an action.

An important characteristic of a command is that it should be processed just once by a single receiver. This is because a command is a single action or transaction you want to perform in the application. For example, the same order creation command should not be processed more than once. This is an important difference between commands and events. Events may be processed multiple times, because many systems or microservices might be interested in the event.

In addition, it is important that a command be processed only once in case the command is not idempotent. A command is idempotent if it can be executed multiple times without changing the result, either because of the nature of the command, or because of the way the system handles the command.

It is a good practice to make your commands and updates idempotent when it makes sense under your domain's business rules and invariants. For instance, to use the same example, if for any reason (retry logic, hacking, etc.) the same CreateOrder command reaches your system multiple times, you should be able to identify it and ensure that you do not create multiple orders. To do so, you need to

attach some kind of identity in the operations and identify whether the command or update was already processed.

You send a command to a single receiver; you do not publish a command. Publishing is for events that state a fact—that something has happened and might be interesting for event receivers. In the case of events, the publisher has no concerns about which receivers get the event or what they do it. But domain or integration events are a different story already introduced in previous sections.

A command is implemented with a class that contains data fields or collections with all the information that is needed in order to execute that command. A command is a special kind of Data Transfer Object (DTO), one that is specifically used to request changes or transactions. The command itself is based on exactly the information that is needed for processing the command, and nothing more.

The following example shows the simplified CreateOrderCommand class. This is an immutable command that is used in the ordering microservice in 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://cqrs.nu/Faq
// https://docs.spine3.org/motivation/immutability.html
// http://blog.gauffin.org/2012/06/griffin-container-introducing-command-support/
// https://learn.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; }
    }
}
```

Basically, the command class contains all the data you need for performing a business transaction by using the domain model objects. Thus, commands are simply data structures that contain read-only data, and no behavior. The command's name indicates its purpose. In many languages like C#, commands are represented as classes, but they are not true classes in the real object-oriented sense.

As an additional characteristic, commands are immutable, because the expected usage is that they are processed directly by the domain model. They do not need to change during their projected lifetime. In a C# class, immutability can be achieved by not having any setters or other methods that change the internal state.

Keep in mind that if you intend or expect commands to go through a serializing/deserializing process, the properties must have a private setter, and the [DataMember] (or [JsonProperty]) attribute. Otherwise, the deserializer won't be able to reconstruct the object at the destination with the required values. You can also use truly read-only properties if the class has a constructor with parameters for all properties, with the usual camelCase naming convention, and annotate the constructor as [JsonConstructor]. However, this option requires more code.

For example, the command class for creating an order is probably similar in terms of data to the order you want to create, but you probably do not need the same attributes. For instance, CreateOrderCommand does not have an order ID, because the order has not been created yet.

Many command classes can be simple, requiring only a few fields about some state that needs to be changed. That would be the case if you are just changing the status of an order from "in process" to "paid" or "shipped" by using a command similar to the following:

```
[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; }
}
```

Some developers make their UI request objects separate from their command DTOs, but that is just a matter of preference. It is a tedious separation with not much additional value, and the objects are almost exactly the same shape. For instance, in eShopOnContainers, some commands come directly from the client-side.

The Command handler class

You should implement a specific command handler class for each command. That is how the pattern works, and it's where you'll use the command object, the domain objects, and the infrastructure repository objects. The command handler is in fact the heart of the application layer in terms of CQRS and DDD. However, all the domain logic should be contained in the domain classes—within the aggregate roots (root entities), child entities, or domain services, but not within the command handler, which is a class from the application layer.

The command handler class offers a strong stepping stone in the way to achieve the Single Responsibility Principle (SRP) mentioned in a previous section.

A command handler receives a command and obtains a result from the aggregate that is used. The result should be either successful execution of the command, or an exception. In the case of an exception, the system state should be unchanged.

The command handler usually takes the following steps:

- It receives the command object, like a DTO (from the mediator or other infrastructure object).
- It validates that the command is valid (if not validated by the mediator).
- It instantiates the aggregate root instance that is the target of the current command.
- It executes the method on the aggregate root instance, getting the required data from the command.
- It persists the new state of the aggregate to its related database. This last operation is the actual transaction.

Typically, a command handler deals with a single aggregate driven by its aggregate root (root entity). If multiple aggregates should be impacted by the reception of a single command, you could use domain events to propagate states or actions across multiple aggregates.

The important point here is that when a command is being processed, all the domain logic should be inside the domain model (the aggregates), fully encapsulated and ready for unit testing. The command handler just acts as a way to get the domain model from the database, and as the final step, to tell the infrastructure layer (repositories) to persist the changes when the model is changed. The advantage of this approach is that you can refactor the domain logic in an isolated, fully encapsulated, rich, behavioral domain model without changing code in the application or infrastructure layers, which are the plumbing level (command handlers, Web API, repositories, etc.).

When command handlers get complex, with too much logic, that can be a code smell. Review them, and if you find domain logic, refactor the code to move that domain behavior to the methods of the domain objects (the aggregate root and child entity).

As an example of a command handler class, the following code shows the same CreateOrderCommandHandler class that you saw at the beginning of this chapter. In this case, it also highlights the Handle method and the operations with the domain model objects/aggregates.

```
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);
    }
}
```

These are additional steps a command handler should take:

- Use the command's data to operate with the aggregate root's methods and behavior.
- Internally within the domain objects, raise domain events while the transaction is executed, but that is transparent from a command handler point of view.
- If the aggregate's operation result is successful and after the transaction is finished, raise integration events. (These might also be raised by infrastructure classes like repositories.)

Additional resources

- **Mark Seemann. At the Boundaries, Applications are Not Object-Oriented**
  https://blog.ploeh.dk/2011/05/31/AttheBoundaries,ApplicationsareNotObject-Oriented/
- **Commands and events**
  https://cqrs.nu/faq/Command%20and%20Events
- **What does a command handler do?**
  https://cqrs.nu/faq/Command%20Handlers
- **Jimmy Bogard. Domain Command Patterns – Handlers**
  https://jimmybogard.com/domain-command-patterns-handlers/
- **Jimmy Bogard. Domain Command Patterns – Validation**
  https://jimmybogard.com/domain-command-patterns-validation/

The Command process pipeline: how to trigger a command handler

The next question is how to invoke a command handler. You could manually call it from each related ASP.NET Core controller. However, that approach would be too coupled and is not ideal.

The other two main options, which are the recommended options, are:

- Through an in-memory Mediator pattern artifact.
- With an asynchronous message queue, in between controllers and handlers.

Use the Mediator pattern (in-memory) in the command pipeline

As shown in Figure 7-25, in a CQRS approach you use an intelligent mediator, similar to an in-memory bus, which is smart enough to redirect to the right command handler based on the type of the command or DTO being received. The single black arrows between components represent the dependencies between objects (in many cases, injected through DI) with their related interactions.

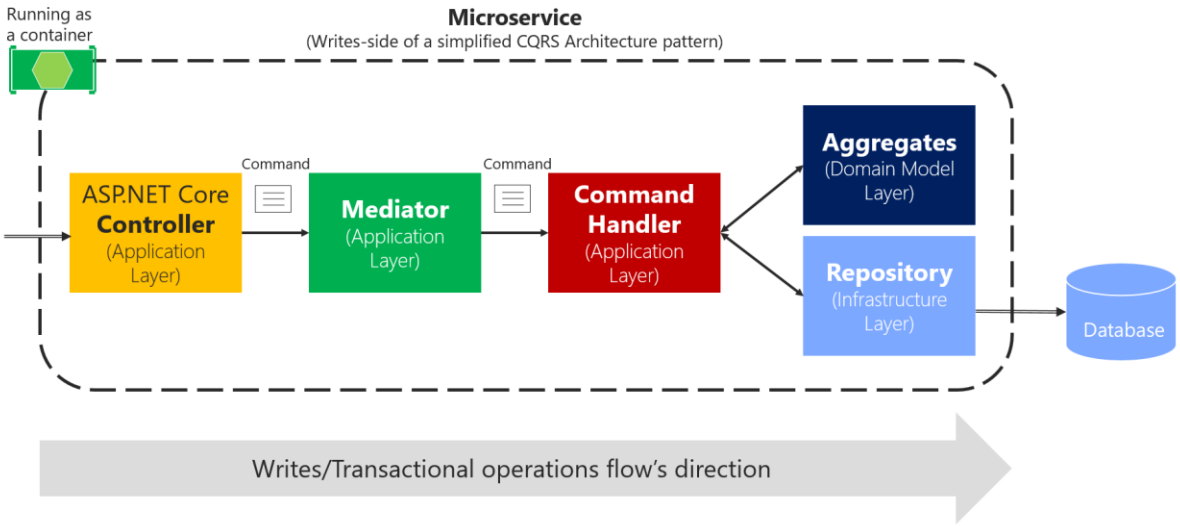


Figure 7-25. Using the Mediator pattern in process in a single CQRS microservice

The above diagram shows a zoom-in from image 7-24: the ASP.NET Core controller sends the command to MediatR's command pipeline, so they get to the appropriate handler.

The reason that using the Mediator pattern makes sense is that in enterprise applications, the processing requests can get complicated. You want to be able to add an open number of cross-cutting concerns like logging, validations, audit, and security. In these cases, you can rely on a mediator pipeline (see Mediator pattern) to provide a means for these extra behaviors or cross-cutting concerns.

A mediator is an object that encapsulates the "how" of this process: it coordinates execution based on state, the way a command handler is invoked, or the payload you provide to the handler. With a mediator component, you can apply cross-cutting concerns in a centralized and transparent way by applying decorators (or pipeline behaviors since MediatR 3). For more information, see the Decorator pattern.

Decorators and behaviors are similar to Aspect Oriented Programming (AOP), only applied to a specific process pipeline managed by the mediator component. Aspects in AOP that implement cross-cutting concerns are applied based on *aspect weavers* injected at compilation time or based on object call interception. Both typical AOP approaches are sometimes said to work "like magic," because it is not easy to see how AOP does its work. When dealing with serious issues or bugs, AOP can be difficult to debug. On the other hand, these decorators/behaviors are explicit and applied only in the context of the mediator, so debugging is much more predictable and easy.

For example, in the eShopOnContainers ordering microservice, has an implementation of two sample behaviors, a LogBehavior class and a ValidatorBehavior class. The implementation of the behaviors is explained in the next section by showing how eShopOnContainers uses MediatR behaviors.

Use message queues (out-of-proc) in the command's pipeline

Another choice is to use asynchronous messages based on brokers or message queues, as shown in Figure 7-26. That option could also be combined with the mediator component right before the command handler.

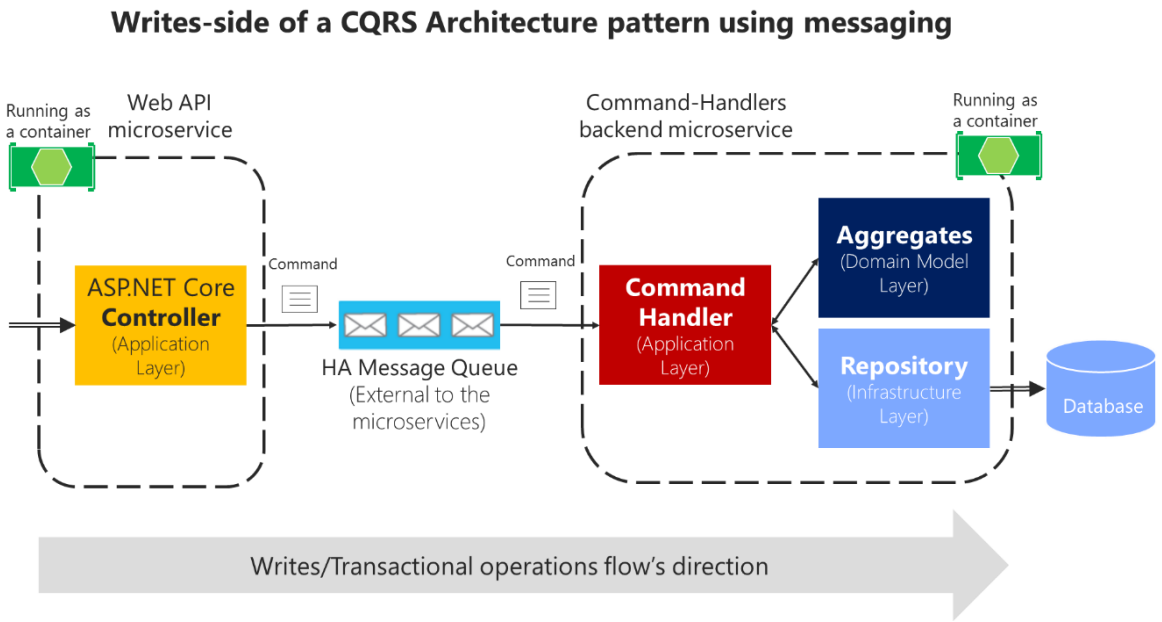


Figure 7-26. Using message queues (out of the process and inter-process communication) with CQRS commands

Command's pipeline can also be handled by a high availability message queue to deliver the commands to the appropriate handler. Using message queues to accept the commands can further complicate your command's pipeline, because you will probably need to split the pipeline into two processes connected through the external message queue. Still, it should be used if you need to have improved scalability and performance based on asynchronous messaging. Consider that in the case of Figure 7-26, the controller just posts the command message into the queue and returns. Then the command handlers process the messages at their own pace. That is a great benefit of queues: the message queue can act as a buffer in cases when hyper scalability is needed, such as for stocks or any other scenario with a high volume of ingress data.

However, because of the asynchronous nature of message queues, you need to figure out how to communicate with the client application about the success or failure of the command's process. As a rule, you should never use "fire and forget" commands. Every business application needs to know if a command was processed successfully, or at least validated and accepted.

Thus, being able to respond to the client after validating a command message that was submitted to an asynchronous queue adds complexity to your system, as compared to an in-process command process that returns the operation's result after running the transaction. Using queues, you might need to return the result of the command process through other operation result messages, which will require additional components and custom communication in your system.

Additionally, async commands are one-way commands, which in many cases might not be needed, as is explained in the following interesting exchange between Burtsev Alexey and Greg Young in an online conversation:

[Burtsev Alexey] I find lots of code where people use async command handling or one-way command messaging without any reason to do so (they are not doing some long operation, they are not executing external async code, they do not even cross-application boundary to be using message bus). Why do they introduce this unnecessary complexity? And actually, I haven't seen a CQRS code example with blocking command handlers so far, though it will work just fine in most cases.

[Greg Young] [...] an asynchronous command doesn't exist; it's actually another event. If I must accept what you send me and raise an event if I disagree, it's no longer you telling me to do something [that is, it's not a command]. It's you telling me something has been done. This seems like a slight difference at first, but it has many implications.

Asynchronous commands greatly increase the complexity of a system, because there is no simple way to indicate failures. Therefore, asynchronous commands are not recommended other than when scaling requirements are needed or in special cases when communicating the internal microservices through messaging. In those cases, you must design a separate reporting and recovery system for failures.

In the initial version of eShopOnContainers, it was decided to use synchronous command processing, started from HTTP requests and driven by the Mediator pattern. That easily allows you to return the success or failure of the process, as in the CreateOrderCommandHandler implementation.

In any case, this should be a decision based on your application's or microservice's business requirements.

Implement the command process pipeline with a mediator pattern (MediatR)

As a sample implementation, this guide proposes using the in-process pipeline based on the Mediator pattern to drive command ingestion and route commands, in memory, to the right command handlers. The guide also proposes applying behaviors in order to separate cross-cutting concerns.

For implementation in .NET, there are multiple open-source libraries available that implement the Mediator pattern. The library used in this guide is the MediatR open-source library (created by Jimmy Bogard), but you could use another approach. MediatR is a small and simple library that allows you to process in-memory messages like a command, while applying decorators or behaviors.

Using the Mediator pattern helps you to reduce coupling and to isolate the concerns of the requested work, while automatically connecting to the handler that performs that work—in this case, to command handlers.

Another good reason to use the Mediator pattern was explained by Jimmy Bogard when reviewing this guide:

I think it might be worth mentioning testing here – it provides a nice consistent window into the behavior of your system. Request-in, response-out. We've found that aspect quite valuable in building consistently behaving tests.

First, let's look at a sample WebAPI controller where you actually would use the mediator object. If you weren't using the mediator object, you'd need to inject all the dependencies for that controller, things like a logger object and others. Therefore, the constructor would be complicated. On the other hand, if you use the mediator object, the constructor of your controller can be a lot simpler, with just a few dependencies instead of many dependencies if you had one per cross-cutting operation, as in the following example:

