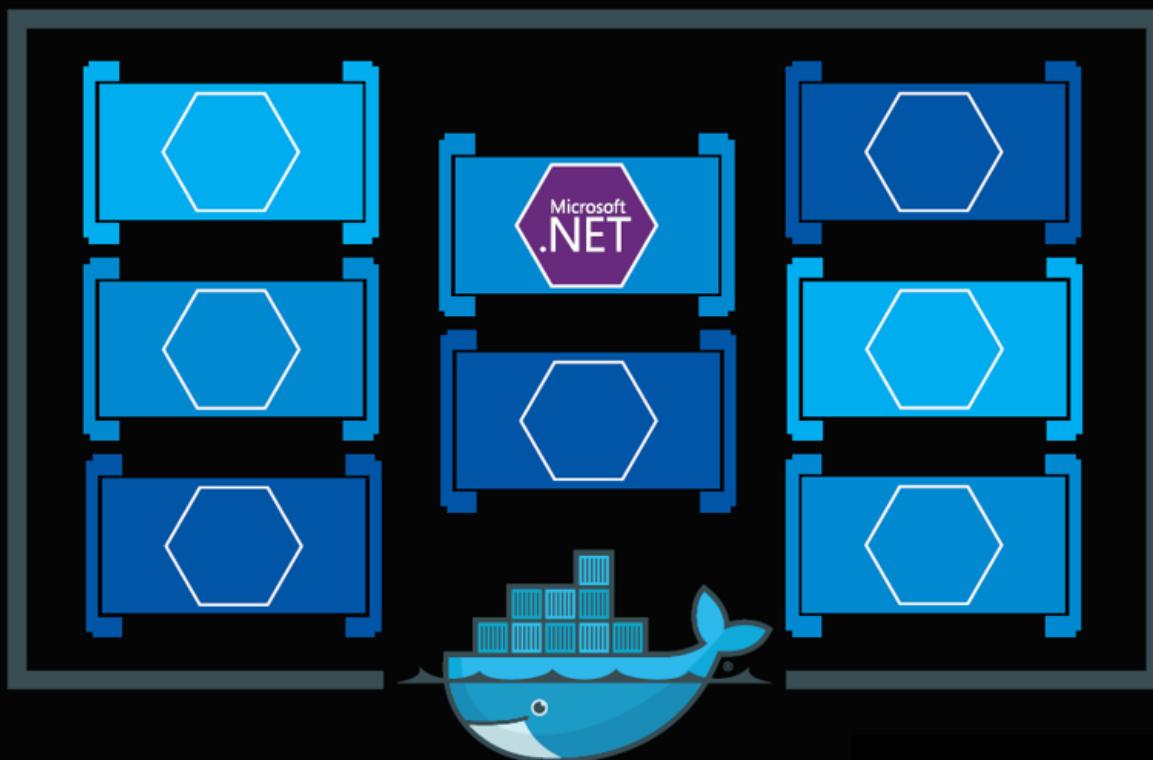


DRAFT



Microsoft

.NET Microservices: Architecture for Containerized .NET Applications



Cesar de la Torre

Bill Wagner

Mike Rousos

Microsoft Corporation



DRAFT VERSION 0.5

----- This eBook (250+ pages) download available at: <https://aka.ms/microservicesebook> -----

----- This is a draft version. We're accepting feedback which will be taken into account -----

----- Please, send direct feedback to dotnet-architecture-ebooks-feedback@service.microsoft.com -----

PUBLISHED BY

Microsoft Developer Division, .NET and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2017 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 <http://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 are property of their respective owners.

Co-Authors:

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

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

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

Editors:

Steve Hoag, Mike Pope

Participants and reviewers:

Full name, Title, Team, Company

----- *TBD SECTION IN DRAFT* -----

----- *Reviewes and participants list will be added once the process is finished* -----

Contents

| | |
|---|-----------|
| Introduction..... | 1 |
| About this guide | 1 |
| What this guide does not cover..... | 1 |
| Who should use this guide..... | 2 |
| How to use this guide..... | 2 |
| Related microservice- and container-based reference application: eShopOnContainers..... | 2 |
| Introduction to Containers and Docker | 3 |
| What is Docker?..... | 4 |
| Comparing Docker containers with virtual machines..... | 5 |
| Docker terminology | 6 |
| Docker containers, images, and registries | 7 |
| Choosing Between .NET Core and .NET Framework for Docker Containers | 9 |
| General guidance | 9 |
| When to choose .NET Core for Docker containers..... | 10 |
| Developing and deploying cross platform | 10 |
| Using containers for new ("green-field") projects..... | 10 |
| Creating and deploying 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 Docker container..... | 12 |
| Using third-party .NET libraries or NuGet packages not available for .NET Core | 12 |
| Using.NET technologies not available for .NET Core | 12 |
| Using a platform or API that doesn't support .NET Core..... | 13 |
| Decision table: .NET frameworks to use for Docker..... | 14 |
| What OS to target with .NET containers..... | 15 |
| Official .NET Docker images | 15 |
| .NET Core and Docker image optimizations for development versus production | 16 |
| Architecting Container- and Microservice-Based Applications | 18 |
| Common container design principles..... | 18 |
| Containerized monolithic applications | 19 |
| Deploying a monolithic application as a container..... | 21 |

| | |
|--|-----------|
| Publishing a single-container based app to Azure App Service | 21 |
| State and data in Docker applications | 22 |
| Service-oriented architecture..... | 24 |
| Microservices architecture | 25 |
| Data sovereignty per microservice..... | 26 |
| The relationship between microservices and the Bounded Context pattern..... | 28 |
| Logical architecture versus physical architecture | 29 |
| Challenges and solutions for distributed data management..... | 30 |
| Identifying domain-model boundaries for each microservice..... | 35 |
| Direct client-to-microservice communication versus the API Gateway pattern | 38 |
| Communication between microservices | 42 |
| Creating composite UI based on microservices, including visual UI shape and layout generated by multiple microservices | 53 |
| Resiliency and high availability in microservices | 55 |
| Health management and diagnostics in microservices | 55 |
| Orchestrating microservices and multi-container applications for high scalability and availability ... | 58 |
| Using container-based orchestrators in Microsoft Azure..... | 60 |
| Using Azure Container Service | 60 |
| Using Azure Service Fabric | 62 |
| Stateless versus stateful microservices..... | 65 |
| Development Process for Docker-Based Applications | 67 |
| Vision | 67 |
| Development environment for Docker apps | 67 |
| Development tools choices: IDE or editor..... | 67 |
| .NET languages and frameworks for Docker containers | 68 |
| Development workflow for Docker apps | 68 |
| Workflow for developing Docker container-based applications | 68 |
| Simplified workflow when developing containers with Visual Studio | 80 |
| Using PowerShell commands in a Dockerfile to set up Windows containers | 81 |
| Deploying Single-Container-Based .NET Core Web Applications on Linux or Windows Nano Server Hosts..... | 82 |
| Vision | 82 |
| Application tour..... | 83 |
| Docker support..... | 84 |
| Troubleshooting..... | 86 |
| Stopping Docker containers | 86 |
| Adding Docker to your projects..... | 86 |
| Migrating Legacy Monolithic .NET Framework Applications to Windows Containers | 87 |
| Vision | 87 |

| | |
|--|------------|
| Benefits | 88 |
| Possible migration paths | 89 |
| Application tour..... | 89 |
| Lifting and shifting | 91 |
| Getting data from the existing catalog .NET Core microservice..... | 93 |
| Development and production environments | 93 |
| Designing and Developing Multi-Container and Microservice-Based .NET Applications | 94 |
| Vision | 94 |
| Designing a microservice-oriented application..... | 94 |
| Application specifications | 94 |
| Development team context | 95 |
| Choosing an architecture..... | 95 |
| Benefits of a microservice-based solution | 98 |
| 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 | 103 |
| Designing a simple CRUD microservice | 103 |
| Implementing a simple CRUD microservice with ASP.NET Core..... | 104 |
| Generating Swagger description metadata from your ASP.NET Core Web API | 111 |
| Defining your multi-container application with docker-compose.yml | 115 |
| Using a database server running as a container | 129 |
| Implementing event-based communication between microservices (integration events) | 134 |
| Using message brokers and services buses for production systems | 135 |
| Integration events..... | 135 |
| The event bus | 136 |
| Testing ASP.NET Core services and web apps | 150 |
| Tackling Business Complexity in a Microservice with DDD and CQRS Patterns..... | 154 |
| Applying simplified CQRS and DDD patterns within a microservice | 155 |
| CQRS and CQS approaches in a DDD microservice..... | 157 |
| CQRS and DDD patterns are not top-level architectures..... | 157 |
| Implementing the Reads/Queries in a CQRS microservice | 158 |
| ViewModels specifically made for client apps, independent from the domain model constraints | 159 |
| Dapper: selected Micro ORM as mechanism to query in the eShopOnContainers sample ordering microservice | 160 |
| Dynamic and static ViewModels..... | 160 |
| Designing a domain-driven design-oriented microservice | 162 |
| Keep the microservice context boundaries relatively small | 162 |

| | |
|---|-----|
| Layers in domain-driven design microservices | 162 |
| Designing a microservice domain model..... | 165 |
| Implementing a microservice's domain model with .NET Core | 170 |
| Domain model structure in a .NET Core Standard Library | 170 |
| Structuring aggregates in a .NET Standard Library | 171 |
| Implementing domain Entities as POCO classes | 171 |
| The SeedWork or reusable base classes and interfaces for your domain model | 175 |
| Repository contracts and interfaces placed in the domain model layer | 177 |
| Value objects..... | 178 |
| Value object implementation in C# | 179 |
| Using Enumeration classes instead of Enums..... | 181 |
| Designing validations in the domain model layer | 183 |
| Implementing validations in the domain model layer | 184 |
| Client side validation (validation in the presentation layers)..... | 186 |
| Domain events..... | 187 |
| What is a domain event?..... | 187 |
| Domain events versus integration events | 188 |
| Implementing domain events..... | 191 |
| How to implement a domain event..... | 191 |
| Raising domain events..... | 192 |
| The deferred approach for raising and dispatching events | 192 |
| Single transaction across aggregates versus eventual consistency across aggregates..... | 193 |
| The domain event dispatcher: mapping from events to event handlers..... | 195 |
| How to subscribe to domain events..... | 196 |
| How to handle domain events..... | 196 |
| Designing the infrastructure persistence layer | 198 |
| The Repository pattern | 198 |
| Implementing the infrastructure persistence layer with Entity Framework Core | 201 |
| Introduction to Entity Framework Core..... | 201 |
| Infrastructure in Entity Framework Core from a DDD perspective..... | 202 |
| Implementing custom repositories with Entity Framework Core | 203 |
| EF DbContext and IUnitOfWork instance lifetime in your IoC container..... | 205 |
| The repository instance lifetime in your IoC container..... | 206 |
| Table mapping | 207 |
| Using NoSQL databases as a persistence infrastructure..... | 209 |
| Designing the microservice's application layer and Web API..... | 212 |
| Using S.O.L.I.D. principles and Dependency Injection..... | 212 |
| Implementing the microservice's application layer and Web API..... | 213 |

| | |
|---|------------|
| Using Dependency Injection to inject infrastructure objects into your application layer | 213 |
| Implementing the Command and Command-Handlers patterns | 216 |
| The Command's process pipeline: how to trigger a command handler | 222 |
| Implementing the Command process pipeline with a mediator pattern (MediatR) | 224 |
| Applying cross-cutting concerns when processing commands with the Mediator and Decorator patterns | 226 |
| Why sagas? | 227 |
| Implementing Resilient Applications | 228 |
| Vision | 228 |
| Handling partial failure | 228 |
| Strategies for handling partial failure | 230 |
| Implementing retries with exponential backoff | 232 |
| Implementing the Circuit Breaker pattern | 239 |
| Using the ResilientHttpClient utility class from eShopOnContainers | 240 |
| Health monitoring | 245 |
| Implementing health checks in ASP.NET Core services | 245 |
| Watchdogs | 248 |
| Health checks when using orchestrators | 249 |
| Advanced monitoring: visualization, analysis, and alerts | 250 |
| Securing .NET Microservices and Web Applications | 251 |
| Implementing authentication in .NET microservices and web applications | 251 |
| Authenticating using ASP.NET Core Identity | 252 |
| Authenticating using external providers | 253 |
| Authenticating with bearer tokens | 255 |
| About authorization in .NET microservices and web applications | 258 |
| Implementing role-based authorization | 259 |
| Implementing policy-based authorization | 259 |
| Storing app secrets safely during development | 261 |
| Storing secrets as environment variables | 261 |
| Storing secrets using the ASP.NET Core Secret Manager | 262 |
| Using Azure Key Vault to protect secrets at production time | 262 |
| Key Takeaways | 265 |

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 Container 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 Core 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 Core. 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've studied this guide, your next step would be to learn about production-ready microservices on Microsoft Azure.