```
public class MyMicroserviceController : Controller
{
    public MyMicroserviceController(IMediator mediator,
                                    IMyMicroserviceQueries microserviceQueries)
    {
        // ...
    }
}
```

You can see that the mediator provides a clean and lean Web API controller constructor. In addition, within the controller methods, the code to send a command to the mediator object is almost one line:

```
[Route("new")]
[HttpPost]
public async Task<IActionResult> ExecuteBusinessOperation([FromBody]RunOpCommand
                                                          runOperationCommand)
{
    var commandResult = await _mediator.SendAsync(runOperationCommand);

    return commandResult ? (IActionResult)Ok() : (IActionResult)BadRequest();
}
```

Implement idempotent Commands

In **eShopOnContainers**, a more advanced example than the above is submitting a CreateOrderCommand object from the Ordering microservice. But since the Ordering business process is a bit more complex and, in our case, it actually starts in the Basket microservice, this action of submitting the CreateOrderCommand object is performed from an integration-event handler named UserCheckoutAcceptedIntegrationEventHandler instead of a simple WebAPI controller called from the client App as in the previous simpler example.

Nevertheless, the action of submitting the Command to MediatR is pretty similar, as shown in the following code.

```
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);
```

However, this case is also slightly more advanced because we're also implementing idempotent commands. The CreateOrderCommand process should be idempotent, so if the same message comes duplicated through the network, because of any reason, like retries, the same business order will be processed just once.

This is implemented by wrapping the business command (in this case CreateOrderCommand) and embedding it into a generic IdentifiedCommand, which is tracked by an ID of every message coming through the network that has to be idempotent.

In the code below, you can see that the IdentifiedCommand is nothing more than a DTO with and ID plus the wrapped business command object.

```
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;
    }
}
```

Then the CommandHandler for the IdentifiedCommand named IdentifiedCommandHandler.cs will basically check if the ID coming as part of the message already exists in a table. If it already exists, that command won't be processed again, so it behaves as an idempotent command. That infrastructure code is performed by the _requestManager.ExistAsync method call below.

```
// 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 embedded 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);
            }
        }
    }
}
```

Since the IdentifiedCommand acts like a business command's envelope, when the business command needs to be processed because it is not a repeated ID, then it takes that inner business command and resubmits it to Mediator, as in the last part of the code shown above when running _mediator.Send(message.Command), from the IdentifiedCommandHandler.cs.

When doing that, it will link and run the business command handler, in this case, the CreateOrderCommandHandler, which is running transactions against the Ordering database, as shown in the following code.

```
// 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);
    }
}
```

Register the types used by MediatR

In order for MediatR to be aware of your command handler classes, you need to register the mediator classes and the command handler classes in your IoC container. By default, MediatR uses Autofac as the IoC container, but you can also use the built-in ASP.NET Core IoC container or any other container supported by MediatR.

The following code shows how to register Mediator's types and commands when using Autofac modules.

```
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
```

```
        //...
    }
}
```

This is where "the magic happens" with MediatR.

As each command handler implements the generic IRequestHandler<T> interface, when you register the assemblies using RegisteredAssemblyTypes method all the types marked as IRequestHandler also gets registered with their Commands. For example:

```
public class CreateOrderCommandHandler
  : IRequestHandler<CreateOrderCommand, bool>
{
```

That is the code that correlates commands with command handlers. The handler is just a simple class, but it inherits from RequestHandler<T>, where T is the command type, and MediatR makes sure it is invoked with the correct payload (the command).

Apply cross-cutting concerns when processing commands with the Behaviors in MediatR

There is one more thing: being able to apply cross-cutting concerns to the mediator pipeline. You can also see at the end of the Autofac registration module code how it registers a behavior type, specifically, a custom LoggingBehavior class and a ValidatorBehavior class. But you could add other custom behaviors, too.

```
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<,>));
    }
}
```

That LoggingBehavior class can be implemented as the following code, which logs information about the command handler being executed and whether it was successful or not.

```
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;
    }
}
```

Just by implementing this behavior class and by registering it in the pipeline (in the MediatorModule above), all the commands processed through MediatR will be logging information about the execution.

The eShopOnContainers ordering microservice also applies a second behavior for basic validations, the ValidatorBehavior class that relies on the FluentValidation library, as shown in the following code:

```
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;
    }
}
```

Here the behavior is raising an exception if validation fails, but you could also return a result object, containing the command result if it succeeded or the validation messages in case it didn't. This would probably make it easier to display validation results to the user.

Then, based on the FluentValidation library, you would create validation for the data passed with CreateOrderCommand, as in the following code:

```
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();
    }
}
```

You could create additional validations. This is a very clean and elegant way to implement your command validations.

In a similar way, you could implement other behaviors for additional aspects or cross-cutting concerns that you want to apply to commands when handling them.

Additional resources

The mediator pattern

- **Mediator pattern**
  https://en.wikipedia.org/wiki/Mediator_pattern

The decorator pattern

- **Decorator pattern**
  https://en.wikipedia.org/wiki/Decorator_pattern

MediatR (Jimmy Bogard)

- **MediatR.** GitHub repo.
  https://github.com/jbogard/MediatR
- **CQRS with MediatR and AutoMapper**
  https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/
- **Put your controllers on a diet: POSTs and commands.**
  https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/

- **Tackling cross-cutting concerns with a mediator pipeline**
  https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/
- **CQRS and REST: the perfect match**
  https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/
- **MediatR Pipeline Examples**
  https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/
- **Vertical Slice Test Fixtures for MediatR and ASP.NET Core**
  https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/
- **MediatR Extensions for Microsoft Dependency Injection Released**
  https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/

Fluent validation

- **Jeremy Skinner. FluentValidation.** GitHub repo.
  https://github.com/JeremySkinner/FluentValidation

Implement resilient applications

Your microservice and cloud-based applications must embrace the partial failures that will certainly occur eventually. You must design your application to be resilient to those partial failures.

Resiliency is the ability to recover from failures and continue to function. It isn't about avoiding failures but accepting the fact that failures will happen and responding to them in a way that avoids downtime or data loss. The goal of resiliency is to return the application to a fully functioning state after a failure.

It's challenging enough to design and deploy a microservices-based application. But you also need to keep your application running in an environment where some sort of failure is certain. Therefore, your application should be resilient. It should be designed to cope with partial failures, like network outages or nodes or VMs crashing in the cloud. Even microservices (containers) being moved to a different node within a cluster can cause intermittent short failures within the application.

The many individual components of your application should also incorporate health monitoring features. By following the guidelines in this chapter, you can create an application that can work smoothly in spite of transient downtime or the normal hiccups that occur in complex and cloud-based deployments.

Important

eShopOnContainer had been using the Polly library to implement resiliency using Typed Clients up until the release 3.0.0.

Starting with release 3.0.0, the HTTP calls resiliency is implemented using a Linkerd mesh, that handles retries in a transparent and configurable fashion, within a Kubernetes cluster, without having to handle those concerns in the code.

The Polly library is still used to add resilience to database connections, specially while starting up the services.

Warning

All code samples and images in this section were valid before using Linkerd and are not updated to reflect the current actual code. So they make sense in the context of this section.

Handle partial failure

In distributed systems like microservices-based applications, there's an ever-present risk of partial failure. For instance, a single microservice/container can fail or might not be available to respond for a short time, or a single VM or server can crash. Since clients and services are separate processes, a service might not be able to respond in a timely way to a client's request. The service might be overloaded and responding very slowly to requests or might simply not be accessible for a short time because of network issues.

For example, consider the Order details page from the eShopOnContainers sample application. If the ordering microservice is unresponsive when the user tries to submit an order, a bad implementation of the client process (the MVC web application)—for example, if the client code were to use synchronous RPCs with no timeout—would block threads indefinitely waiting for a response. Besides creating a bad user experience, every unresponsive wait consumes or blocks a thread, and threads are extremely valuable in highly scalable applications. If there are many blocked threads, eventually the application's runtime can run out of threads. In that case, the application can become globally unresponsive instead of just partially unresponsive, as shown in Figure 8-1.

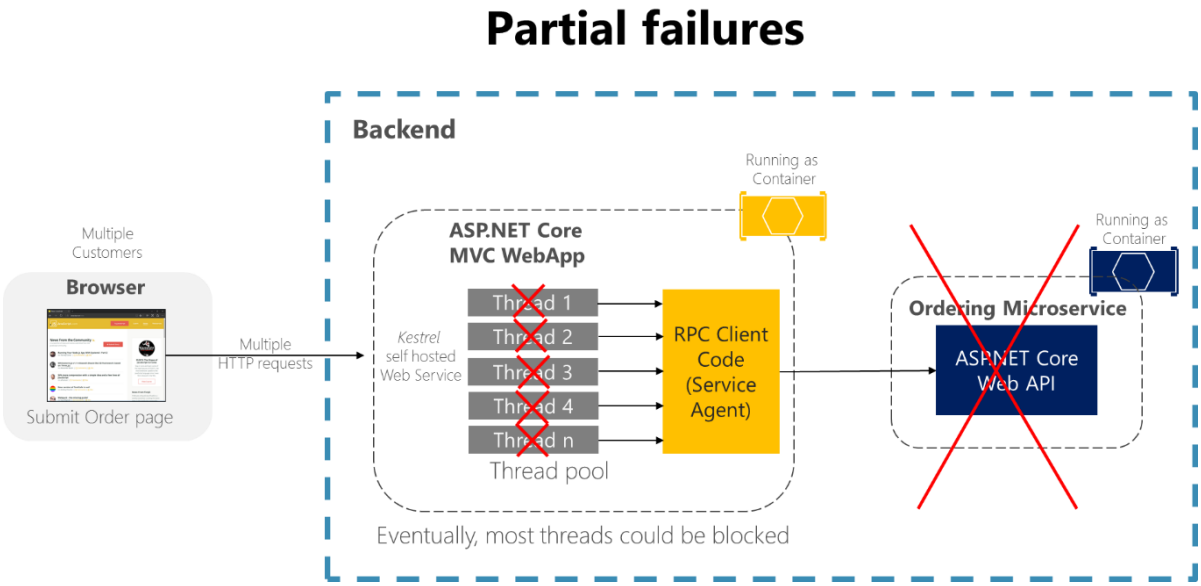


Figure 8-1. Partial failures because of dependencies that impact service thread availability

In a large microservices-based application, any partial failure can be amplified, especially if most of the internal microservices interaction is based on synchronous HTTP calls (which is considered an anti-pattern). Think about a system that receives millions of incoming calls per day. If your system has a bad design that's based on long chains of synchronous HTTP calls, these incoming calls might result in many more millions of outgoing calls (let's suppose a ratio of 1:4) to dozens of internal microservices as synchronous dependencies. This situation is shown in Figure 8-2, especially dependency #3, that starts a chain, calling dependency #4, which then calls #5.

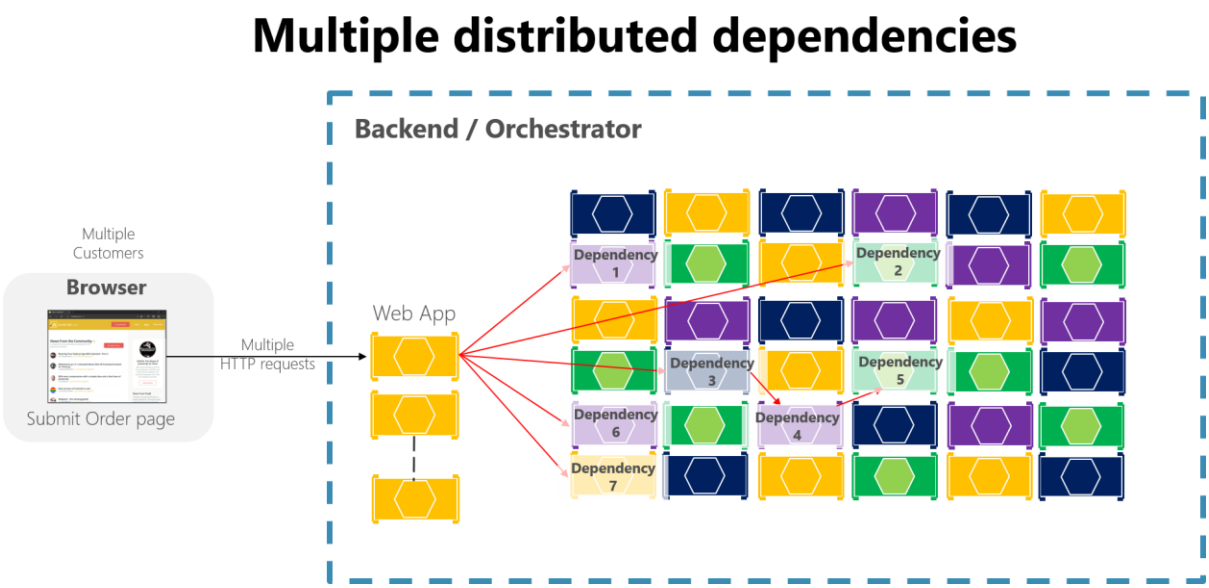


Figure 8-2. The impact of having an incorrect design featuring long chains of HTTP requests

Intermittent failure is guaranteed in a distributed and cloud-based system, even if every dependency itself has excellent availability. It's a fact you need to consider.

If you do not design and implement techniques to ensure fault tolerance, even small downtimes can be amplified. As an example, 50 dependencies each with 99.99% of availability would result in several hours of downtime each month because of this ripple effect. When a microservice dependency fails while handling a high volume of requests, that failure can quickly saturate all available request threads in each service and crash the whole application.

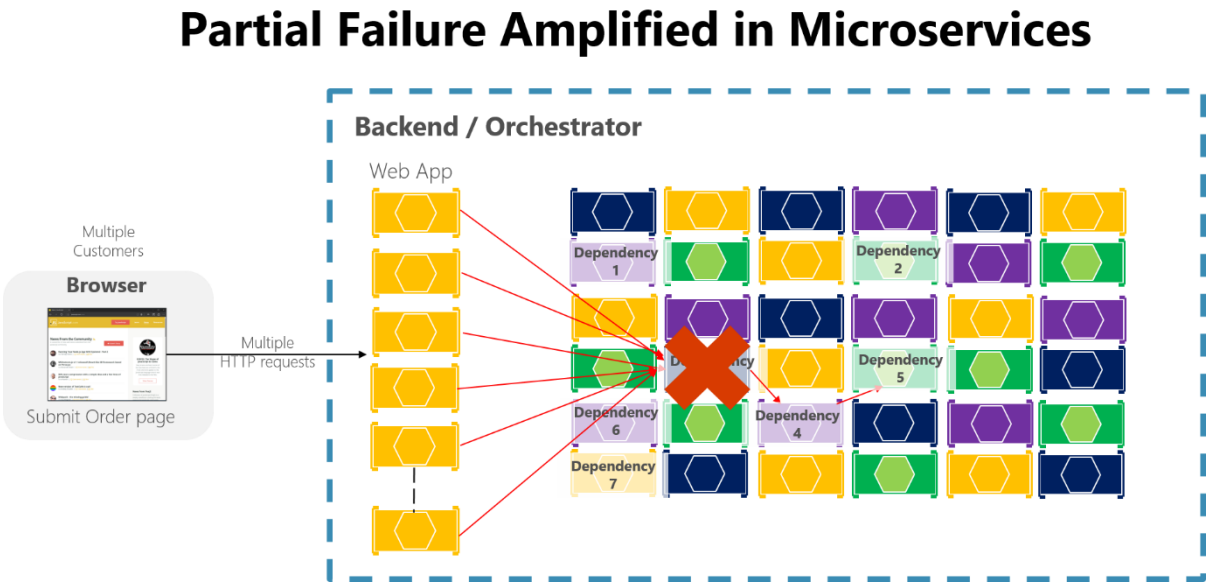


Figure 8-3. Partial failure amplified by microservices with long chains of synchronous HTTP calls

To minimize this problem, in the section Asynchronous microservice integration enforce microservice's autonomy, this guide encourages you to use asynchronous communication across the internal microservices.

In addition, it's essential that you design your microservices and client applications to handle partial failures—that is, to build resilient microservices and client applications.

Strategies to handle partial failure

To deal with partial failures, use one of the strategies described here.

Use asynchronous communication (for example, message-based communication) across internal microservices. It's highly advisable not to create long chains of synchronous HTTP calls across the internal microservices because that incorrect design will eventually become the main cause of bad outages. On the contrary, except for the front-end communications between the client applications and the first level of microservices or fine-grained API Gateways, it's recommended to use only asynchronous (message-based) communication once past the initial request/response cycle, across the internal microservices. Eventual consistency and event-driven architectures will help to minimize ripple effects. These approaches enforce a higher level of microservice autonomy and therefore prevent against the problem noted here.

Use retries with exponential backoff. This technique helps to avoid short and intermittent failures by performing call retries a certain number of times, in case the service was not available only for a short time. This might occur due to intermittent network issues or when a microservice/container is moved to a different node in a cluster. However, if these retries are not designed properly with circuit breakers, it can aggravate the ripple effects, ultimately even causing a Denial of Service (DoS).

Work around network timeouts. In general, clients should be designed not to block indefinitely and to always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.

Use the Circuit Breaker pattern. In this approach, the client process tracks the number of failed requests. If the error rate exceeds a configured limit, a "circuit breaker" trips so that further attempts fail immediately. (If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless.) After a timeout period, the client should try again and, if the new requests are successful, close the circuit breaker.

Provide fallbacks. In this approach, the client process performs fallback logic when a request fails, such as returning cached data or a default value. This is an approach suitable for queries, and is more complex for updates or commands.

Limit the number of queued requests. Clients should also impose an upper bound on the number of outstanding requests that a client microservice can send to a particular service. If the limit has been reached, it's probably pointless to make additional requests, and those attempts should fail immediately. In terms of implementation, the Polly Bulkhead Isolation policy can be used to fulfill this requirement. This approach is essentially a parallelization throttle with SemaphoreSlim as the implementation. It also permits a "queue" outside the bulkhead. You can proactively shed excess load even before execution (for example, because capacity is deemed full). This makes its response to

certain failure scenarios faster than a circuit breaker would be, since the circuit breaker waits for the failures. The BulkheadPolicy object in Polly exposes how full the bulkhead and queue are, and offers events on overflow so can also be used to drive automated horizontal scaling.

Additional resources

- **Resiliency patterns**
  https://learn.microsoft.com/azure/architecture/framework/resiliency/reliability-patterns
- **Adding Resilience and Optimizing Performance**
  https://learn.microsoft.com/previous-versions/msp-n-p/jj591574(v=pandp.10)
- **Bulkhead.** GitHub repo. Implementation with Polly policy.
  https://github.com/App-vNext/Polly/wiki/Bulkhead
- **Designing resilient applications for Azure**
  https://learn.microsoft.com/azure/architecture/framework/resiliency/app-design
- **Transient fault handling**
  https://learn.microsoft.com/azure/architecture/best-practices/transient-faults

Implement retries with exponential backoff

Retries with exponential backoff is a technique that retries an operation, with an exponentially increasing wait time, up to a maximum retry count has been reached (the exponential backoff). This technique embraces the fact that cloud resources might intermittently be unavailable for more than a few seconds for any reason. For example, an orchestrator might be moving a container to another node in a cluster for load balancing. During that time, some requests might fail. Another example could be a database like SQL Azure, where a database can be moved to another server for load balancing, causing the database to be unavailable for a few seconds.

There are many approaches to implement retries logic with exponential backoff.

Implement resilient Entity Framework Core SQL connections

For Azure SQL DB, Entity Framework (EF) Core already provides internal database connection resiliency and retry logic. But you need to enable the Entity Framework execution strategy for each DbContext connection if you want to have resilient EF Core connections.

For instance, the following code at the EF Core connection level enables resilient SQL connections that are retried if the connection fails.

```
// Program.cs from any ASP.NET Core Web API
// Other code ...
builder.Services.AddDbContext<CatalogContext>(options =>
    {
```

```
        options.UseSqlServer(builder.Configuration["ConnectionString"],
        sqlServerOptionsAction: sqlOptions =>
        {
            sqlOptions.EnableRetryOnFailure(
            maxRetryCount: 10,
            maxRetryDelay: TimeSpan.FromSeconds(30),
            errorNumbersToAdd: null);
        });
    });
```

Execution strategies and explicit transactions using BeginTransaction and multiple DbContexts

When retries are enabled in EF Core connections, each operation you perform using EF Core becomes its own retryable operation. Each query and each call to SaveChanges will be retried as a unit if a transient failure occurs.

However, if your code initiates a transaction using BeginTransaction, you're defining your own group of operations that need to be treated as a unit. Everything inside the transaction has to be rolled back if a failure occurs.

If you try to execute that transaction when using an EF execution strategy (retry policy) and you call SaveChanges from multiple DbContexts, you'll get an exception like this one:

System.InvalidOperationException: The configured execution strategy 'SqlServerRetryingExecutionStrategy' does not support user initiated transactions. Use the execution strategy returned by 'DbContext.Database.CreateExecutionStrategy()' to execute all the operations in the transaction as a retriable unit.

The solution is to manually invoke the EF execution strategy with a delegate representing everything that needs to be executed. If a transient failure occurs, the execution strategy will invoke the delegate again. For example, the following code shows how it's implemented in eShopOnContainers with two multiple DbContexts (_catalogContext and the IntegrationEventLogContext) when updating a product and then saving the ProductPriceChangedIntegrationEvent object, which needs to use a different DbContext.

```
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();
    }
}
```

The first DbContext is _catalogContext and the second DbContext is within the _catalogIntegrationEventService object. The Commit action is performed across all DbContext objects using an EF execution strategy.

To achieve this multiple DbContext commit, the SaveEventAndCatalogContextChangesAsync uses a ResilientTransaction class, as shown in the following code:

```
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://learn.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());
        });
    }
}
```

The ResilientTransaction.ExecuteAsync method basically begins a transaction from the passed DbContext (_catalogContext) and then makes the EventLogService use that transaction to save changes from the IntegrationEventLogContext and then commits the whole transaction.