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. This 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 ebook)
<https://aka.ms/dockerlifecyclebook>

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 Core) 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 Core 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 who do not 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 development and microservice patterns for implementing applications using .NET Core 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 a reference app for .NET Core 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 app and a native mobile app). It also includes the back-end microservices and containers for all required server-side operations.

This microservice and container-based application source code is open source and available at the [eShopOnContainers GitHub repo](#).

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 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.

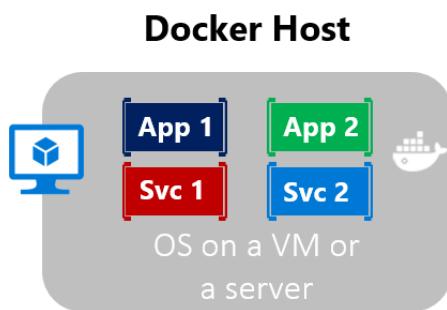


Figure 2-1. Multiple containers running on a container host

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 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 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.

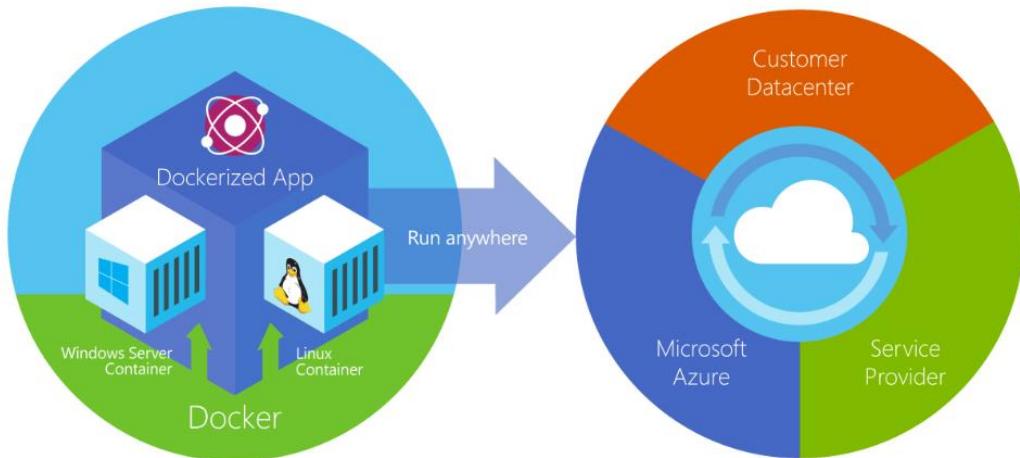


Figure 2-2. Docker deploys containers at all layers of the hybrid cloud

Docker image containers can run natively on Linux and Windows. However, Windows images can run only on Windows hosts and Linux images can run only on Linux hosts, being a host 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 the Mac use a Docker host that is Linux based, and they can create images only for Linux containers. (Developers working on the Mac can edit code or run the Docker CLI from macOS, but as of the time of this writing, containers do not 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 Community Edition \(CE\)](#) for Windows or for macOS. These products install the necessary VM (the Docker host) to host the containers. Docker also makes available [Docker Enterprise Edition \(EE\)](#), which is designed for enterprise development and is used by IT teams who build, ship, and run large business-critical applications in production.

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 is not 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.

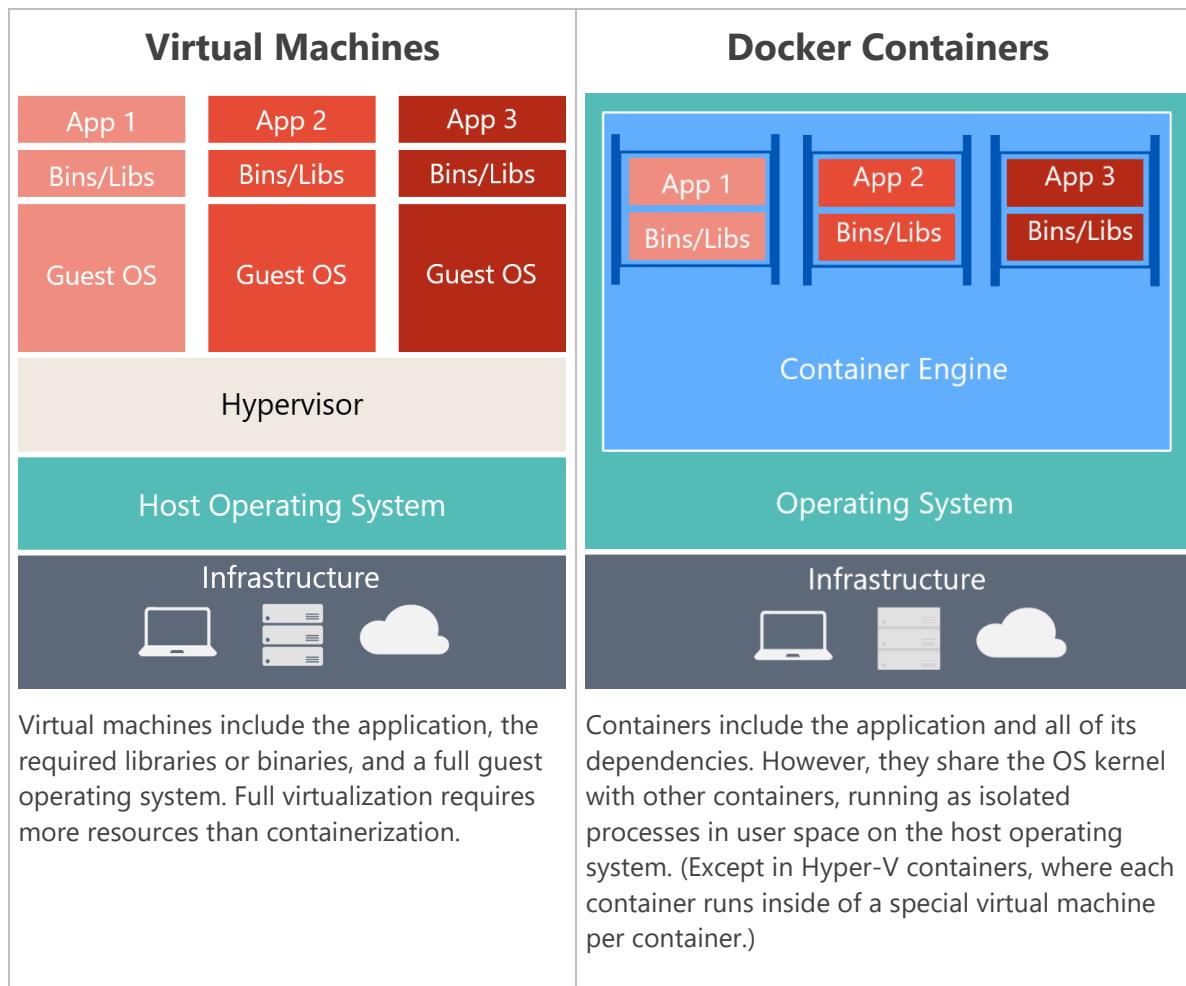


Figure 2-3. Comparison of traditional virtual machines to Docker containers

Because containers require far fewer resources (for example, they do not need a full OS), they are 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 is not only a technology, but also a philosophy and a process.

When using Docker, you will not 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 will run the way it was intended to on all deployment targets (Dev, QA, staging, production, etc.).

Docker terminology

This section lists terms and definitions you should be familiar with before getting deeper into Docker. For further definitions, an extensive [glossary](#) is provided by Docker (<https://docs.docker.com/v1.11/engine/reference/glossary/>).

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.

Container: An instance of a Docker image. A container represents the execution of a single application, process, or service. A container 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.

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

Dockerfile: A text file that contains instructions for how to build a Docker image.

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 Docker CLI command `docker build`.

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 and a Windows image.

Registry: A service providing 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.

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 is 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 is convenient for private images to be managed within the enterprise. Docker Trusted Registry is included as part of the Docker Datacenter product. For more information, see [Docker Trusted Registry \(DTR\)](#).

Docker Community Edition (CE): Development tools for Windows and macOS for building, running, and testing containers locally. Docker CE for Windows provides both Linux and Windows containers development environments. 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 CE for Mac is based on the Apple Hypervisor framework and the [xhyve hypervisor](#), which provides a Linux Docker host virtual machine on Mac OS X. Docker CE for Windows and Mac replaces Docker Toolbox, which was based on Oracle VirtualBox.

Docker Enterprise Edition (EE): An enterprise-scale version of Docker tools for Linux and Windows development.

Compose: A command-line tool and .yml file format with metadata for defining and running multi-container applications. You define a single application based on multiple images with one or multiple .yml files that can override values depending on the environment. Once you have 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 the can scale to multiple instances of the services spread across multiple hosts within the cluster. Docker clusters can be created with Docker Swarm, Mesosphere DC/OS, Kubernetes, and Azure Service Fabric. (If you use Docker Swarm for managing a cluster, you typically refer to it the cluster as a swarm instead of a cluster.)

Orchestrator: A tool that simplifies 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 Mesosphere DC/OS, Kubernetes, Docker Swarm, and Azure Service Fabric.

Docker containers, images, and registries

The 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 image is instantiated to create a container, which is the app or service running as a container on the Docker host, 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](#), and other vendors provide registries for different collections of images. 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.