```
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://learn.microsoft.com/ef/core/miscellaneous/connection-resiliency
        var strategy = _context.Database.CreateExecutionStrategy();
        await strategy.ExecuteAsync(async () =>
        {
            await using var transaction = await _context.Database.BeginTransactionAsync();
            await action();
            await transaction.CommitAsync();
        });
    }
}
```

Additional resources

- **Connection Resiliency and Command Interception with EF in an ASP.NET MVC Application**
  https://learn.microsoft.com/aspnet/mvc/overview/getting-started/getting-started-with-ef-using-mvc/connection-resiliency-and-command-interception-with-the-entity-framework-in-an-asp-net-mvc-application

- **Cesar de la Torre. Using Resilient Entity Framework Core SQL Connections and Transactions**
  https://devblogs.microsoft.com/cesardelatorre/using-resilient-entity-framework-core-sql-connections-and-transactions-retries-with-exponential-backoff/

Use IHttpClientFactory to implement resilient HTTP requests

IHttpClientFactory is a contract implemented by DefaultHttpClientFactory, an opinionated factory, available since .NET Core 2.1, for creating HttpClient instances to be used in your applications.

Issues with the original HttpClient class available in .NET

The original and well-known HttpClient class can be easily used, but in some cases, it isn't being properly used by many developers.

Though this class implements IDisposable, declaring and instantiating it within a using statement is not preferred because when the HttpClient object gets disposed of, the underlying socket is not immediately released, which can lead to a *socket exhaustion* problem. For more information about this issue, see the blog post You're using HttpClient wrong and it's destabilizing your software.

Therefore, HttpClient is intended to be instantiated once and reused throughout the life of an application. Instantiating an HttpClient class for every request will exhaust the number of sockets available under heavy loads. That issue will result in SocketException errors. Possible approaches to solve that problem are based on the creation of the HttpClient object as singleton or static, as explained in this Microsoft article on HttpClient usage. This can be a good solution for short-lived console apps or similar, that run a few times a day.

Another issue that developers run into is when using a shared instance of HttpClient in long-running processes. In a situation where the HttpClient is instantiated as a singleton or a static object, it fails to handle the DNS changes as described in this issue of the dotnet/runtime GitHub repository.

However, the issue isn't really with HttpClient per se, but with the default constructor for HttpClient, because it creates a new concrete instance of HttpMessageHandler, which is the one that has *sockets exhaustion* and DNS changes issues mentioned above.

To address the issues mentioned above and to make HttpClient instances manageable, .NET Core 2.1 introduced two approaches, one of them being IHttpClientFactory. It's an interface that's used to configure and create HttpClient instances in an app through Dependency Injection (DI). It also provides extensions for Polly-based middleware to take advantage of delegating handlers in HttpClient.

The alternative is to use SocketsHttpHandler with configured PooledConnectionLifetime. This approach is applied to long-lived, static or singleton HttpClient instances. To learn more about different strategies, see HttpClient guidelines for .NET.

Polly is a transient-fault-handling library that helps developers add resiliency to their applications, by using some pre-defined policies in a fluent and thread-safe manner.

Benefits of using IHttpClientFactory

The current implementation of IHttpClientFactory, that also implements IHttpMessageHandlerFactory, offers the following benefits:

- Provides a central location for naming and configuring logical HttpClient objects. For example, you may configure a client (Service Agent) that's pre-configured to access a specific microservice.
- Codify the concept of outgoing middleware via delegating handlers in HttpClient and implementing Polly-based middleware to take advantage of Polly's policies for resiliency.
- HttpClient already has the concept of delegating handlers that could be linked together for outgoing HTTP requests. You can register HTTP clients into the factory and you can use a Polly handler to use Polly policies for Retry, CircuitBreakers, and so on.
- Manage the lifetime of HttpMessageHandler to avoid the mentioned problems/issues that can occur when managing HttpClient lifetimes yourself.

Tip

The HttpClient instances injected by DI can be disposed of safely, because the associated HttpMessageHandler is managed by the factory. Injected HttpClient instances are *Transient* from a DI perspective, while HttpMessageHandler instances can be regarded as *Scoped*. HttpMessageHandler instances have their own DI scopes, **separate** from the application scopes (for example, ASP.NET incoming request scopes). For more information, see Using HttpClientFactory in .NET.

Note

The implementation of IHttpClientFactory (DefaultHttpClientFactory) is tightly tied to the DI implementation in the Microsoft.Extensions.DependencyInjection NuGet package. If you need to use HttpClient without DI or with other DI implementations, consider using a static or singleton HttpClient with PooledConnectionLifetime set up. For more information, see HttpClient guidelines for .NET.

Multiple ways to use IHttpClientFactory

There are several ways that you can use IHttpClientFactory in your application:

- Basic usage
- Use Named Clients
- Use Typed Clients
- Use Generated Clients

For the sake of brevity, this guidance shows the most structured way to use IHttpClientFactory, which is to use Typed Clients (Service Agent pattern). However, all options are documented and are currently listed in this article covering the IHttpClientFactory usage.

Note

If your app requires cookies, it might be better to avoid using IHttpClientFactory in your app. For alternative ways of managing clients, see Guidelines for using HTTP clients

How to use Typed Clients with IHttpClientFactory

So, what's a "Typed Client"? It's just an HttpClient that's pre-configured for some specific use. This configuration can include specific values such as the base server, HTTP headers or time outs.

The following diagram shows how Typed Clients are used with IHttpClientFactory:

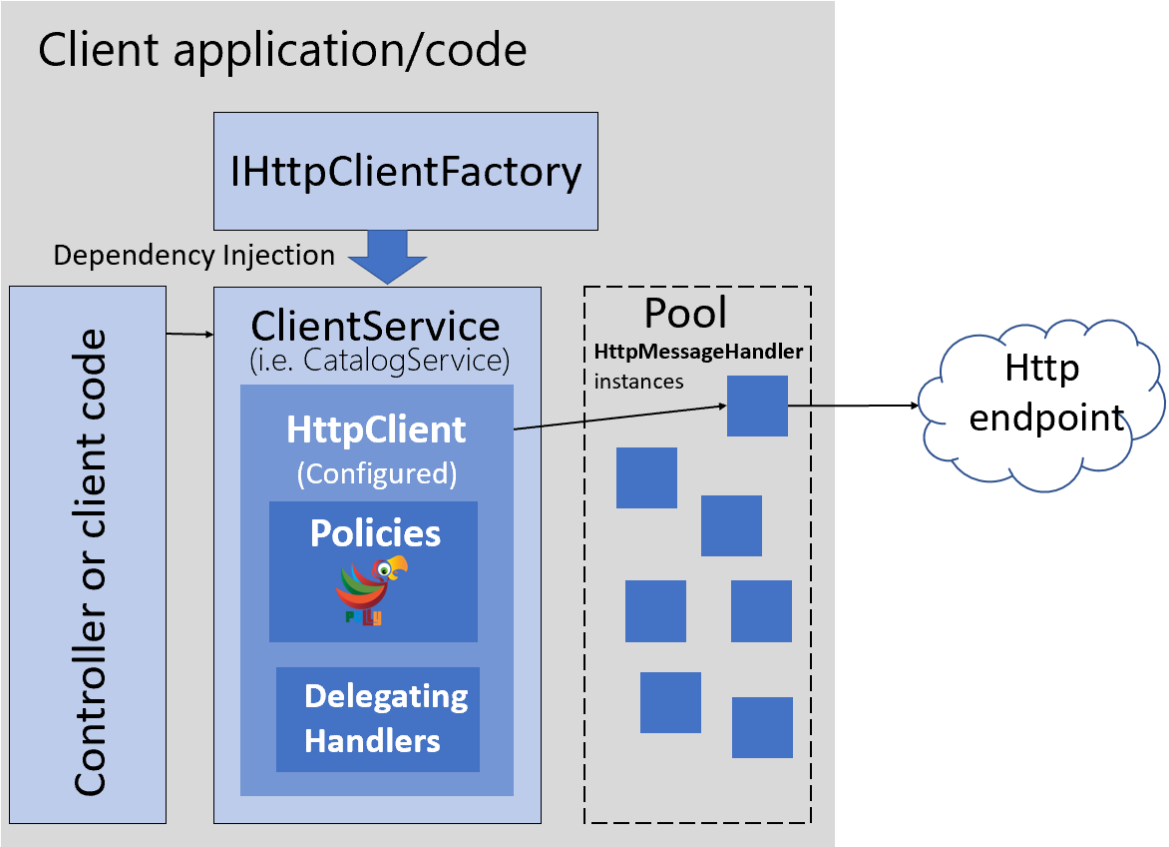


Figure 8-4. Using IHttpClientFactory with Typed Client classes.

In the above image, a ClientService (used by a controller or client code) uses an HttpClient created by the registered IHttpClientFactory. This factory assigns an HttpMessageHandler from a pool to the HttpClient. The HttpClient can be configured with Polly's policies when registering the IHttpClientFactory in the DI container with the extension method AddHttpClient.

To configure the above structure, add IHttpClientFactory in your application by installing the Microsoft.Extensions.Http NuGet package that includes the AddHttpClient extension method for IServiceCollection. This extension method registers the internal DefaultHttpClientFactory class to be used as a singleton for the interface IHttpClientFactory. It defines a transient configuration for the HttpMessageHandlerBuilder. This message handler (HttpMessageHandler object), taken from a pool, is used by the HttpClient returned from the factory.

In the next snippet, you can see how AddHttpClient() can be used to register Typed Clients (Service Agents) that need to use HttpClient.

```
// Program.cs
//Add http client services at ConfigureServices(IServiceCollection services)
builder.Services.AddHttpClient<ICatalogService, CatalogService>();
builder.Services.AddHttpClient<IBasketService, BasketService>();
builder.Services.AddHttpClient<IOrderingService, OrderingService>();
```

Registering the client services as shown in the previous snippet, makes the DefaultClientFactory create a standard HttpClient for each service. The typed client is registered as transient with DI container. In the preceding code, AddHttpClient() registers *CatalogService*, *BasketService*, *OrderingService* as transient services so they can be injected and consumed directly without any need for additional registrations.

You could also add instance-specific configuration in the registration to, for example, configure the base address, and add some resiliency policies, as shown in the following:

```
builder.Services.AddHttpClient<ICatalogService, CatalogService>(client =>
{
    client.BaseAddress = new Uri(builder.Configuration["BaseUrl"]);
})
    .AddPolicyHandler(GetRetryPolicy())
    .AddPolicyHandler(GetCircuitBreakerPolicy());
```

In this next example, you can see the configuration of one of the above policies:

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
retryAttempt)));
}
```

You can find more details about using Polly in the Next article.

HttpClient lifetimes

Each time you get an HttpClient object from the IHttpClientFactory, a new instance is returned. But each HttpClient uses an HttpMessageHandler that's pooled and reused by the IHttpClientFactory to reduce resource consumption, as long as the HttpMessageHandler's lifetime hasn't expired.

Pooling of handlers is desirable as each handler typically manages its own underlying HTTP connections; creating more handlers than necessary can result in connection delays. Some handlers also keep connections open indefinitely, which can prevent the handler from reacting to DNS changes.

The HttpMessageHandler objects in the pool have a lifetime that's the length of time that an HttpMessageHandler instance in the pool can be reused. The default value is two minutes, but it can be overridden per Typed Client. To override it, call SetHandlerLifetime() on the IHttpClientBuilder that's returned when creating the client, as shown in the following code:

```
//Set 5 min as the lifetime for the HttpMessageHandler objects in the pool used for the
Catalog Typed Client
builder.Services.AddHttpClient<ICatalogService, CatalogService>()
    .SetHandlerLifetime(TimeSpan.FromMinutes(5));
```

Each Typed Client can have its own configured handler lifetime value. Set the lifetime to InfiniteTimeSpan to disable handler expiry.

Implement your Typed Client classes that use the injected and configured HttpClient

As a previous step, you need to have your Typed Client classes defined, such as the classes in the sample code, like 'BasketService', 'CatalogService', 'OrderingService', etc. – A Typed Client is a class that accepts an HttpClient object (injected through its constructor) and uses it to call some remote HTTP service. For example:

```
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;
    }
}
```

The Typed Client (CatalogService in the example) is activated by DI (Dependency Injection), which means it can accept any registered service in its constructor, in addition to HttpClient.

A Typed Client is effectively a transient object, that means a new instance is created each time one is needed. It receives a new HttpClient instance each time it's constructed. However, the HttpMessageHandler objects in the pool are the objects that are reused by multiple HttpClient instances.

Use your Typed Client classes

Finally, once you have your typed classes implemented, you can have them registered and configured with AddHttpClient(). After that you can use them wherever services are injected by DI, such as in Razor page code or an MVC web app controller, shown in the below code from 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
        }

        }
}
```

Up to this point, the above code snippet only shows the example of performing regular HTTP requests. But the 'magic' comes in the following sections where it shows how all the HTTP requests made by HttpClient can have resilient policies such as retries with exponential backoff, circuit breakers, security features using auth tokens, or even any other custom feature. And all of these can be done just by adding policies and delegating handlers to your registered Typed Clients.

Additional resources

- **HttpClient guidelines for .NET**
  https://learn.microsoft.com/en-us/dotnet/fundamentals/networking/http/httpclient-guidelines
- **Using HttpClientFactory in .NET**
  https://learn.microsoft.com/en-us/dotnet/core/extensions/httpclient-factory
- **Using HttpClientFactory in ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/fundamentals/http-requests
- **HttpClientFactory source code in the dotnet/runtime GitHub repository**
  https://github.com/dotnet/runtime/tree/release/7.0/src/libraries/Microsoft.Extensions.Http/
- **Polly (.NET resilience and transient-fault-handling library)**
  https://thepollyproject.azurewebsites.net/

Implement HTTP call retries with exponential backoff with IHttpClientFactory and Polly policies

The recommended approach for retries with exponential backoff is to take advantage of more advanced .NET libraries like the open-source Polly library.

Polly is a .NET library that provides resilience and transient-fault handling capabilities. You can implement those capabilities by applying Polly policies such as Retry, Circuit Breaker, Bulkhead Isolation, Timeout, and Fallback. Polly targets .NET Framework 4.x and .NET Standard 1.0, 1.1, and 2.0 (which supports .NET Core and later).

The following steps show how you can use Http retries with Polly integrated into IHttpClientFactory, which is explained in the previous section.

Install .NET packages

First, you will need to install the Microsoft.Extensions.Http.Polly package.

- Install with Visual Studio
- Install with dotnet CLI
- Install with nuget.exe CLI
- Install with Package Manager Console (PowerShell)

Reference the .NET 7 packages

IHttpClientFactory is available since .NET Core 2.1, however, we recommend you use the latest .NET 7 packages from NuGet in your project. You typically also need to reference the extension package Microsoft.Extensions.Http.Polly.

Configure a client with Polly's Retry policy, in app startup

The **AddPolicyHandler()** method is what adds policies to the HttpClient objects you'll use. In this case, it's adding a Polly's policy for Http Retries with exponential backoff.

To have a more modular approach, the Http Retry Policy can be defined in a separate method within the *Program.cs* file, as shown in the following code:

```
static IAsyncPolicy<HttpResponseMessage> GetRetryPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .OrResult(msg => msg.StatusCode == System.Net.HttpStatusCode.NotFound)
        .WaitAndRetryAsync(6, retryAttempt => TimeSpan.FromSeconds(Math.Pow(2,
                                                                    retryAttempt)));
}
```

As shown in previous sections, you need to define a named or typed client HttpClient configuration in your standard *Program.cs* app configuration. Now you add incremental code specifying the policy for the Http retries with exponential backoff, as follows:

```
// Program.cs
builder.Services.AddHttpClient<IBasketService, BasketService>()
        .SetHandlerLifetime(TimeSpan.FromMinutes(5))  //Set lifetime to five minutes
        .AddPolicyHandler(GetRetryPolicy());
```

With Polly, you can define a Retry policy with the number of retries, the exponential backoff configuration, and the actions to take when there's an HTTP exception, such as logging the error. In this case, the policy is configured to try six times with an exponential retry, starting at two seconds.

Add a jitter strategy to the retry policy

A regular Retry policy can affect your system in cases of high concurrency and scalability and under high contention. To overcome peaks of similar retries coming from many clients in partial outages, a good workaround is to add a jitter strategy to the retry algorithm/policy. This strategy can improve the overall performance of the end-to-end system. As recommended in Polly: Retry with Jitter, a good

jitter strategy can be implemented by smooth and evenly distributed retry intervals applied with a well-controlled median initial retry delay on an exponential backoff. This approach helps to spread out the spikes when the issue arises. The principle is illustrated by the following example:

```
var delay = Backoff.DecorrelatedJitterBackoffV2(medianFirstRetryDelay:
TimeSpan.FromSeconds(1), retryCount: 5);

var retryPolicy = Policy
    .Handle<FooException>()
    .WaitAndRetryAsync(delay);
```

Additional resources

- **Retry pattern** https://learn.microsoft.com/azure/architecture/patterns/retry
- **Polly and IHttpClientFactory** https://github.com/App-vNext/Polly/wiki/Polly-and-HttpClientFactory
- **Polly (.NET resilience and transient-fault-handling library)** https://github.com/App-vNext/Polly
- **Polly: Retry with Jitter** https://github.com/App-vNext/Polly/wiki/Retry-with-jitter
- **Marc Brooker. Jitter: Making Things Better With Randomness** https://brooker.co.za/blog/2015/03/21/backoff.html

Implement the Circuit Breaker pattern

As noted earlier, you should handle faults that might take a variable amount of time to recover from, as might happen when you try to connect to a remote service or resource. Handling this type of fault can improve the stability and resiliency of an application.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections and timeouts, or if resources are responding slowly or are temporarily unavailable. These faults typically correct themselves after a short time, and a robust cloud application should be prepared to handle them by using a strategy like the "Retry pattern".

However, there can also be situations where faults are due to unanticipated events that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations, it might be pointless for an application to continually retry an operation that's unlikely to succeed.

Instead, the application should be coded to accept that the operation has failed and handle the failure accordingly.

Using Http retries carelessly could result in creating a Denial of Service (DoS) attack within your own software. As a microservice fails or performs slowly, multiple clients might repeatedly retry failed requests. That creates a dangerous risk of exponentially increasing traffic targeted at the failing service.

Therefore, you need some kind of defense barrier so that excessive requests stop when it isn't worth to keep trying. That defense barrier is precisely the circuit breaker.

The Circuit Breaker pattern has a different purpose than the "Retry pattern". The "Retry pattern" enables an application to retry an operation in the expectation that the operation will eventually succeed. The Circuit Breaker pattern prevents an application from performing an operation that's likely to fail. An application can combine these two patterns. However, the retry logic should be sensitive to any exception returned by the circuit breaker, and it should abandon retry attempts if the circuit breaker indicates that a fault is not transient.

Implement Circuit Breaker pattern with IHttpClientFactory and Polly

As when implementing retries, the recommended approach for circuit breakers is to take advantage of proven .NET libraries like Polly and its native integration with IHttpClientFactory.

Adding a circuit breaker policy into your IHttpClientFactory outgoing middleware pipeline is as simple as adding a single incremental piece of code to what you already have when using IHttpClientFactory.

The only addition here to the code used for HTTP call retries is the code where you add the Circuit Breaker policy to the list of policies to use, as shown in the following incremental code.