Basic taxonomy in Docker

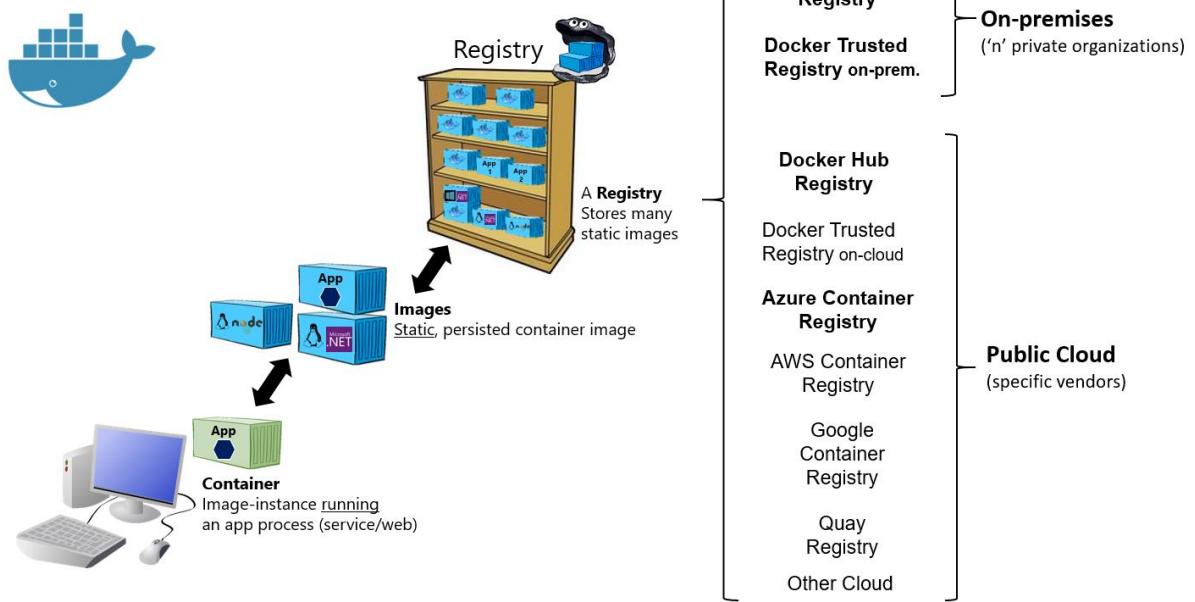


Figure 2-4. Taxonomy of Docker terms and concepts

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 reasons.
- You want to have minimum network latency between your images and your chosen deployment environment. For example, if your production environment is in Azure’s cloud you will probably want to store your images in *Azure Container Registry* so the 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 Core and .NET Framework for Docker Containers

There are two supported choices of frameworks for building server-side containerized Docker applications with .NET: [.NET Framework](#) and [.NET Core](#). Both share a lot of the same .NET platform components and you can share code across the two. However, there are fundamental differences between the two and your choice will depend on what you want to accomplish. This section provides guidance on when to use each.

General guidance

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

You should use .NET Core 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 Core as the default choice. It has many benefits and fits much better with the containers philosophy and style of working.

An additional benefit of using .NET Core 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 don't use containers, since containers isolate the versions of .NET that the app needs (as long as they are compatible with the underlying OS).

You should 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 Core.

- You need to use third-party .NET libraries or NuGet packages that are not available for .NET Core.

Using .NET Framework on Docker can improve your deployment experiences by minimizing deployment issues. This “lift and shift” scenario is important for “dockerizing” legacy applications (at least, those that are not based on microservices).

When to choose .NET Core for Docker containers

The modularity and lightweight nature of .NET Core makes it perfect for containers. When you deploy and start a container, its image is far smaller with .NET Core 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.

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

The following is a more detailed explanation of why to pick .NET Core.

Developing and deploying cross platform

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

.NET Core also supports macOS as a development platform, but when deploying containers to a Docker host, that host (currently) must 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. [Visual Studio for Mac](#) is an evolution of Xamarin Studio running in macOS, but as of the time of this writing, it still does not support Docker development. You can also use [Visual Studio Code](#) (VS Code) on macOS, Linux, and Windows. VS Code fully supports .NET Core, including IntelliSense and debugging. Since it is a lightweight editor, you can use it to develop containerized apps in the Mac in conjunction with the Docker CLI and the .NET Core CLI (dotnet cli). You can also target .NET Core 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 Core command-line tools (dotnet 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 Core makes it perfect for containers and microservices architectures. When you create and deploy a container, its image is far smaller with .NET Core than with .NET Framework.

Creating and deploying microservices on containers

You could use the full .NET framework for microservices-based applications (without containers) when using plain processes, because .NET Framework is already installed and shared across processes. However, if you are using containers, the image for .NET Framework (Windows Server Core plus the full .NET Framework within each image) is probably too heavy for a microservices-on-containers approach.

In contrast, .NET Core is the best candidate if you are embracing a microservices-oriented system that's based on containers, because .NET Core is lightweight. In addition, its related container images, either the Linux image or the Windows Nano image, are lean and small.

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, 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 Core container image.

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

There are many orchestrators you can use when targeting microservices and containers. For large and complex microservice systems being deployed as Linux containers, [Azure Container Service](#) has multiple orchestrator offerings (Mesos DC/OS, Kubernetes, and Docker Swarm), which makes it a good choice. You can also use Azure Service Fabric for Linux, which supports Docker Linux containers. (At the time of this writing, this offering was still in [Preview](#). Check the [Azure Service Fabric](#) for the latest status.)

For large and complex microservice systems being deployed as Windows containers, most orchestrators are currently in a less mature state. However, you currently can use Azure Service Fabric for Windows containers, as well as Azure Container Service. Azure Service Fabric is well established for running mission-critical Windows applications.

All these platforms support .NET Core and make them ideal for hosting your microservices.

Deploying high density in scalable systems

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

This is especially relevant for microservices architectures, where you could have hundreds of microservices (containers) running. With ASP.NET Core images (based on the .NET Core 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 Core 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 Docker container

You might want to use Docker containers just to simplify deployment, even if you're not creating microservices. For example, perhaps you want to improve your DevOps workflow with Docker and so you can have better isolated test environments and you can also eliminate deployment issues caused by missing dependencies when moving to production environments. 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, you won't need to migrate your existing applications to .NET Core; you can use Docker containers that include the full .NET Framework. However, a recommended approach is to use .NET Core 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 Core

Third-party libraries are quickly embracing the [.NET Standard Library](#), which enables code sharing across all .NET flavors, including .NET Core. With the .NET Standard Library 2.0, this will be even easier, because the .NET Core API surface will become significantly bigger. Your .NET Core applications will be able to directly use existing .NET Framework libraries.

Be aware that whenever you run a library or process based on the traditional .NET Framework, because of its dependencies on Windows, the container image used for that application or service will need to be based on a Windows Container image.

Using .NET technologies not available for .NET Core

Some .NET Framework technologies are not available in the current version of .NET Core (version 1.1 as of this writing). Some of them will be available in later .NET Core releases (.NET Core 2), but others don't apply to the new application patterns targeted by .NET Core and may never be available.

The following list shows most of the technologies that are not available in .NET Core 1.1:

- 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 Core.
- ASP.NET Web Pages. This technology is slated to be included in a future .NET Core release, as explained in the [.NET Core roadmap](#).
- ASP.NET SignalR. As of the .NET Core 1.1 release (November 2016), ASP.NET SignalR is not available for ASP.NET Core (neither client nor server). There are plans to include it in a future release, as explained in the .NET Core roadmap. A preview is available at the [Server-side](#) and [Client Library](#) GitHub repositories.

- WCF services. Even when there's a [WCF-Client library](#) to consume WCF services from .NET Core (as of early 2017), the WCF server implementation is only available on .NET Framework. This scenario is being considered for future releases of .NET Core.
- 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 Core.
- Language support. As of the release of Visual Studio 2017, Visual Basic and F# don't have tooling support for .NET Core, but this support is planned for updated versions of Visual Studio.

In addition to the technologies listed in the official [.NET Core roadmap](#), other features might be ported to .NET Core. For a full list, look at the items tagged as [port-to-core](#) on the CoreFX GitHub site. Note that this list doesn't represent a commitment from Microsoft to bring those components to .NET Core—the items simply capture requests from the community. If you care about any of the components listed above, consider participating in the discussions on GitHub so that your voice can be heard. And if you think something is missing, please [file a new issue in the CoreFX repository](#).

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

Some Microsoft or third-party platforms don't support .NET Core. For example, some Azure services provide an SDK that is not yet available for consumption on .NET Core. This is temporary, because all Azure services will eventually use .NET Core. For example, the [Azure DocumentDB SDK for .NET Core](#) was released as a preview on November 16, 2016, but it is now generally available (GA) as a stable version.

In the meantime, you can always use the equivalent REST API from the Azure service instead of the client SDK.

Additional resources

- **.NET Core Guide**
<https://docs.microsoft.com/en-us/dotnet/articles/core/index>
- **Porting from .NET Framework to .NET Core**
<https://docs.microsoft.com/en-us/dotnet/articles/core/porting/index>
- **.NET Framework on Docker Guide**
<https://docs.microsoft.com/en-us/dotnet/articles/framework/docker/>
- **.NET Components Overview**
<https://docs.microsoft.com/en-us/dotnet/articles/standard/components>

Decision table: .NET frameworks to use for Docker

The following decision table summarizes whether to use .NET Framework or .NET Core.

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).

| Architecture / App Type | Linux containers | Windows containers |
|---|---|---|
| Microservices on containers | .NET Core | .NET Core |
| Monolithic app | .NET Core | .NET Framework .NET Core |
| Best-in-class performance and scalability | .NET Core | .NET Core |
| Windows Server legacy app ("brown-field") migration to containers | -- | .NET Framework |
| New container-based development ("green-field") | .NET Core | .NET Core |
| ASP.NET Core | .NET Core | .NET Core (recommended) .NET Framework |
| ASP.NET 4 (MVC 5, Web API 2, and Web Forms) | -- | .NET Framework |
| SignalR services | .NET Core (future release) | .NET Framework .NET Core (future release) |
| WCF, WF, and other legacy frameworks | WCF in .NET Core (in the roadmap) | .NET Framework WCF in .NET Core (in the roadmap) |
| Consumption of Azure services | .NET Core (eventually all Azure services will provide client SDKs for .NET Core) | .NET Framework .NET Core (eventually all Azure services will provide client SDKs for .NET Core) |

What OS to target with .NET containers

Given the diversity of operating systems supported by Docker and the differences between .NET Framework and .NET Core, you should target specific OS and versions depending on the framework you are using. For instance, in Linux there are many distros available, but only few of them are supported in the official .NET Docker images (like Debian and Alpine). For Windows you can use Windows Server Core or Nano Server; these OSs provide different characteristics (like IIS versus Kestrel) that might be needed by .NET Framework or .NET Core.

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

What OS to target with .NET containers

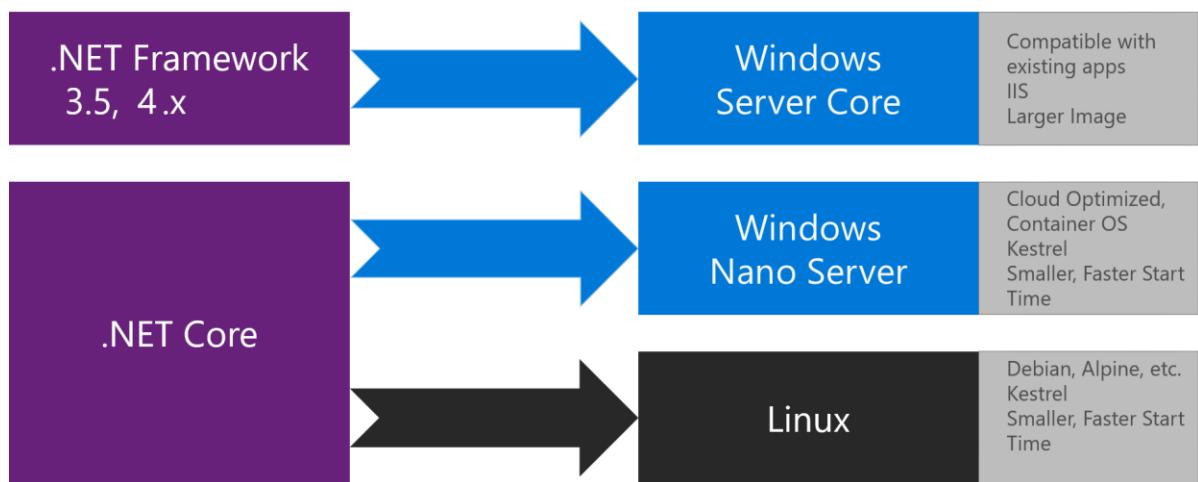


Figure 3-1. Operating systems to target depending on versions of the .NET frameworks

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 full .NET Framework and Windows Server Core, which is a not-so-common scenario for Docker.

When adding 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.

| | |
|---|---|
| microsoft/dotnet:1.1-runtime | .NET Core 1.1 runtime-only on Linux |
| microsoft/dotnet:1.1-runtime-nanoserver | .NET Core 1.1 runtime-only on Windows Nano Server |

Official .NET Docker images

The Official .NET Docker images are Docker images created and optimized by Microsoft. They are publicly available in the Microsoft repositories on [Docker Hub](#). 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, etc.).

Microsoft's vision for .NET repositories is to have granular and focused repos, where a repo represents a specific scenario or workload. For instance, the [microsoft/aspnetcore](#) images should be used when using ASP.NET Core on Docker, because those ASP.NET Core images provide additional optimizations so containers can spin-up faster.

On the other hand, the .NET Core images ([microsoft/dotnet](#)) are intended for console apps based on .NET Core. For example, batch processes, Azure WebJobs, and other console scenarios should use .NET Core. Those images don't include the ASP.NET Core stack, resulting in a smaller container image.

Most image repos provide extensive tagging to help you select not just a specific framework version, but also to choose an OS (Linux distro or Windows version).

For further information about the official .NET Docker images provided by Microsoft, see the [.NET Docker Images summary](#).

.NET Core 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 Core apps.
- Images used to *run* .NET Core 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's 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 of our tools, like [yo docker](#) for Visual Studio Code, use the development ASP.NET Core image ([microsoft/aspnetcore-build](#)) during development; you could even use that image as a build container. When building inside a Docker container, what's important are the elements that are needed in order to compile your app. This includes the compiler and any other .NET dependencies, plus web development dependencies like npm, Gulp, and Bower.

Why is this type of build image important? You don't deploy this image to production. Instead, it's an image 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. For instance, rather than manually installing all your application dependencies directly on a build agent host (a VM, for instance), the build agent would instantiate a .NET Core build image with all the dependencies required to build the application. Your build agent only needs to know how to run this Docker image. This simplifies your CI environment and makes it much more predictable.

During production

What's important in production is how fast you can deploy and start your containers based on a production .NET Core image. Therefore, the runtime-only image based on [microsoft/aspnetcore](#) is small so 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 Docker run to processing results. In the

Docker model, there's no need for compilation from C# code, as when running `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 packages.

Although there are multiple versions of the .NET Core and ASP.NET Core images, they all share one or more layers. The amount of disk space needed to store, or the delta (between your custom image and its base image) to pull the image from your registry, is small, because all the images share the same base layer, and might share other layers as well.

When you explore the .NET image repositories at Docker Hub, you will find multiple image versions classified or marked with tags, so you can decide which one to use depending on the version you need, like those in the following table:

| | |
|---|--|
| <code>microsoft/aspnetcore:1.1</code> | ASP.NET Core, with runtime only and ASP.NET Core optimizations, on Linux |
| <code>microsoft/aspnetcore-build:1.0-1.1</code> | ASP.NET Core, with SDKs included, on Linux |
| <code>microsoft/dotnet:1.1-runtime</code> | .NET Core 1.1, with runtime only, on Linux |
| <code>microsoft /dotnet:1.1-runtime-deps</code> | .NET Core 1.1, with runtime and framework dependencies for self-contained apps, on Linux |
| <code>microsoft/dotnet:1.1.0-sdk-msbuild</code> | .NET Core 1.1 with SDKs included, on Linux |

Architecting Container- and Microservice-Based Applications

Earlier in this guide, you learned fundamental concepts about containers and Docker. That was the basic information you need in order to get started. However, enterprise applications can be complex and are often composed of multiple services instead of a single service (container). For those cases, you need to understand additional architectural approaches, such as service-oriented architecture (SOA) and more advanced microservices and container-orchestration concepts. This chapter describes not just microservices on containers, but any containerized application.

Common 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 the process or to batch it.

When you design a container image, you'll see an [ENTRYPOINT](#) definition in the Dockerfile. This 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.

Even though it is not very common, 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. You can, for example, use [Supervisor](#) or a similar tool to take care of launching multiple processes inside a single container. However, even when you can find architectures holding multiple processes per container, that approach but it is not very common.

Containerized 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 monolithic, but structured as several libraries, components, or even layers (application layer, domain layer, data-access layer, etc.). Externally, however, it is 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 scale up, you just add more copies with a load balancer in front. The simplicity comes from managing a single deployment in a single container or VM.

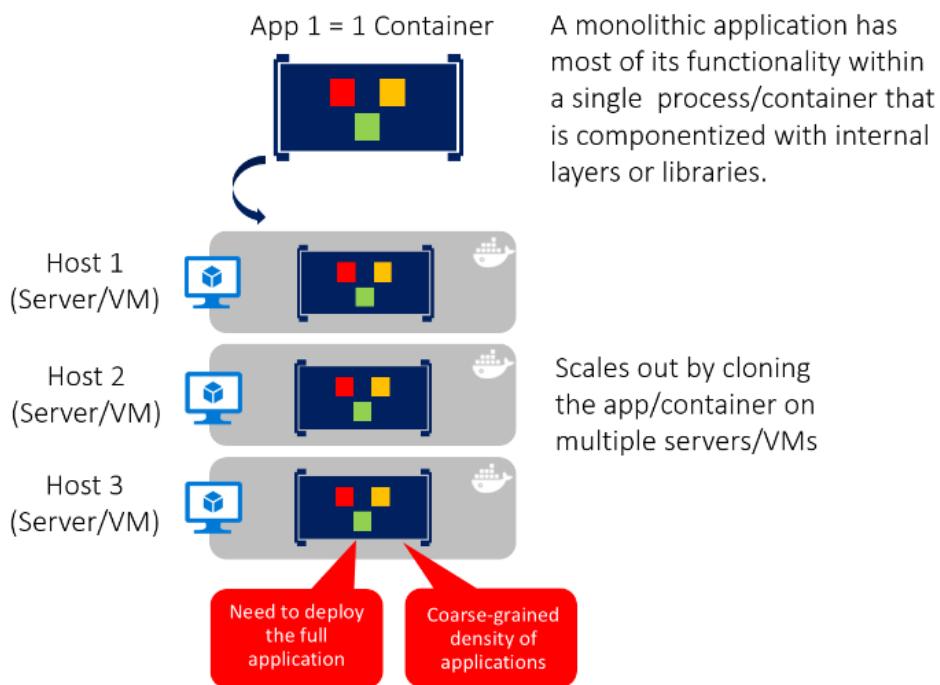


Figure 4-1. Containerized monolithic application architecture example

You can include multiple components, libraries, or internal layers in each container, as illustrated in Figure 4-1. However, this monolithic pattern might conflict with the container principle “a container does one thing, and does it in one process”.

The downside of this approach becomes evident if the application grows, requiring it to scale. If the entire application can scale, it’s not 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 component, 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 likely have only a handful of employees, in a single region, that need to manage the content and marketing campaigns. If you scale the monolithic design, all of the code for these different tasks is deployed multiple times.

There are multiple ways to scale an application, from horizontal duplication, scaling by splitting different areas of the application and finally partitioning or splitting similar things or data. Those possibilities are explained in the article [3 dimensions to scaling](#) from microservices.io.

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 didn't 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. Host running multiple apps/containers

Monolithic applications in Microsoft Azure can be deployed using dedicated VMs for each instance. Additionally, using [Azure VM 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 app lives within a single container (image instance).

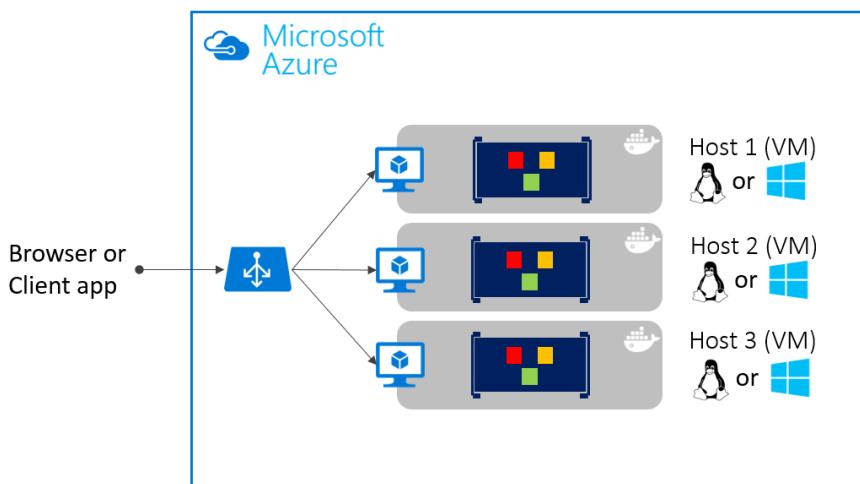


Figure 4-3. Example of multiple hosts scaling up a single container application

Deployment to the various hosts can be managed with traditional deployment techniques. The 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 VM 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 is not 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 might forget to account for some specific configuration or file left on disk.

While monolithic applications can benefit from Docker, we're only touching 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 app to Azure App Service

Whether you want to get a validation of a container deployed to Azure or when an app is simply a single-container app, 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.

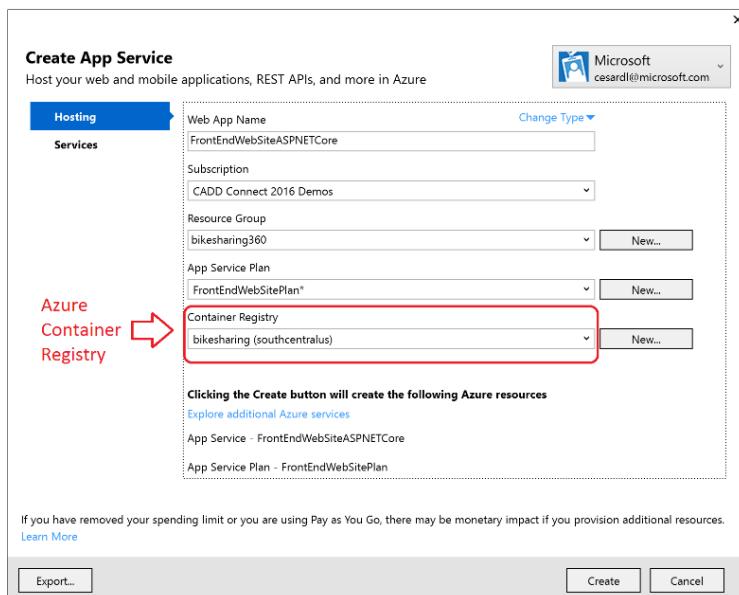


Figure 4-4. Publishing a container to Azure App Service from Visual Studio apps/containers

Without Docker, if you needed other capabilities, frameworks, or dependencies that weren'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 Service Fabric, Azure Cloud Services, or even VMs, where you had further control and you could install a required component or framework for your application.

Container support in Visual Studio 2017 gives you the ability to include whatever you want in your app environment, as shown in Figure 4-4. Since you are running it in a container, if you add a dependency to your app, 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.

Without Docker, if you needed other capabilities, frameworks, or dependencies that weren'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 Service Fabric, Azure Cloud Services, or even VMs, where you had further control and you could install a required component or framework for your application.

Container support in Visual Studio 2017 gives you the ability to include whatever you want in your app environment, as shown in Figure 4-4. Since you are running it in a container, if you add a dependency to your app, 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.

State and data in Docker applications

In most of the cases, you can think of a container as an instance of a process. A process does not 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. Container images, like processes, should be assumed to have multiple instances or that they will eventually be killed; if they're managed with a container orchestrator, it should be assumed that they might get moved from one node or VM to another.

Docker provides a feature named as *overlay file system*. This implements a copy-on-write task that stores updated information to the root file system of the container. That information is additional to the original image on which the container is based. If the container were 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 the container design which by default is stateless.

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

- [Data volumes](#) that mount to the host.