```
// Program.cs
var retryPolicy = GetRetryPolicy();
var circuitBreakerPolicy = GetCircuitBreakerPolicy();

builder.Services.AddHttpClient<IBasketService, BasketService>()
        .SetHandlerLifetime(TimeSpan.FromMinutes(5))  // Sample: default lifetime is 2
minutes
        .AddHttpMessageHandler<HttpClientAuthorizationDelegatingHandler>()
        .AddPolicyHandler(retryPolicy)
        .AddPolicyHandler(circuitBreakerPolicy);
```

The AddPolicyHandler() method is what adds policies to the HttpClient objects you'll use. In this case, it's adding a Polly policy for a circuit breaker.

To have a more modular approach, the Circuit Breaker Policy is defined in a separate method called GetCircuitBreakerPolicy(), as shown in the following code:

```
// also in Program.cs
static IAsyncPolicy<HttpResponseMessage> GetCircuitBreakerPolicy()
{
    return HttpPolicyExtensions
        .HandleTransientHttpError()
        .CircuitBreakerAsync(5, TimeSpan.FromSeconds(30));
}
```

In the code example above, the circuit breaker policy is configured so it breaks or opens the circuit when there have been five consecutive faults when retrying the Http requests. When that happens, the circuit will break for 30 seconds: in that period, calls will be failed immediately by the circuit-breaker rather than actually be placed. The policy automatically interprets relevant exceptions and HTTP status codes as faults.

Circuit breakers should also be used to redirect requests to a fallback infrastructure if you had issues in a particular resource that's deployed in a different environment than the client application or

service that's performing the HTTP call. That way, if there's an outage in the datacenter that impacts only your backend microservices but not your client applications, the client applications can redirect to the fallback services. Polly is planning a new policy to automate this failover policy scenario.

All those features are for cases where you're managing the failover from within the .NET code, as opposed to having it managed automatically for you by Azure, with location transparency.

From a usage point of view, when using HttpClient, there's no need to add anything new here because the code is the same than when using HttpClient with IHttpClientFactory, as shown in previous sections.

Test Http retries and circuit breakers in eShopOnContainers

Whenever you start the eShopOnContainers solution in a Docker host, it needs to start multiple containers. Some of the containers are slower to start and initialize, like the SQL Server container. This is especially true the first time you deploy the eShopOnContainers application into Docker because it needs to set up the images and the database. The fact that some containers start slower than others can cause the rest of the services to initially throw HTTP exceptions, even if you set dependencies between containers at the docker-compose level, as explained in previous sections. Those docker-compose dependencies between containers are just at the process level. The container's entry point process might be started, but SQL Server might not be ready for queries. The result can be a cascade of errors, and the application can get an exception when trying to consume that particular container.

You might also see this type of error on startup when the application is deploying to the cloud. In that case, orchestrators might be moving containers from one node or VM to another (that is, starting new instances) when balancing the number of containers across the cluster's nodes.

The way 'eShopOnContainers' solves those issues when starting all the containers is by using the Retry pattern illustrated earlier.

Test the circuit breaker in eShopOnContainers

There are a few ways you can break/open the circuit and test it with eShopOnContainers.

One option is to lower the allowed number of retries to 1 in the circuit breaker policy and redeploy the whole solution into Docker. With a single retry, there's a good chance that an HTTP request will fail during deployment, the circuit breaker will open, and you get an error.

Another option is to use custom middleware that's implemented in the **Basket** microservice. When this middleware is enabled, it catches all HTTP requests and returns status code 500. You can enable the middleware by making a GET request to the failing URI, like the following:

- GET http://localhost:5103/failing
  This request returns the current state of the middleware. If the middleware is enabled, the request return status code 500. If the middleware is disabled, there's no response.
- GET http://localhost:5103/failing?enable
  This request enables the middleware.

- GET http://localhost:5103/failing?disable
  This request disables the middleware.

For instance, once the application is running, you can enable the middleware by making a request using the following URI in any browser. Note that the ordering microservice uses port 5103.

http://localhost:5103/failing?enable

You can then check the status using the URI http://localhost:5103/failing, as shown in Figure 8-5.

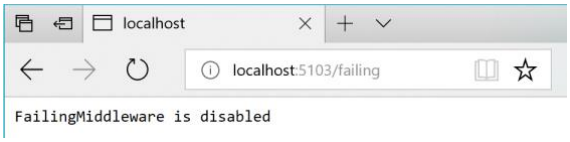


Figure 8-5. Checking the state of the "Failing" ASP.NET middleware – In this case, disabled.

At this point, the Basket microservice responds with status code 500 whenever you call invoke it.

Once the middleware is running, you can try making an order from the MVC web application. Because the requests fail, the circuit will open.

In the following example, you can see that the MVC web application has a catch block in the logic for placing an order. If the code catches an open-circuit exception, it shows the user a friendly message telling them to wait.

```
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)";
    }
}
```

Here's a summary. The Retry policy tries several times to make the HTTP request and gets HTTP errors. When the number of retries reaches the maximum number set for the Circuit Breaker policy (in this case, 5), the application throws a BrokenCircuitException. The result is a friendly message, as shown in Figure 8-6.

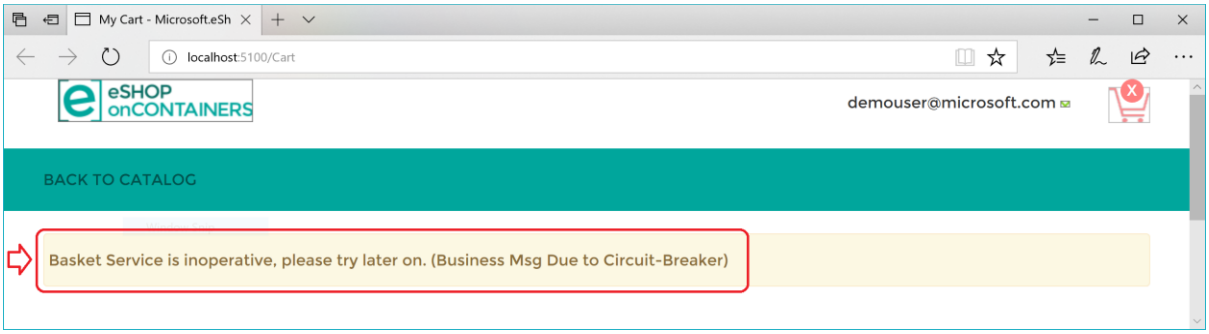


Figure 8-6. Circuit breaker returning an error to the UI

You can implement different logic for when to open/break the circuit. Or you can try an HTTP request against a different back-end microservice if there's a fallback datacenter or redundant back-end system.

Finally, another possibility for the CircuitBreakerPolicy is to use Isolate (which forces open and holds open the circuit) and Reset (which closes it again). These could be used to build a utility HTTP endpoint that invokes Isolate and Reset directly on the policy. Such an HTTP endpoint could also be used, suitably secured, in production for temporarily isolating a downstream system, such as when you want to upgrade it. Or it could trip the circuit manually to protect a downstream system you suspect to be faulting.

Additional resources

- **Circuit Breaker pattern**
  https://learn.microsoft.com/azure/architecture/patterns/circuit-breaker

Health monitoring

Health monitoring can allow near-real-time information about the state of your containers and microservices. Health monitoring is critical to multiple aspects of operating microservices and is especially important when orchestrators perform partial application upgrades in phases, as explained later.

Microservices-based applications often use heartbeats or health checks to enable their performance monitors, schedulers, and orchestrators to keep track of the multitude of services. If services cannot send some sort of "I'm alive" signal, either on demand or on a schedule, your application might face risks when you deploy updates, or it might just detect failures too late and not be able to stop cascading failures that can end up in major outages.

In the typical model, services send reports about their status, and that information is aggregated to provide an overall view of the state of health of your application. If you're using an orchestrator, you can provide health information to your orchestrator's cluster, so that the cluster can act accordingly. If you invest in high-quality health reporting that's customized for your application, you can detect and fix issues for your running application much more easily.

Implement health checks in ASP.NET Core services

When developing an ASP.NET Core microservice or web application, you can use the built-in health checks feature that was released in ASP .NET Core 2.2 (Microsoft.Extensions.Diagnostics.HealthChecks). Like many ASP.NET Core features, health checks come with a set of services and a middleware.

Health check services and middleware are easy to use and provide capabilities that let you validate if any external resource needed for your application (like a SQL Server database or a remote API) is working properly. When you use this feature, you can also decide what it means that the resource is healthy, as we explain later.

To use this feature effectively, you need to first configure services in your microservices. Second, you need a front-end application that queries for the health reports. That front-end application could be a custom reporting application, or it could be an orchestrator itself that can react accordingly to the health states.

Use the HealthChecks feature in your back-end ASP.NET microservices

In this section, you'll learn how to implement the HealthChecks feature in a sample ASP.NET Core 7.0 Web API application when using the Microsoft.Extensions.Diagnostics.HealthChecks package. The Implementation of this feature in a large-scale microservices like the eShopOnContainers is explained in the next section.

To begin, you need to define what constitutes a healthy status for each microservice. In the sample application, we define the microservice is healthy if its API is accessible via HTTP and its related SQL Server database is also available.

In .NET 7, with the built-in APIs, you can configure the services, add a Health Check for the microservice and its dependent SQL Server database in this way:

```
// Program.cs from .NET 7 Web API sample

//...
// Registers required services for health checks
builder.Services.AddHealthChecks()
    // Add a health check for a SQL Server database
    .AddCheck(
        "OrderingDB-check",
        new SqlConnectionHealthCheck(builder.Configuration["ConnectionString"]),
        HealthStatus.Unhealthy,
        new string[] { "orderingdb" });
```

In the previous code, the services.AddHealthChecks() method configures a basic HTTP check that returns a status code **200** with "Healthy". Further, the AddCheck() extension method configures a custom SqlConnectionHealthCheck that checks the related SQL Database's health.

The AddCheck() method adds a new health check with a specified name and the implementation of type IHealthCheck. You can add multiple Health Checks using AddCheck method, so a microservice won't provide a "healthy" status until all its checks are healthy.

SqlConnectionHealthCheck is a custom class that implements IHealthCheck, which takes a connection string as a constructor parameter and executes a simple query to check if the connection to the SQL database is successful. It returns HealthCheckResult.Healthy() if the query was executed successfully and a FailureStatus with the actual exception when it fails.