- [Data volume containers](#) that provide shared storage across containers using an external container.
- [Volume plugins](#) that mount volumes to remote services, providing long-term persistence.
- Remote data sources like SQL or NoSQL databases, or cache services like [Redis](#).
- [Azure Storage](#), which provides geo-distributable storage, providing a good long-term persistence solution for containers.

The following provides more detail about these options.

Data 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 happening to a directory on the host OS. This directory is not tied to the lifetime of the container itself, and the directory can be accessed from code running directly on the host OS or by another container that maps the same host directory to itself. 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 within the data volume won't be deleted. The data in a volume can be accessed from the host OS, as well.

Data volume containers are an evolution or improvement over regular data volumes. A data volume container is a simple container that has one or more data volumes within it. The data volume container provides access to containers from a central mount point. This way of access of access is convenient because it is abstracting the location of the original data. Other than that, its behavior is similar to a regular data volume, so data is persisted in this dedicated container independently of the lifecycle of the application's containers.

As shown in the 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 volumes can't access a volume from one host server or VM to another.

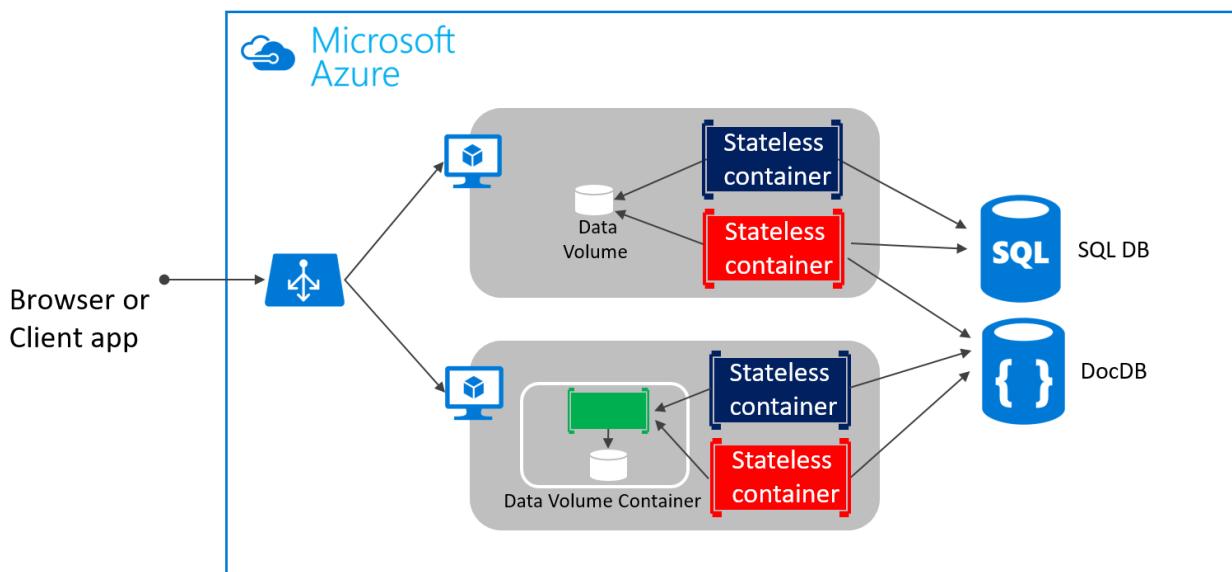


Figure 4-5. Data volumes and external data sources for container-based applications

When using data volumes, it is not possible to manage data shared between containers that run on different Docker hosts. And when Docker containers are managed by an orchestrator, containers are expected to be moved between hosts depending on the optimizations performed by the cluster.

Therefore, it is not recommended to use data volumes for business data, but they are a good mechanism to work with trace files, temporal files, or similar, which won't impact the business data consistency.

Volume plugins like [Flocker](#) provide data access across all hosts in a cluster. While not all volume plugins are created equally, volume plugins typically provide externalized persistent reliable storage from immutable containers.

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

Azure Storage 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 a 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.

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 are deploying based on single Docker hosts. This is where Docker clustering software or an orchestrator will help you out, as explained in later sections where we describe deployment approaches for microservices.

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

Microservices derive from SOA, but at the same time, SOA is different than microservices architecture. Things like big central brokers, central orchestrators at the organization level and the [Enterprise Service Bus \(ESB\)](#) typical in SOA, are in most of the cases anti-patterns in the microservice community. You could 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, each service running in its own process and communicating 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) based on different data storage technologies (SQL, NoSQL) and different programming languages.

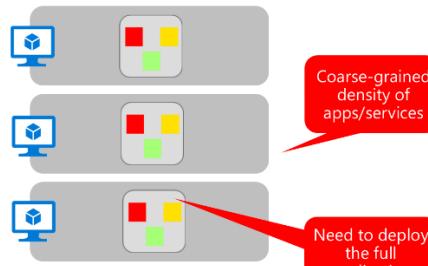
What size should a microservice be? When developing a microservice, size shouldn't be the important point; instead, the important point should be to create loosely coupled services so you have autonomy of development, deployment and scale, per each service. Of course, when identifying and designing microservices, 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 allow granular and autonomous lifecycle per service.

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.

Monolithic deployment approach

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



Microservices application approach

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

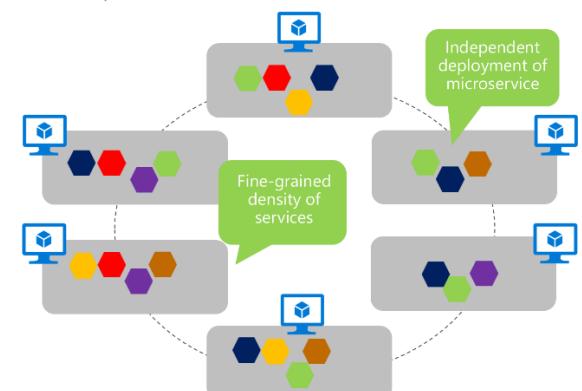
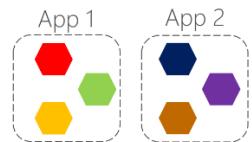


Figure 4-6. Monolithic deployment versus the microservices approach

As Figure 4-6 shows, with the microservices approach, it allows agile changes and rapid iteration per microservice, because you're able to change specific, small areas of complex, large, and scalable applications.

Architecting fine-grained microservices-based applications enables continuous integration and continuous delivery practices, and accelerates delivery of new functions into the application. Fine-grained composition of applications also allows you run and test microservices in isolation, and to evolve them autonomously while maintaining clear contracts between them. As long as you don't break the interfaces or contracts, you can change the internal implementation of any microservice or add new functionality without breaking other microservices.

In order to go into production with a microservices system, the following key topics are enablers that will help you to be successful:

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

Of those topics, 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](#) ebook.

Additional resources

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

Data sovereignty per microservice

An important rule to follow in the 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/behavior). Each DDD Bounded Context would correlate to one business microservice (one or several services). This point about the Bounded Context pattern is explained 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, often a normalized SQL database, for the whole application and all its internal subsystems, as shown in Figure 4-7.

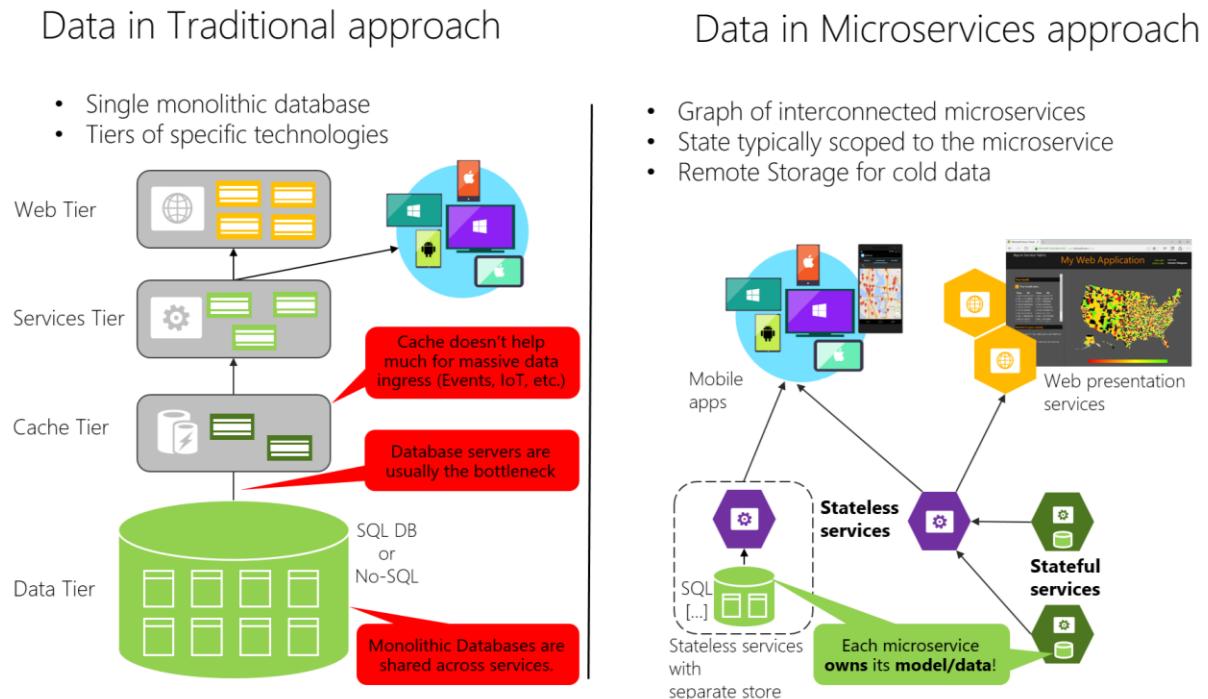


Figure 4-7. Data sovereignty comparison: microservices versus monolithic database

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 are not 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 app. This approach provides a way to easily write a query that combines data from multiple tables.

However, data access becomes much more complex when you move to a microservices architecture. Even when ACID transactions can, and most often should also be used within a microservice or Bounded Context, however, the data owned by each microservice is private to that microservice and can only be accessed via its microservice API. 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, which would break the microservice lifecycle autonomy. But distributed data structures mean that you cannot 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; similarly, many other relational database features are not 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 is not always the best choice. For some use cases, a NoSQL database such as Azure DocumentDB or MongoDB might have a more convenient data model and offer much better performance and scalability than a SQL database like SQL Server or Azure SQL DB. In other cases, a relational DB 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, including loosely coupled services, and better performance, scalability, costs, and manageability. However, it can introduce some distributed data management challenges that will be 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 be named differently between different Bounded Contexts even when different domain entities might share the same identity (that is, the unique ID value with which the entity would be retrieved from persistence). For instance, in a user-profile Bounded Context or microservice, the User domain entity might share identity with the Buyer domain entity in the ordering Bounded Context or microservice.

A microservice is therefore like a Bounded Context, but it also specifies that it is a distributed service. It is built as a separate process per 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 is simply a logical boundary, like a generic subsystem, within a monolithic-deployment application.

It is important to highlight that defining a service per 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 by several physical services. But ultimately, both patterns, Bounded Context and microservice, are closely related.

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

The terms in the ubiquitous language (mainly domain entities) can be named differently within different Bounded-Context even when different domain entities might share the same identity (that is, the unique ID value with which the entity would be retrieved from persistence). For instance, in a user-

profile Bounded Context or microservice, the User domain entity might share identity with the Buyer domain entity in the ordering Bounded Context or microservice (this is actually an example in the [eShopOnContainers](#) application).

Additional resources

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

Logical architecture versus physical architecture

First of all, building microservices does not require to use any specific technology. For instance, Docker Containers are not mandatory in order to create a microservices based architecture. Those microservices could also be runnin as plain processes. Microservices is a logical architecture.

Going further, 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 is not 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 is a difference between an app'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 does not.

Although you might have identified certain business microservices or Bounded Contexts, it doesn't mean that the best way to implement it is always by creating a single service (such as an ASP.NET Web API) or single Docker container per 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 bounded context or business microservice is a logical architecture that could coincide or not with the physical architecture. The important point is that a business microservice (or Bounded Context) has to be autonomous by allowing code and state to be independently versioned, deployed, and scaled.

As shown in figure 4-8, the Catalog business microservice could be composed by several services/processes, like several ASP.NET Web API services or any other kind of services using Http or any other protocol and more importantly, sharing the same data, as long as these services are very cohesive around the same business domain.

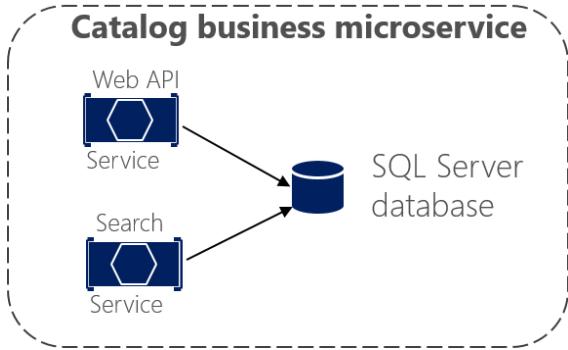


Figure 4-8. Busines microservice with several physical services/containers

Those services in the example would be sharing the same data model because the REST Catalog API service targets the same data than the Search service. But from a physical implementation point of view (service or container) you are splitting that functionality so you can scale up/down each service depending on the needs. Maybe the regular Catalog REST API service usually needs to have many more instances than the search service, or viceversa.

In short, the logical architecture of microservices doesn't always have to coincide with the physical/deployment architecture. In this guide, whenever we are mentioning a microservice, we mean a business/logical microservice that could map to a single service or more. In most of the cases it will be a single service, but depending on the case, could be more.

Challenges and solutions for distributed data management

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

This is probably the first challenge anyone encounters. Each microservice has to be a piece or area of your application and each microservice should be autonomous with all the benefits and challenges that it conveys. But, how do you actually identify and define those boundaries? That is a great question.

First of all, you need to focus on the application's logical domain models and related data and try to identify decoupled islands of data, different contexts within the same application. Each context could have a different business language (meaning different business terms). Those contexts should be defined and managed independently. The terms and entities used in those different contexts might sound similar but soon you can start identifying that in a particular context a similar business concept with the same identity is used for different purposes and tasks and even called in a different way than in another context. For instance, a user can be referred as a plain user in the identity/membership context, as a customer in a CRM context, as a buyer in an ordering context and so forth.

The way you identify those boundaries between the multiple application's contexts with a different domain per context is precisely how you can identify the boundaries for each business microservice and its related domain model and data underneath, always attempting to minimize the coupling between those islands or microservices. This guide tries to drill down into this identification and domain model design in the upcoming section named "*Identifying domain-model boundaries for each microservice*".

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

A second challenge is the question of how you can implement queries that retrieve data from several microservices while avoiding chatty and inefficient communication from remote client apps that are talking to those microservices. An example could be a single screen from a mobile app that needs to show user info 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 will 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 coming from multiple microservices with different databases owned by each microservice, the recommended approach would be to handle the data aggregations in aggregation microservices named API Gateway. However, you need to be really careful about this pattern as it can be a pinch point in your system and a microservice autonomy violator. In order to limit that case, you might want to have multiple fined-grained API Gateways. This approach, the API Gateway pattern, is explained in the *API Gateway* section in further details, when talking about inter-microservice communication.

CQRS with query/reads tables. This solution for aggregating data from multiple microservices is also known as the [Materialized View pattern](#) that generates, in advanced, a read-only table with the data owned by multiple microservices. That table will have a format suited to the client app's needs. This is a good approach when the data query/join across multiple microservices would have a poor performance if done in real time due to the nature of the data distributed across the mentioned microservices. Think about a similar case in regards data needs but using a single database. In that case, you would use a complex join that you implement with a SQL query involving multiple tables. However, when handling multiple databases, each database owned by a different microservice, and you cannot query those databases to make a SQL join, complex queries involving multiple microservices become an important challenge. Therefore, you can address the requirement with a CQRS approach by creating a denormalized query table in a different database that is used just for queries. That table will be designed based on the data you need for that complex query, with a one-to-one relationship between fields needed by your application's screen and the columns in the query table.

This approach not only solves the original problem but also improves the application performance considerably when compared with a complex relational join targeting multiple tables, because you already have the query result persisted in an ad-hoc table for that query. Of course, using CQRS with query/reads tables means additional development work, and you again need to embrace eventual consistency. But performance and high scalability and the fact that you have data split in multiple databases might require these types of approaches.

"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 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 don't 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 you constantly need to aggregate information from multiple microservices for complex queries needed by your application, that 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 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 by this approach 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 their stock level. The Ordering microservice manages orders and must verify that a new order doesn't exceed the available catalog product's stock (it could also have been logic that handles backordered products, it depends on your domain). In a hypothetical monolithic version of this app, the Ordering subsystem could simply use an ACID transaction to check the available stock, create the order in the Orders table, and update the available stock in the Products table.

However, in a microservices architecture based application, the Order and Product tables are owned by their respective microservices, so one microservice should never include databases owned by other microservice in their transactions, neither in their queries, as shown in Figure 4-9.

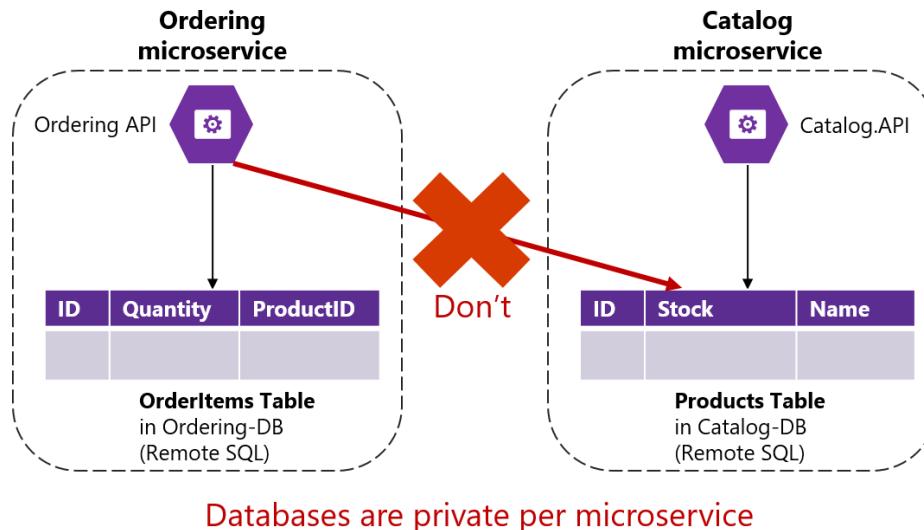


Figure 4-9. A microservice cannot directly access a table in another microservice

The Ordering microservice should not update the Products table directly, because the Products table is owned by the Catalog microservice. To make an update to the Catalog microservice, the Ordering microservice should only do it through asynchronous communication as when using integration

events (message/event based communication), such as performed at the [eShopOnContainers reference application](#).

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 apps must remain up and running, and developers can work around strong consistency by using techniques for working with weak or eventual consistency. This is precisely the approach taken by most microservice-based architectures. Even more, ACID and two-phase commit transactions are not just against microservices principles but also most NoSQL databases (like Azure Document DB, MongoDB, etc.) do not support two-phase commit transactions, neither. 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 should microservices communicate across boundaries?

This is a critical question and a real challenge. It doesn't really mean what communication protocol you should use (Http/REST, AMQP/Messaging, etc.) but what communication style you'll use and most of all how much coupled your microservices will be. Depending on that, when failure starts to surface, the consequences and impact on your system will vary significantly.

In a distributed system, like a microservices-based application, with so many artifacts moving around and distributed services across many servers/hosts, things will fail, eventually.

Partial failure and even larger outages will happen, so you need to embrace those facts and design your microservices and the inter communication across them based on precisely the risks common in this type of distributed system.

A very popular approach is to implement Http/REST based microservices due to their simplicity. To use Http is perfectly valid, the issue/risk here is related to how you use it. If you use request/response Http calls just to interact with your microservices from the client applications or the API Gateways, that is perfectly fine. But, if you start creating a chain of synchronous Http calls across multiple microservices, communicating them across their boundaries like if they were objects within a monolithic application, that will be a serious problem.

For instance, let's say that your client application calls an Http API at the Gateway API or any initial microservice like the Ordering microservice. If from that point, the Ordering microservice continues to call additional microservices through Http within the same request/response cycle, then you will be creating a chain of Http calls. It might sound fair, initially, however, there are important points to consider when going this path:

- *Blocking and low performance.* Due to the synchronous nature of Http, the original request/response won't return until all the internal Http calls are performed. Imagine if these calls increase significantly because of any reason and at the same time one of the intermediate http calls to a microservice is blocked. That means the performance will be

impacted and the overall scalability will be exponentially affected while additional Http requests increase.

- *Coupling microservices with Http*: Business microservices should not be coupled with other business microservices. Ideally, they should not know about the existence of additional microservices. If you do, like when implementing a chain of Http calls across microservices, achieving real autonomy per microservice will be almost impossible and most of all, failure in one microservice will impact the rest of the chain.
- *When failure arises in any microservice*. If you implemented a chain of microservices linked by http calls, when any of the microservices fails (and they will fail, for sure, eventually) 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 when you might implement client logic with retries with exponential backoff or circuit breaker mechanisms, the more complex those http call chains are, the more complex will be to implement that failure strategy based on http.

You could argue that if your internal microservices are communicating via Http, as described, by creating chains of Http requests, then the desired microservice autonomy is impacted and you still will have a monolithic application but based on Http instead of intra-process communication mechanisms.

Of course, communication between the client apps, the optional API Gateway and the first level of microservices will be usually done using Http. The issues will arise if you expand the request/response cycle as a chain across additional internal microservices.

Therefore, in order to enforce microservices' autonomy and have a better resiliency, if possible, you should minimize the use of chains of request/response communication across the internal microservices and try to use only asynchronous interaction for inter microservice communication, either by using asynchronous message/event based communication or through Http polling but out of the original http request/response cycle. At least, try to minimize, as much as possible, those chains of Http calls across the internal microservices.

This approach is explained with additional details in the sections "*Asynchronous microservice integration enforcing 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://msdn.microsoft.com/en-us/library/dn589800.aspx>
- **Martin Fowler. CQRS (Command and Query Responsibility Segregation)**
<http://martinfowler.com/bliki/CQRS.html>
- **Materialized View**
<https://msdn.microsoft.com/en-us/library/dn589782.aspx>
- **Charles Row. ACID vs. BASE: The Shifting pH of Database Transaction Processing**
<http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/>
- **Compensating Transaction**
<https://msdn.microsoft.com/en-us/library/dn589804.aspx>
- **Udi Dahan. Service Oriented Composition**
<http://udidahan.com/2014/07/30/service-oriented-composition-with-video/>

Identifying domain-model boundaries for each microservice

The goal when identifying model boundaries and size for each microservice is not 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 is not on the size, but instead on the business capabilities. Also, if there is clear cohesion needed for a certain area of the application based on a high number of dependencies, that should probably be a 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 is not 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 pattern (part of Domain-Driven Design), as introduced earlier. Sometimes, a BC could be composed by several physical services, but not vice-versa.

A domain model with specific domain entities applies within a concrete bounded context or microservice. A bounded context 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, which are the same goals for microservices.