```
// Sample SQL Connection Health Check
public class SqlConnectionHealthCheck : IHealthCheck
{
    private const 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();
    }
}
```

Note that in the previous code, Select 1 is the query used to check the Health of the database. To monitor the availability of your microservices, orchestrators like Kubernetes periodically perform health checks by sending requests to test the microservices. It's important to keep your database queries efficient so that these operations are quick and don't result in a higher utilization of resources.

Finally, add a middleware that responds to the url path /hc:

```
// Program.cs from .NET 7 Web Api sample

app.MapHealthChecks("/hc");
```

When the endpoint <yourmicroservice>/hc is invoked, it runs all the health checks that are configured in the AddHealthChecks() method in the Startup class and shows the result.

HealthChecks implementation in eShopOnContainers

Microservices in eShopOnContainers rely on multiple services to perform its task. For example, the Catalog.API microservice from eShopOnContainers depends on many services, such as Azure Blob Storage, SQL Server, and RabbitMQ. Therefore, it has several health checks added using the AddCheck() method. For every dependent service, a custom IHealthCheck implementation that defines its respective health status would need to be added.

The open-source project AspNetCore.Diagnostics.HealthChecks solves this problem by providing custom health check implementations for each of these enterprise services, that are built on top of .NET 7. Each health check is available as an individual NuGet package that can be easily added to the project. eShopOnContainers uses them extensively in all its microservices.

For instance, in the Catalog.API microservice, the following NuGet packages were added:

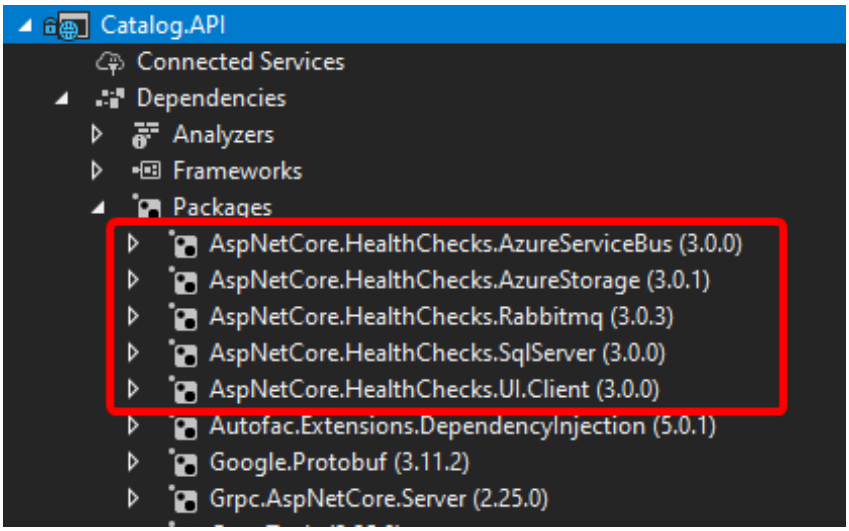


Figure 8-7. Custom Health Checks implemented in Catalog.API using AspNetCore.Diagnostics.HealthChecks

In the following code, the health check implementations are added for each dependent service and then the middleware is configured:

```
// Extension method 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};Endpoint
Suffix=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;
}
```

Finally, add the HealthCheck middleware to listen to "/hc" endpoint:

```
// HealthCheck middleware
app.UseHealthChecks("/hc", new HealthCheckOptions()
{
    Predicate = _ => true,
    ResponseWriter = UIResponseWriter.WriteHealthCheckUIResponse
});
```

Query your microservices to report about their health status

When you've configured health checks as described in this article and you have the microservice running in Docker, you can directly check from a browser if it's healthy. You have to publish the container port in the Docker host, so you can access the container through the external Docker host IP or through host.docker.internal, as shown in figure 8-8.

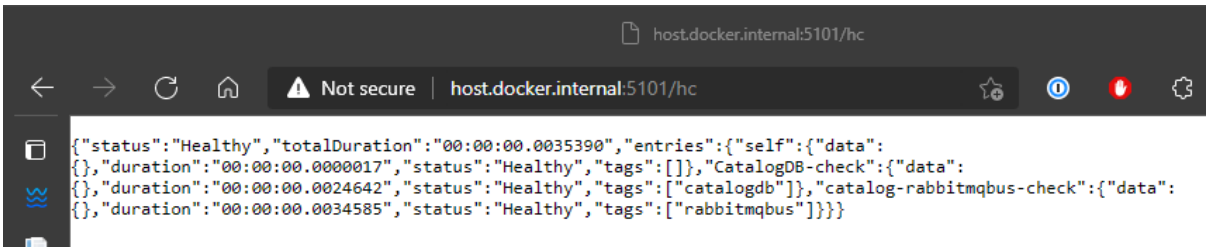


Figure 8-8. Checking health status of a single service from a browser

In that test, you can see that the Catalog.API microservice (running on port 5101) is healthy, returning HTTP status 200 and status information in JSON. The service also checked the health of its SQL Server database dependency and RabbitMQ, so the health check reported itself as healthy.

Use watchdogs

A watchdog is a separate service that can watch health and load across services, and report health about the microservices by querying with the HealthChecks library introduced earlier. This can help prevent errors that would not be detected based on the view of a single service. Watchdogs also are a good place to host code that can perform remediation actions for known conditions without user interaction.

The eShopOnContainers sample contains a web page that displays sample health check reports, as shown in Figure 8-9. This is the simplest watchdog you could have since it only shows the state of the microservices and web applications in eShopOnContainers. Usually a watchdog also takes actions when it detects unhealthy states.

Fortunately, AspNetCore.Diagnostics.HealthChecks also provides AspNetCore.HealthChecks.UI NuGet package that can be used to display the health check results from the configured URIs.

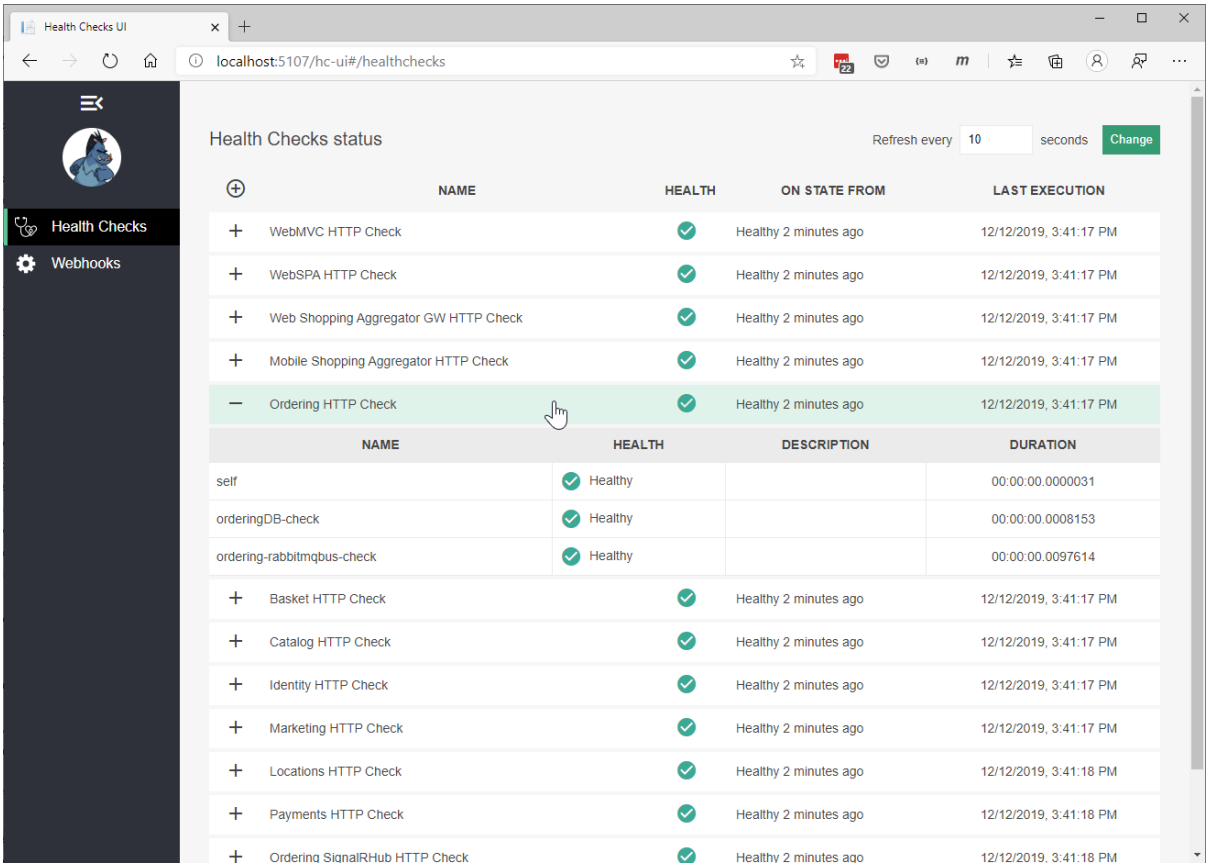


Figure 8-9. Sample health check report in eShopOnContainers

In summary, this watchdog service queries each microservice's "/hc" endpoint. This will execute all the health checks defined within it and return an overall health state depending on all those checks. The HealthChecksUI is easy to consume with a few configuration entries and two lines of code that needs to be added into the *Startup.cs* of the watchdog service.

Sample configuration file for health check UI:

```
// Configuration
{
  "HealthChecksUI": {
    "HealthChecks": [
      {
        "Name": "Ordering HTTP Check",
        "Uri": "http://host.docker.internal:5102/hc"
      },
      {
        "Name": "Ordering HTTP Background Check",
        "Uri": "http://host.docker.internal:5111/hc"
      },
      //...
    ]}
}
```

Program.cs file that adds HealthChecksUI:

```
// Program.cs from WebStatus(Watch Dog) service
//
// Registers required services for health checks
builder.Services.AddHealthChecksUI();
// build the app, register other middleware
app.UseHealthChecksUI(config => config.UIPath = "/hc-ui");
```

Health checks when using orchestrators

To monitor the availability of your microservices, orchestrators like Kubernetes and Service Fabric periodically perform health checks by sending requests to test the microservices. When an orchestrator determines that a service/container is unhealthy, it stops routing requests to that instance. It also usually creates a new instance of that container.

For instance, most orchestrators can use health checks to manage zero-downtime deployments. Only when the status of a service/container changes to healthy will the orchestrator start routing traffic to service/container instances.

Health monitoring is especially important when an orchestrator performs an application upgrade. Some orchestrators (like Azure Service Fabric) update services in phases—for example, they might update one-fifth of the cluster surface for each application upgrade. The set of nodes that's upgraded at the same time is referred to as an *upgrade domain*. After each upgrade domain has been upgraded and is available to users, that upgrade domain must pass health checks before the deployment moves to the next upgrade domain.

Another aspect of service health is reporting metrics from the service. This is an advanced capability of the health model of some orchestrators, like Service Fabric. Metrics are important when using an orchestrator because they are used to balance resource usage. Metrics also can be an indicator of system health. For example, you might have an application that has many microservices, and each instance reports a requests-per-second (RPS) metric. If one service is using more resources (memory, processor, etc.) than another service, the orchestrator could move service instances around in the cluster to try to maintain even resource utilization.

Note that Azure Service Fabric provides its own Health Monitoring model, which is more advanced than simple health checks.

Advanced monitoring: visualization, analysis, and alerts

The final part of monitoring is visualizing the event stream, reporting on service performance, and alerting when an issue is detected. You can use different solutions for this aspect of monitoring.

You can use simple custom applications showing the state of your services, like the custom page shown when explaining the AspNetCore.Diagnostics.HealthChecks. Or you could use more advanced tools like Azure Monitor to raise alerts based on the stream of events.

Finally, if you're storing all the event streams, you can use Microsoft Power BI or other solutions like Kibana or Splunk to visualize the data.

Additional resources

- **HealthChecks and HealthChecks UI for ASP.NET Core**
  https://github.com/Xabaril/AspNetCore.Diagnostics.HealthChecks
- **Introduction to Service Fabric health monitoring**
  https://learn.microsoft.com/azure/service-fabric/service-fabric-health-introduction
- **Azure Monitor**
  https://azure.microsoft.com/services/monitor/

Make secure .NET Microservices and Web Applications

There are so many aspects about security in microservices and web applications that the topic could easily take several books like this one. So, in this section, we'll focus on authentication, authorization, and application secrets.

Implement authentication in .NET microservices and web applications

It's often necessary for resources and APIs published by a service to be limited to certain trusted users or clients. The first step to making these sorts of API-level trust decisions is authentication. Authentication is the process of reliably verifying a user's identity.

In microservice scenarios, authentication is typically handled centrally. If you're using an API Gateway, the gateway is a good place to authenticate, as shown in Figure 9-1. If you use this approach, make sure that the individual microservices cannot be reached directly (without the API Gateway) unless additional security is in place to authenticate messages whether they come from the gateway or not.

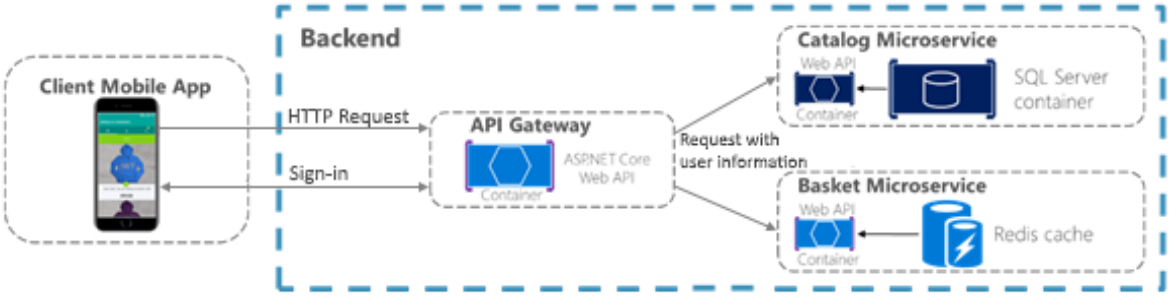


Figure 9-1. Centralized authentication with an API Gateway

When the API Gateway centralizes authentication, it adds user information when forwarding requests to the microservices. If services can be accessed directly, an authentication service like Azure Active

Directory or a dedicated authentication microservice acting as a security token service (STS) can be used to authenticate users. Trust decisions are shared between services with security tokens or cookies. (These tokens can be shared between ASP.NET Core applications, if needed, by implementing cookie sharing.) This pattern is illustrated in Figure 9-2.

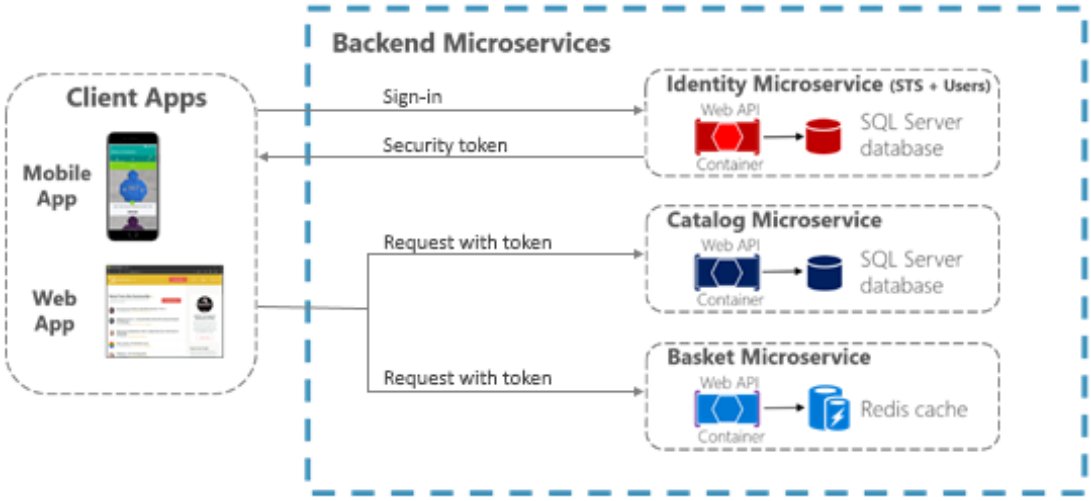


Figure 9-2. Authentication by identity microservice; trust is shared using an authorization token

When microservices are accessed directly, trust, that includes authentication and authorization, is handled by a security token issued by a dedicated microservice, shared between microservices.

Authenticate with ASP.NET Core Identity

The primary mechanism in ASP.NET Core for identifying an application's users is the ASP.NET Core Identity membership system. ASP.NET Core Identity stores user information (including sign-in information, roles, and claims) in a data store configured by the developer. Typically, the ASP.NET Core Identity data store is an Entity Framework store provided in the Microsoft.AspNetCore.Identity.EntityFrameworkCore package. However, custom stores or other third-party packages can be used to store identity information in Azure Table Storage, CosmosDB, or other locations.

Tip

ASP.NET Core 2.1 and later provides ASP.NET Core Identity as a Razor Class Library, so you won't see much of the necessary code in your project, as was the case for previous versions. For details on how to customize the Identity code to suit your needs, see Scaffold Identity in ASP.NET Core projects.

The following code is taken from the ASP.NET Core Web Application MVC 3.1 project template with individual user account authentication selected. It shows how to configure ASP.NET Core Identity using Entity Framework Core in the *Program.cs* file.

```
//...
builder.Services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(
        builder.Configuration.GetConnectionString("DefaultConnection")));
```

```
builder.Services.AddDefaultIdentity<IdentityUser>(options =>
    options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<ApplicationDbContext>();

builder.Services.AddRazorPages();
//...
Once ASP.NET Core Identity is configured, you enable it by adding the
app.UseAuthentication() and endpoints.MapRazorPages() as shown in the following code in the
service's Program.cs file:
//...
app.UseRouting();

app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapRazorPages();
});
//...
```

Important

The lines in the preceding code **MUST BE IN THE ORDER SHOWN** for Identity to work correctly.

Using ASP.NET Core Identity enables several scenarios:

- Create new user information using the UserManager type (userManager.CreateAsync).
- Authenticate users using the SignInManager type. You can use signInManager.SignInAsync to sign in directly, or signInManager.PasswordSignInAsync to confirm the user's password is correct and then sign them in.
- Identify a user based on information stored in a cookie (which is read by ASP.NET Core Identity middleware) so that subsequent requests from a browser will include a signed-in user's identity and claims.

ASP.NET Core Identity also supports two-factor authentication.

For authentication scenarios that make use of a local user data store and that persist identity between requests using cookies (as is typical for MVC web applications), ASP.NET Core Identity is a recommended solution.

Authenticate with external providers

ASP.NET Core also supports using external authentication providers to let users sign in via OAuth 2.0 flows. This means that users can sign in using existing authentication processes from providers like Microsoft, Google, Facebook, or Twitter and associate those identities with an ASP.NET Core identity in your application.

To use external authentication, besides including the authentication middleware as mentioned before, using the app.UseAuthentication() method, you also have to register the external provider in *Program.cs* as shown in the following example:

```
//...
services.AddDefaultIdentity<IdentityUser>(options => options.SignIn.RequireConfirmedAccount
= true)
    .AddEntityFrameworkStores<ApplicationDbContext>();

services.AddAuthentication()
    .AddMicrosoftAccount(microsoftOptions =>
    {
        microsoftOptions.ClientId =
builder.Configuration["Authentication:Microsoft:ClientId"];
        microsoftOptions.ClientSecret =
builder.Configuration["Authentication:Microsoft:ClientSecret"];
    })
    .AddGoogle(googleOptions => { ... })
    .AddTwitter(twitterOptions => { ... })
    .AddFacebook(facebookOptions => { ... });
//...
```

Popular external authentication providers and their associated NuGet packages are shown in the following table:

Provider	Package
Microsoft	**Microsoft.AspNetCore.Authentication.MicrosoftAccount**
Google	**Microsoft.AspNetCore.Authentication.Google**
Facebook	**Microsoft.AspNetCore.Authentication.Facebook**
Twitter	**Microsoft.AspNetCore.Authentication.Twitter**

In all cases, you must complete an application registration procedure that is vendor dependent and that usually involves:

1. Getting a Client Application ID.
2. Getting a Client Application Secret.
3. Configuring a redirection URL, that's handled by the authorization middleware and the registered provider
4. Optionally, configuring a sign-out URL to properly handle sign out in a Single Sign On (SSO) scenario.

For details on configuring your app for an external provider, see the [External provider authentication in the ASP.NET Core documentation]).

Tip

All details are handled by the authorization middleware and services previously mentioned. So, you just have to choose the **Individual User Account** authentication option when you create the ASP.NET Core web application project in Visual Studio, as shown in Figure 9-3, besides registering the authentication providers previously mentioned.

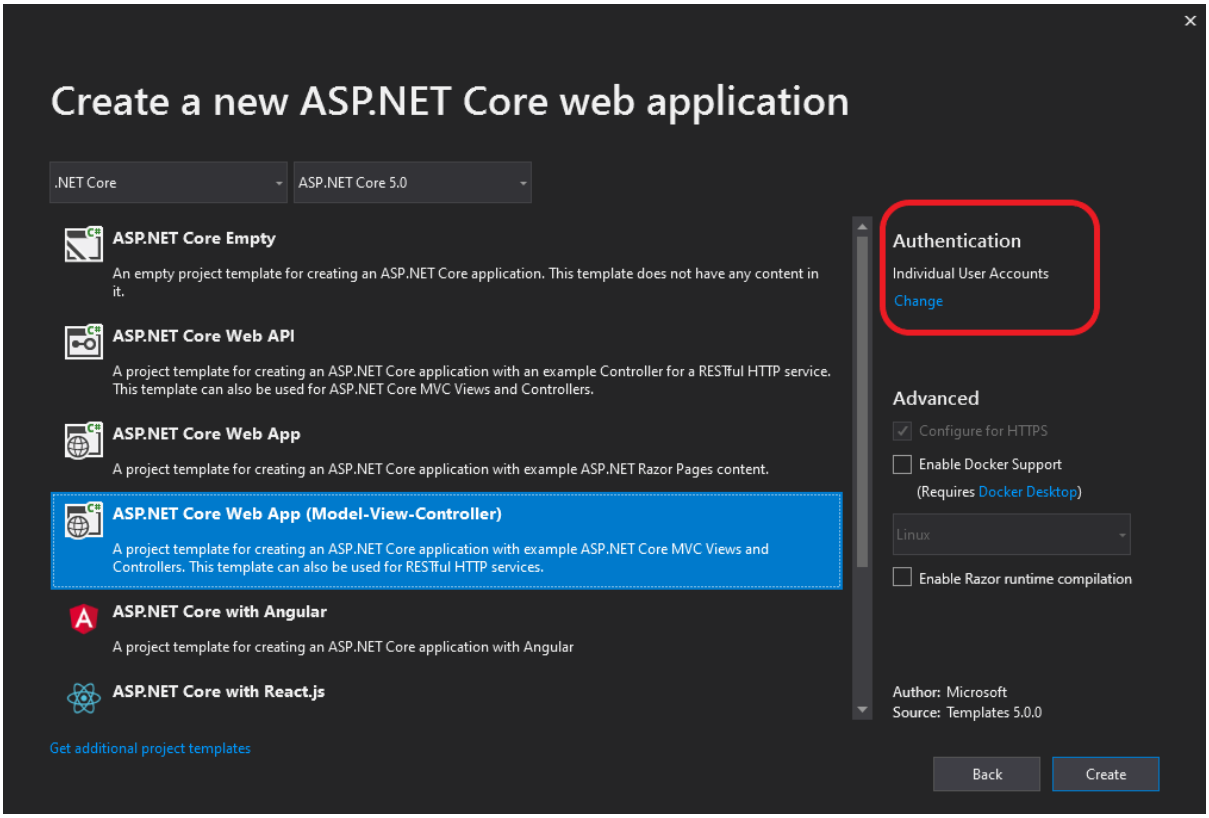


Figure 9-3. Selecting the Individual User Accounts option, for using external authentication, when creating a web application project in Visual Studio 2019.

In addition to the external authentication providers listed previously, third-party packages are available that provide middleware for using many more external authentication providers. For a list, see the AspNet.Security.OAuth.Providers repository on GitHub.

You can also create your own external authentication middleware to solve some special need.

Authenticate with bearer tokens

Authenticating with ASP.NET Core Identity (or Identity plus external authentication providers) works well for many web application scenarios in which storing user information in a cookie is appropriate. In other scenarios, though, cookies are not a natural means of persisting and transmitting data.

For example, in an ASP.NET Core Web API that exposes RESTful endpoints that might be accessed by Single Page Applications (SPAs), by native clients, or even by other Web APIs, you typically want to use bearer token authentication instead. These types of applications do not work with cookies, but can easily retrieve a bearer token and include it in the authorization header of subsequent requests. To enable token authentication, ASP.NET Core supports several options for using OAuth 2.0 and OpenID Connect.

Authenticate with an OpenID Connect or OAuth 2.0 Identity provider

If user information is stored in Azure Active Directory or another identity solution that supports OpenID Connect or OAuth 2.0, you can use the **Microsoft.AspNetCore.Authentication.OpenIdConnect** package to authenticate using the OpenID Connect workflow. For example, to authenticate to the Identity.Api microservice in eShopOnContainers, an ASP.NET Core web application can use middleware from that package as shown in the following simplified example in *Program.cs*:

```
// Program.cs

var identityUrl = builder.Configuration.GetValue<string>("IdentityUrl");
var callBackUrl = builder.Configuration.GetValue<string>("CallBackUrl");
var sessionCookieLifetime = builder.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");
});

// Build the app
//…
app.UseAuthentication();
//…
app.UseEndpoints(endpoints =>
{
    //...
});
```

When you use this workflow, the ASP.NET Core Identity middleware is not needed, because all user information storage and authentication is handled by the Identity service.

Issue security tokens from an ASP.NET Core service

If you prefer to issue security tokens for local ASP.NET Core Identity users rather than using an external identity provider, you can take advantage of some good third-party libraries.

IdentityServer4 and OpenIddict are OpenID Connect providers that integrate easily with ASP.NET Core Identity to let you issue security tokens from an ASP.NET Core service. The IdentityServer4 documentation has in-depth instructions for using the library. However, the basic steps to using IdentityServer4 to issue tokens are as follows.

1. You configure IdentityServer4 in *Program.cs* by making a call to builder.Services.AddIdentityServer.
2. You call app.UseIdentityServer in *Program.cs* to add IdentityServer4 to the application's HTTP request processing pipeline. This lets the library serve requests to OpenID Connect and OAuth2 endpoints like /connect/token.
3. You configure identity server by setting the following data:
   - The credentials to use for signing.
   - The Identity and API resources that users might request access to:
     - API resources represent protected data or functionality that a user can access with an access token. An example of an API resource would be a web API (or set of APIs) that requires authorization.
     - Identity resources represent information (claims) that are given to a client to identify a user. The claims might include the user name, email address, and so on.
   - The clients that will be connecting in order to request tokens.
   - The storage mechanism for user information, such as ASP.NET Core Identity or an alternative.

When you specify clients and resources for IdentityServer4 to use, you can pass an IEnumerable collection of the appropriate type to methods that take in-memory client or resource stores. Or for more complex scenarios, you can provide client or resource provider types via Dependency Injection.

A sample configuration for IdentityServer4 to use in-memory resources and clients provided by a custom IClientStore type might look like the following example:

```
// Program.cs

builder.Services.AddSingleton<IClientStore, CustomClientStore>();
builder.Services.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<ApplicationUser>();
//...
```

Consume security tokens

Authenticating against an OpenID Connect endpoint or issuing your own security tokens covers some scenarios. But what about a service that simply needs to limit access to those users who have valid security tokens that were provided by a different service?

For that scenario, authentication middleware that handles JWT tokens is available in the **Microsoft.AspNetCore.Authentication.JwtBearer** package. JWT stands for "JSON Web Token" and is a common security token format (defined by RFC 7519) for communicating security claims. A simplified example of how to use middleware to consume such tokens might look like this code fragment, taken from the Ordering.Api microservice of eShopOnContainers.

```
// Program.cs

var identityUrl = builder.Configuration.GetValue<string>("IdentityUrl");

// Add Authentication services

builder.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";
});

// Build the app

app.UseAuthentication();
//…
app.UseEndpoints(endpoints =>
{
    //...
});
```

The parameters in this usage are:

- Audience represents the receiver of the incoming token or the resource that the token grants access to. If the value specified in this parameter does not match the parameter in the token, the token will be rejected.
- Authority is the address of the token-issuing authentication server. The JWT bearer authentication middleware uses this URI to get the public key that can be used to validate the token's signature. The middleware also confirms that the iss parameter in the token matches this URI.

Another parameter, RequireHttpsMetadata, is useful for testing purposes; you set this parameter to false so you can test in environments where you don't have certificates. In real-world deployments, JWT bearer tokens should always be passed only over HTTPS.

With this middleware in place, JWT tokens are automatically extracted from authorization headers. They are then deserialized, validated (using the values in the Audience and Authority parameters), and stored as user information to be referenced later by MVC actions or authorization filters.

The JWT bearer authentication middleware can also support more advanced scenarios, such as using a local certificate to validate a token if the authority is not available. For this scenario, you can specify a TokenValidationParameters object in the JwtBearerOptions object.

Additional resources

- **Sharing cookies between applications**
  https://learn.microsoft.com/aspnet/core/security/cookie-sharing
- **Introduction to Identity**
  https://learn.microsoft.com/aspnet/core/security/authentication/identity
- **Rick Anderson. Two-factor authentication with SMS**
  https://learn.microsoft.com/aspnet/core/security/authentication/2fa
- **Enabling authentication using Facebook, Google and other external providers**
  https://learn.microsoft.com/aspnet/core/security/authentication/social/
- **Michell Anicas. An Introduction to OAuth 2**
  https://www.digitalocean.com/community/tutorials/an-introduction-to-oauth-2
- **AspNet.Security.OAuth.Providers** (GitHub repo for ASP.NET OAuth providers)
  https://github.com/aspnet-contrib/AspNet.Security.OAuth.Providers/tree/dev/src
- **IdentityServer4. Official documentation**
  https://identityserver4.readthedocs.io/en/latest/

About authorization in .NET microservices and web applications

After authentication, ASP.NET Core Web APIs need to authorize access. This process allows a service to make APIs available to some authenticated users, but not to all. Authorization can be done based on users' roles or based on custom policy, which might include inspecting claims or other heuristics.

Restricting access to an ASP.NET Core MVC route is as easy as applying an Authorize attribute to the action method (or to the controller's class if all the controller's actions require authorization), as shown in following example:

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
```

```
    public ActionResult Logout()
    {
    }
}
```

By default, adding an Authorize attribute without parameters will limit access to authenticated users for that controller or action. To further restrict an API to be available for only specific users, the attribute can be expanded to specify required roles or policies that users must satisfy.