One other tool that informs about 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 it happens the opposite and the company's organization is formed by the software, so you need to reverse Conway's law and build the boundaries the way you want the company to be organized leaning towards business process consulting.

In order to identify bounded contexts, a DDD pattern that can be used for this is the [Context Mapping pattern](#). With Context Mapping, you identify the various contexts in the application and their boundaries. It is common to have a different context and boundary per small subsystem, for instance. The Context Map is a way to define and make explicit those boundaries between domains. A Bounded Context is autonomous and includes the details of a single domain, details like the domain entities, and defines integration contracts with other bounded contexts or subsystems. This is very similar to the definition of a microservice: it is autonomous, it implements certain domain capability and it must provide interfaces. This is why Context Mapping and the bounded Context pattern good approaches for identifying the domain model boundaries of your microservices.

When dealing with a large application, you will 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 is very hard to disambiguate all domain terms across all the domains related to a large application, but the most important thing is that you should not try to unify the terms but embrace the differences and richness provided by each domain or area of the system. If you try to have a single unified database for the whole application, that unified vocabulary will be awkward and won't sound right to none of the multiple domain experts. Therefore, bounded contexts (implemented as microservices) will help you to clarify where you can use certain domain terms and where you will need to split the system and create additional bounded contexts with different domains.

You will know if you got to the right boundaries and sizes of each bounded context and domain model if you have few strong relationships between domain models and you don't usually need to merge information from multiple domain models when performing typical application operations.

Perhaps, the best answer to the question of how big a domain model of each microservice should be is the following: it should have an autonomous and as much isolated bounded context as possible that will enable you to work without having to constantly think or switch to other contexts (other microservice's models). In Figure 4-10 you can see how multiple microservices (multiple bounded contexts) each have their own model and how their entities can be defined, depending on the specific requirements for each of the identified domains in your application.

Identifying a domain model per microservice or Bounded Context

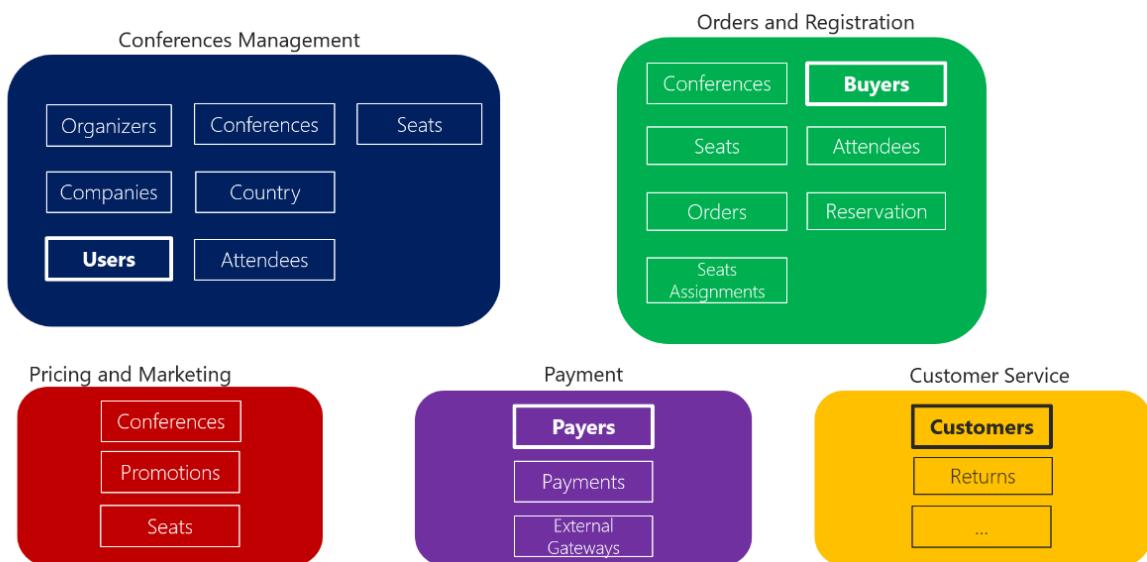


Figure 4-10. Identifying entities and microservice model boundaries

Figure 4-10 illustrates a sample scenario related to an online conference management system. You have identified several bounded contexts that could be implemented as microservices, based on multiple identified domains that domain experts defined for you. As you can see, there are entities that are present just in a single microservice's model, like Payments in the Payment microservice or subsystem. 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 the Figure 4-11.

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

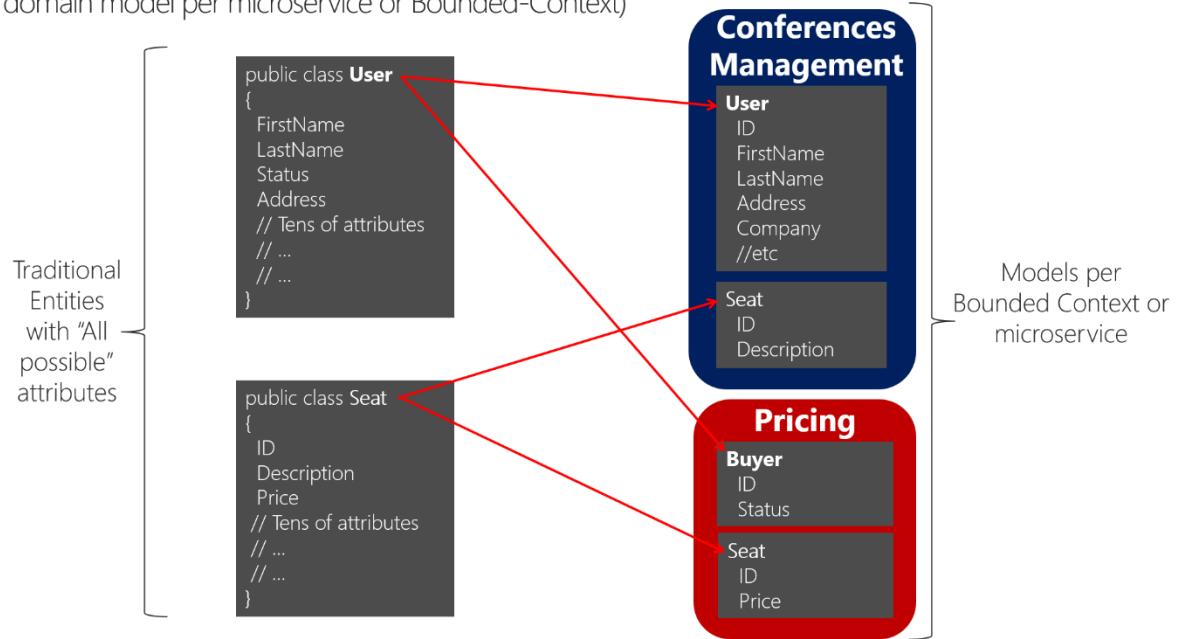


Figure 4-11. Decomposing traditional data models into multiple domain models

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 is actually a buyer. Each microservice or bounded context 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 don't need the address or the ID of the user, just 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 is 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 of not sharing the same user entity with the same number of attributes across domains (services). One benefit is to reduce duplication, so each microservice's model don't have data that it doesn't need. Another benefit is having a master microservice that owns a certain type of data per entity so that updates and queries for a certain type of data are driven only by that microservice.

Direct client-to-microservice communication versus the API Gateway pattern

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.

Direct Client-To-Microservice communication Architecture

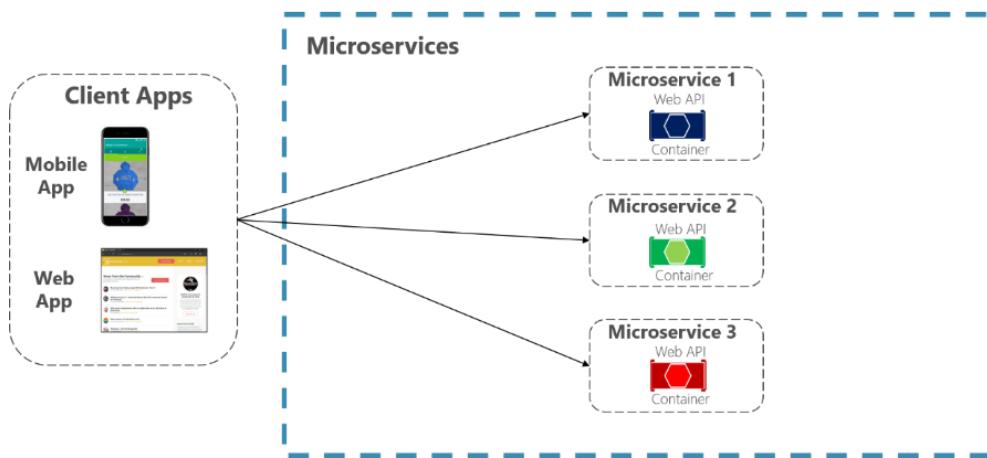


Figure 4-12. Using a direct client-to-microservice communication architecture

In this approach, each microservice has a public endpoint, sometimes with a different TCP port for each microservice. An example of an 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 distributes the requests across the microservices. Even going further, in production environments you could have an ADC (*Application Delivery Controller*) like [Azure Application Gateway](#), in between your microservices and the Internet acting as a transparent tier with not just load balancing capabilities but also securing your services and offering SSL termination which improves the load of your hosts by offloading CPU intensive SSL termination to the application gateway plus other routing capabilities. In any case, this load balancer and ADC would be transparent from a logical Application architecture point of view.

This direct client-to-microservice communication architecture is good enough for a small microservice-based application. However, when you build large and complex microservice-based applications (for example, when handling tens of microservice types), that approach faces possible issues as explained in the following cases. You need to consider the following questions when developing a large application based on microservices:

- *How can client apps minimize the number of requests to the backend and reduce chatty communication to many microservices?*

Requiring interaction with multiple microservices to build a single UI screen increases the number of required roundtrips across the Internet. This increases latency and complexity in the UI side. Ideally, responses should be efficiently aggregated in the server side—this 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, in order 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 usually 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 façade 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 data responses can be more efficient, probably by aggregating data from multiple microservices and returning a single set of data, compressing that data and sometimes even eliminating part of the data in the response that is not needed by the mobile app. Again, a façade or API in between can be very convenient for this scenario.

Using an API Gateway

When you design and build large or complex microservice-based applications with multiple client apps, a good approach to consider for your architecture can be an [API Gateway](#). This is a service that provides a single entry point for certain groups/sets of internal application's microservices. It is 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 named as “Back end for Front End” as you build it while thinking about the client apps' needs.

Figure 4-13 shows how an API Gateway can fit into a microservice-based architecture.

Using the API Gateway Service

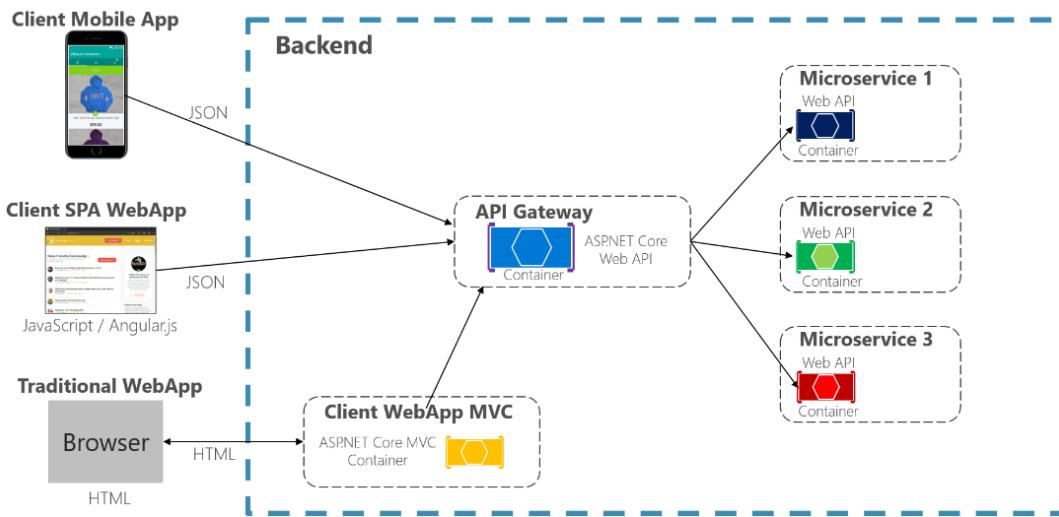


Figure 4-13. Using the API Gateway pattern in a microservice-based architecture

In this example, the API Gateway would be implemented as a custom Web API service running as a container.