Implement role-based authorization

ASP.NET Core Identity has a built-in concept of roles. In addition to users, ASP.NET Core Identity stores information about different roles used by the application and keeps track of which users are assigned to which roles. These assignments can be changed programmatically with the RoleManager type that updates roles in persisted storage, and the UserManager type that can grant or revoke roles from users.

If you're authenticating with JWT bearer tokens, the ASP.NET Core JWT bearer authentication middleware will populate a user's roles based on role claims found in the token. To limit access to an MVC action or controller to users in specific roles, you can include a Roles parameter in the Authorize annotation (attribute), as shown in the following code fragment:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

In this example, only users in the Administrator or PowerUser roles can access APIs in the ControlPanel controller (such as executing the SetTime action). The ShutDown API is further restricted to allow access only to users in the Administrator role.

To require a user be in multiple roles, you use multiple Authorize attributes, as shown in the following example:

```
[Authorize(Roles = "Administrator, PowerUser")]
[Authorize(Roles = "RemoteEmployee ")]
[Authorize(Policy = "CustomPolicy")]
public ActionResult API1 ()
{
}
```

In this example, to call API1, a user must:

- Be in the Administrator *or* PowerUser role, *and*
- Be in the RemoteEmployee role, *and*

- Satisfy a custom handler for CustomPolicy authorization.

Implement policy-based authorization

Custom authorization rules can also be written using authorization policies. This section provides an overview. For more information, see the ASP.NET Authorization Workshop.

Custom authorization policies are registered in the Startup.ConfigureServices method using the service.AddAuthorization method. This method takes a delegate that configures an AuthorizationOptions argument.

```
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)));
});
```

As shown in the example, policies can be associated with different types of requirements. After the policies are registered, they can be applied to an action or controller by passing the policy's name as the Policy argument of the Authorize attribute (for example, [Authorize(Policy="EmployeesOnly")]) Policies can have multiple requirements, not just one (as shown in these examples).

In the previous example, the first AddPolicy call is just an alternative way of authorizing by role. If [Authorize(Policy="AdministratorsOnly")] is applied to an API, only users in the Administrator role will be able to access it.

The second AddPolicy call demonstrates an easy way to require that a particular claim should be present for the user. The RequireClaim method also optionally takes expected values for the claim. If values are specified, the requirement is met only if the user has both a claim of the correct type and one of the specified values. If you're using the JWT bearer authentication middleware, all JWT properties will be available as user claims.

The most interesting policy shown here is in the third AddPolicy method, because it uses a custom authorization requirement. By using custom authorization requirements, you can have a great deal of control over how authorization is performed. For this to work, you must implement these types:

- A Requirements type that derives from IAuthorizationRequirement and that contains fields specifying the details of the requirement. In the example, this is an age field for the sample MinimumAgeRequirement type.
- A handler that implements AuthorizationHandler, where T is the type of IAuthorizationRequirement that the handler can satisfy. The handler must implement the HandleRequirementAsync method, which checks whether a specified context that contains information about the user satisfies the requirement.

If the user meets the requirement, a call to context.Succeed will indicate that the user is authorized. If there are multiple ways that a user might satisfy an authorization requirement, multiple handlers can be created.

In addition to registering custom policy requirements with AddPolicy calls, you also need to register custom requirement handlers via Dependency Injection (services.AddTransient<IAuthorizationHandler, MinimumAgeHandler>()).

An example of a custom authorization requirement and handler for checking a user's age (based on a DateOfBirth claim) is available in the ASP.NET Core authorization documentation.

Authorization and minimal apis

ASP.NET supports minimal APIs as an alternative to controller-based APIs. Authorization policies are the recommended way to configure authorization for minimal APIs, as this example demonstrates:

```
// Program.cs
builder.Services.AddAuthorizationBuilder()
  .AddPolicy("admin_greetings", policy =>
        policy
            .RequireRole("admin")
            .RequireScope("greetings_api"));

// build the app

app.MapGet("/hello", () => "Hello world!")
  .RequireAuthorization("admin_greetings");
```

Additional resources

- **ASP.NET Core Authentication**
  https://learn.microsoft.com/aspnet/core/security/authentication/identity
- **ASP.NET Core Authorization**
  https://learn.microsoft.com/aspnet/core/security/authorization/introduction
- **Role-based Authorization**
  https://learn.microsoft.com/aspnet/core/security/authorization/roles
- **Custom Policy-Based Authorization**
  https://learn.microsoft.com/aspnet/core/security/authorization/policies
- **Authentication and authorization in minimal APIs** https://learn.microsoft.com/aspnet/core/fundamentals/minimal-apis/security

Store application secrets safely during development

To connect with protected resources and other services, ASP.NET Core applications typically need to use connection strings, passwords, or other credentials that contain sensitive information. These sensitive pieces of information are called *secrets*. It's a best practice to not include secrets in source

code and making sure not to store secrets in source control. Instead, you should use the ASP.NET Core configuration model to read the secrets from more secure locations.

You must separate the secrets for accessing development and staging resources from the ones used for accessing production resources, because different individuals will need access to those different sets of secrets. To store secrets used during development, common approaches are to either store secrets in environment variables or by using the ASP.NET Core Secret Manager tool. For more secure storage in production environments, microservices can store secrets in an Azure Key Vault.

Store secrets in environment variables

One way to keep secrets out of source code is for developers to set string-based secrets as environment variables on their development machines. When you use environment variables to store secrets with hierarchical names, such as the ones nested in configuration sections, you must name the variables to include the complete hierarchy of its sections, delimited with colons (:).

For example, setting an environment variable Logging:LogLevel:Default to Debug value would be equivalent to a configuration value from the following JSON file:

```
{
    "Logging": {
        "LogLevel": {
            "Default": "Debug"
        }
    }
}
```

To access these values from environment variables, the application just needs to call AddEnvironmentVariables on its ConfigurationBuilder when constructing an IConfigurationRoot object.

Note

Environment variables are commonly stored as plain text, so if the machine or process with the environment variables is compromised, the environment variable values will be visible.

Store secrets with the ASP.NET Core Secret Manager

The ASP.NET Core Secret Manager tool provides another method of keeping secrets out of source code **during development**. To use the Secret Manager tool, install the package **Microsoft.Extensions.Configuration.SecretManager** in your project file. Once that dependency is present and has been restored, the dotnet user-secrets command can be used to set the value of secrets from the command line. These secrets will be stored in a JSON file in the user's profile directory (details vary by OS), away from source code.

Secrets set by the Secret Manager tool are organized by the UserSecretsId property of the project that's using the secrets. Therefore, you must be sure to set the UserSecretsId property in your project file, as shown in the snippet below. The default value is a GUID assigned by Visual Studio, but the actual string is not important as long as it's unique in your computer.

```
<PropertyGroup>
    <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

Using secrets stored with Secret Manager in an application is accomplished by calling AddUserSecrets<T> on the ConfigurationBuilder instance to include secrets for the application in its configuration. The generic parameter T should be a type from the assembly that the UserSecretId was applied to. Usually, using AddUserSecrets<Startup> is fine.

The AddUserSecrets<Startup>() is included in the default options for the Development environment when using the CreateDefaultBuilder method in *Program.cs*.

Use Azure Key Vault to protect secrets at production time

Secrets stored as environment variables or stored by the Secret Manager tool are still stored locally and unencrypted on the machine. A more secure option for storing secrets is Azure Key Vault, which provides a secure, central location for storing keys and secrets.

The **Azure.Extensions.AspNetCore.Configuration.Secrets** package allows an ASP.NET Core application to read configuration information from Azure Key Vault. To start using secrets from an Azure Key Vault, you follow these steps:

1. Register your application as an Azure AD application. (Access to key vaults is managed by Azure AD.) This can be done through the Azure management portal.

   Alternatively, if you want your application to authenticate using a certificate instead of a password or client secret, you can use the New-AzADApplication PowerShell cmdlet. The certificate that you register with Azure Key Vault needs only your public key. Your application will use the private key.

2. Give the registered application access to the key vault by creating a new service principal. You can do this using the following PowerShell commands:

```
$sp = New-AzADServicePrincipal -ApplicationId "<Application ID guid>"
Set-AzKeyVaultAccessPolicy -VaultName "<VaultName>" -ServicePrincipalName
$sp.ServicePrincipalNames[0] -PermissionsToSecrets all -ResourceGroupName "<KeyVault
Resource Group>"
```

3. Include the key vault as a configuration source in your application by calling the AzureKeyVaultConfigurationExtensions.AddAzureKeyVault extension method when you create an IConfigurationRoot instance.

Note that calling AddAzureKeyVault requires the application ID that was registered and given access to the key vault in the previous steps. Or you can firstly running the Azure CLI command: az login, then using an overload of AddAzureKeyVault that takes a DefaultAzureCredential in place of the client.

Important

We recommend that you register Azure Key Vault as the last configuration provider, so it can override configuration values from previous providers.

Additional resources

- **Using Azure Key Vault to protect application secrets**
  https://learn.microsoft.com/azure/architecture/multitenant-identity
- **Safe storage of app secrets during development**
  https://learn.microsoft.com/aspnet/core/security/app-secrets
- **Configuring data protection**
  https://learn.microsoft.com/aspnet/core/security/data-protection/configuration/overview
- **Data Protection key management and lifetime in ASP.NET Core**
  https://learn.microsoft.com/aspnet/core/security/data-protection/configuration/default-settings

.NET Microservices Architecture key takeaways

As a summary and key takeaways, the following are the most important conclusions from this guide.

Benefits of using containers. Container-based solutions provide important cost savings because they help reduce deployment problems caused by failing dependencies in production environments. Containers significantly improve DevOps and production operations.

Containers will be ubiquitous. Docker-based containers are becoming the de facto standard in the industry, supported by key vendors in the Windows and Linux ecosystems, such as Microsoft, Amazon AWS, Google, and IBM. Docker will probably soon be ubiquitous in both the cloud and on-premises datacenters.

Containers as a unit of deployment. A Docker container is becoming the standard unit of deployment for any server-based application or service.

Microservices. The microservices architecture is becoming the preferred approach for distributed and large or complex mission-critical applications based on many independent subsystems in the form of autonomous services. In a microservice-based architecture, the application is built as a collection of services that are developed, tested, versioned, deployed, and scaled independently. Each service can include any related autonomous database.

Domain-driven design and SOA. The microservices architecture patterns derive from service-oriented architecture (SOA) and domain-driven design (DDD). When you design and develop microservices for environments with evolving business needs and rules, it's important to consider DDD approaches and patterns.

Microservices challenges. Microservices offer many powerful capabilities, like independent deployment, strong subsystem boundaries, and technology diversity. However, they also raise many new challenges related to distributed application development, such as fragmented and independent data models, resilient communication between microservices, eventual consistency, and operational complexity that results from aggregating logging and monitoring information from multiple microservices. These aspects introduce a much higher complexity level than a traditional monolithic application. As a result, only specific scenarios are suitable for microservice-based applications. These include large and complex applications with multiple evolving subsystems. In these cases, it's worth investing in a more complex software architecture, because it will provide better long-term agility and application maintenance.

Containers for any application. Containers are convenient for microservices, but can also be useful for monolithic applications based on the traditional .NET Framework, when using Windows Containers. The benefits of using Docker, such as solving many deployment-to-production issues and providing state-of-the-art Dev and Test environments, apply to many different types of applications.

CLI versus IDE. With Microsoft tools, you can develop containerized .NET applications using your preferred approach. You can develop with a CLI and an editor-based environment by using the Docker CLI and Visual Studio Code. Or you can use an IDE-focused approach with Visual Studio and its unique features for Docker, such as multi-container debugging.

Resilient cloud applications. In cloud-based systems and distributed systems in general, there is always the risk of partial failure. Since clients and services are separate processes (containers), a service might not be able to respond in a timely way to a client's request. For example, a service might be down because of a partial failure or for maintenance; the service might be overloaded and responding slowly to requests; or it might not be accessible for a short time because of network issues. Therefore, a cloud-based application must embrace those failures and have a strategy in place to respond to those failures. These strategies can include retry policies (resending messages or retrying requests) and implementing circuit-breaker patterns to avoid exponential load of repeated requests. Basically, cloud-based applications must have resilient mechanisms—either based on cloud infrastructure or custom, as the high-level ones provided by orchestrators or service buses.

Security. Our modern world of containers and microservices can expose new vulnerabilities. There are several ways to implement basic application security, based on authentication and authorization. However, container security must consider additional key components that result in inherently safer applications. A critical element of building safer apps is having a secure way of communicating with other apps and systems, something that often requires credentials, tokens, passwords, and the like, commonly referred to as application secrets. Any secure solution must follow security best practices, such as encrypting secrets while in transit and at rest, and preventing secrets from leaking when consumed by the final application. Those secrets need to be stored and kept safely, as when using Azure Key Vault.

Orchestrators. Container-based orchestrators, such as Azure Kubernetes Service and Azure Service Fabric are key part of any significant microservice and container-based application. These applications carry with them high complexity, scalability needs, and go through constant evolution. This guide has introduced orchestrators and their role in microservice-based and container-based solutions. If your application needs are moving you toward complex containerized apps, you will find it useful to seek out additional resources for learning more about orchestrators.