It is also important to highlight that you should implement several API Gateways so you have different façades depending on the needs from each client app. Each API Gateway would provide a different API tailored for each client app or even depending on the form-factor or device by applying specific adapter code which underneath will be calling multiple internal microservices.

But since the API Gateway is actually an aggregator you need to be careful with it. Usually, it won't be a good idea to have a single API Gateway aggregating all the internal microservices of your application as it would act as a monolithic aggregator or orchestrator and will violate the microservice's autonomy by coupling all the microservices. Therefore, the API Gateways should be segregated based on business boundaries and needs but not as an aggregator for the whole application.

Sometimes, a granular API Gateway can also be a microservice by itself, even with a domain/business name and with some data related, e.i. a Redis Cache, as example. And having those boundaries dictated by the business/domain will help you to get a better design.

Even going further, this granularity in the API gateways tier can be especially useful for more advanced *Composite UI applications* based on microservices introduced in a later section named "*Composite UI based on microservices*", because the concept of a fine-grained API Gateway is very similar to an *UI Composition* service.

Therefore, for many medium and large size applications, using this custom-built API Gateway pattern is usually a good approach, but not as a single monolithic aggregator or unique central backend.

Another approach is to use a product like [Azure API Management](#) as shown in figure 4-14. This approach not only solves your API Gateway needs, but provides features like gathering insights from your APIs.

It is worth to say that if using an API management solution, an API gateway is only a component within a full API management solution that offer many more features. They are not exactly synonymous but the API Gateway could be considered a subset contained as part of a API Management solution.

API Gateway with Azure API Management Architecture

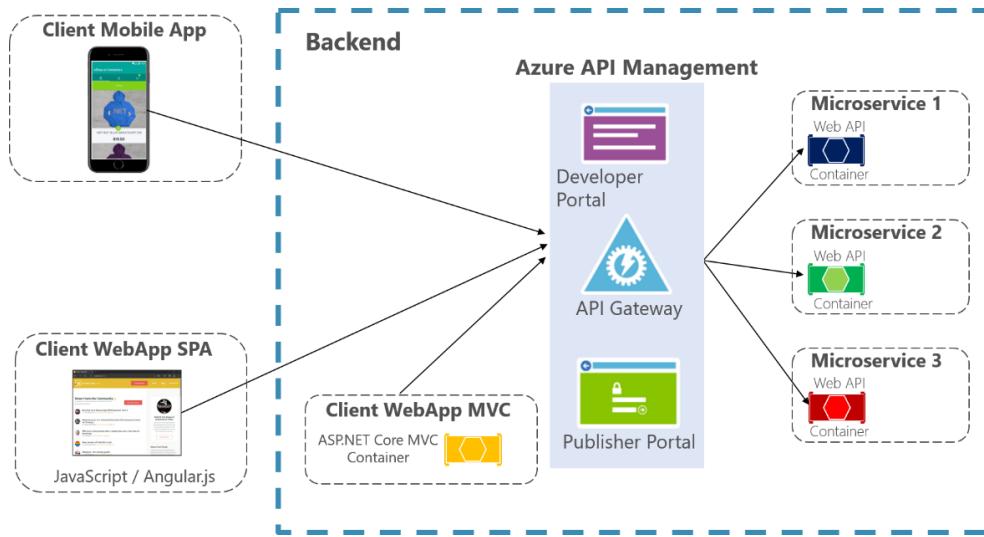


Figure 4-14. Using Azure API Management for your API Gateway

These insights help you get an understanding of how your APIs are being used and how they are performing 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) we are limiting the architecture 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 review and adopt Azure API Management as the base for your API Gateways.

Drawbacks in the API Gateway pattern

- The most important drawback, which has to be handled carefully is that, when implementing an API Gateway, you are coupling that tier with the internal microservices. This complement might derive into real danger as it could be becoming "the new ESB", as explained at Clemens Vasters' session named "[Messaging and Microservices](#)" at GOTO 2016.
- Using a microservices API gateway creates an additional possible point of failure.
- Increased response time due to the additional network call through the API gateway microservice. However, this extra call is usually less impactful than having a too chatty interface if directly calling the internal microservices from the client apps.

- Possibility of bottleneck if not scaled-out properly
- Additional development cost and future maintenance for the API Gateway microservices if you built custom logic and data aggregation at the API Gateway level. Thus, developers must update the API Gateway in order to expose each microservice's endpoints. Going further, implementation changes in the internal microservices will cause code changes at the API Gateway level. However, if the chosen API Gateway is just applying security, logging and versioning (like when using Azure API Management), this additional development cost might not apply.
- Management bottleneck if the API Gateway is managed/developed by a single team. This is why a better approach is to have several fined-grained API Gateways depending on the client apps needs. You could also segregate the API Gateway internally in multiple areas/layers owned by the several teams working on the internal microservices.

Additional resources

- **Charles Richardson. Pattern: API Gateway / Backend for Front-End**
<http://microservices.io/patterns/apigateway.html>
- **Azure API Management**
<https://azure.microsoft.com/en-us/services/api-management/>
- **Udi Dahan. Service Oriented Composition**
<http://udidahan.com/2014/07/30/service-oriented-composition-with-video/>
- **Clemens Vasters. Messaging and Microservices at GOTO 2016**
<https://www.youtube.com/watch?v=rXi5CLjlQ9k>

Communication between microservices

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 are creating objects with code (for example, `new ClassName()`), or can be invoked in a decoupled way if you are 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. If you don't design your distributed system properly you will face all the issues explained at "[The fallacies of distributed computing](#)".

The solution is not unique but multiple. It lies on isolating the business microservices as much as possible, using asynchronous communication between microservices and replacing fine-grained communication (typical in intra-process communication between objects) with a coarser-grained communication, by grouping calls, and by returning to the client sets of data that aggregate multiple internal calls.

A microservices-based application is a distributed system running on multiple processes or services, usually even across multiple servers/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 "[smart endpoints and dumb pipes](#)". This slogan encourages a design that is 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 or dimensions.

The first axis is defining 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 is independent of the client code execution that could be synchronous (thread is blocked) or asynchronous (thread is not blocked, response will get to a callback, eventually). The important point here is that the protocol (HTTP/S) is synchronous and the client code can only continue its task when it receives the HTTP server response.
- *Asynchronous protocol.* Other protocols like AMQP (protocol supported by many operating systems and cloud environments) using messaging are asynchronous. The client code or message sender usually won't wait for a response. It will just send the message like when sending a message to a RabbitMQ queue or any other message broker.

The second axis is defining 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 from none to multiple receivers or services. 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 is 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/S when invoking a regular Web API HTTP service. Also, messaging protocols for asynchronous communication between microservices.

However, these dimensions axes are good to know so you have clarity on the possible communication mechanisms to use, but they are not the important concerns when building microservices. Neither the synchronicity related to the client thread execution nor the synchronicity related to the protocol are the important points when integrating microservices, but only if you are able to integrate your microservices asynchronously while maintaining the microservices independent with no direct dependencies between each other, as explained in the following section.

Asynchronous microservices integration enforcing 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 less inter communications between microservices, the better. But, of course, in many cases you will need somehow to integrate the microservices. When you need to do that, the critical and mandatory rule here is that the communication between the microservices should be asynchronous. That doesn't mean that you have to use one or other protocol (asynchronous protocol versus synchronous protocol), it just means that the communication between microservices should be done only by propagating data asynchronously, but never depending on other microservices for the initial service's 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 from an original microservice you think you need to call other microservices (like performing a HTTP request for a data query) in order to be able to provide a response to the client application, that means you have an architecture that won't be resilient when some microservices fail.

Even more, having dependencies between microservices (like doing Http requests between them for querying data) not only makes your microservices not autonomous/independent but their performance will be impacted, the overall response time for the client apps will be getting worse the more you add synchronous dependencies (like query reauests) between them.

If your microservice needs to raise any additional action to another microservice, don't do that synchronously and as part of the original microservice request/reply operation, do it asynchronously by using any technique, whether it is based on asynchronous messaging or integration events, queues or any other way but out of the original synchronous request/reply operation.

And finally, and this is where most of the issues arise when building microservices, if your initial microservice needs data that is originally owned by other microservices, simply replicate or propagate that data (only the attributes you need) into your service's database by using eventual consistency (commonly, by using integration events, as explained in upcoming sections).

As introduced in the section "Identifying domain-model boundaries for each microservice", duplicating some data across several microservices is not a wrong design but healthy, as when doing that you can translate that data into the specific language or terms of that additional Domain or Bounded Context. For instance, in the [eShopOnContainers](#) application you have an initial microservice named Identity.API 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 just a few attributes needed by the Ordering domain since it might not need to have the whole user profile.

The protocol you might use to communicate and even propagate data asynchronously across microservices in order to have eventual consistency, can be any. As mentioned, you could use integration events using an event bus or message broker (async push way based on a

publish/subscription approach) 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 to be used 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 are using a synchronous request/response-based communication mechanism, protocols such as HTTP and REST approaches are the most common, especially if you are publishing your services outside the Docker host or microservice cluster. If you are communicating between services internally (within your Docker host or microservices cluster) you might also want to use binary format communication mechanisms (like Service Fabric remoting or 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 is not a standard, it is 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 (Docker orchestrators or Azure Service Fabric), or for proprietary client applications that talk to the microservices.

Request/response communication with HTTP and REST

When using a request/response communication, a client 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 real-time UI (a live user interface) from client apps. Therefore, in a microservice architecture you will probably use this communication mechanism for most queries, as shown in Figure 4-15.

Request/Response Communication for Live Queries and Updates HTTP and REST based Services

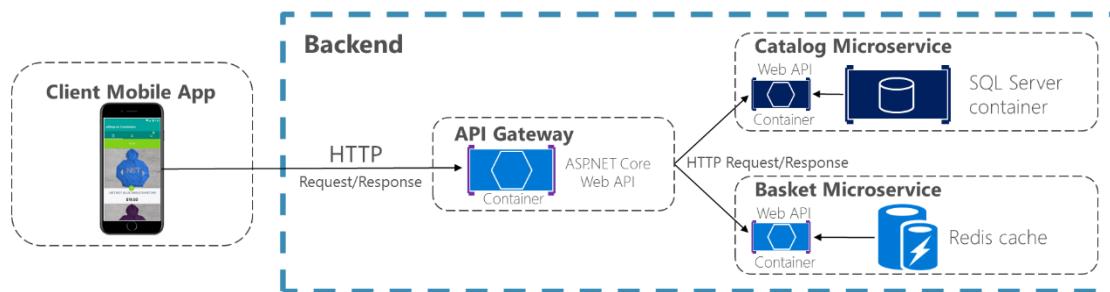


Figure 4-15. Using HTTP request/response communication (synchronous or asynchronous)

When using a request/response communication, the client 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 explained in the next section.

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

There is 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.
<http://martinfowler.com/articles/richardsonMaturityModel.html>
- **Swagger.** The official site.
<http://swagger.io/>

Push and real-time communication based on HTTP

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

As shown in the figure 4-16, 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.

Push and real-time communication based on HTTP

One-to-many communication

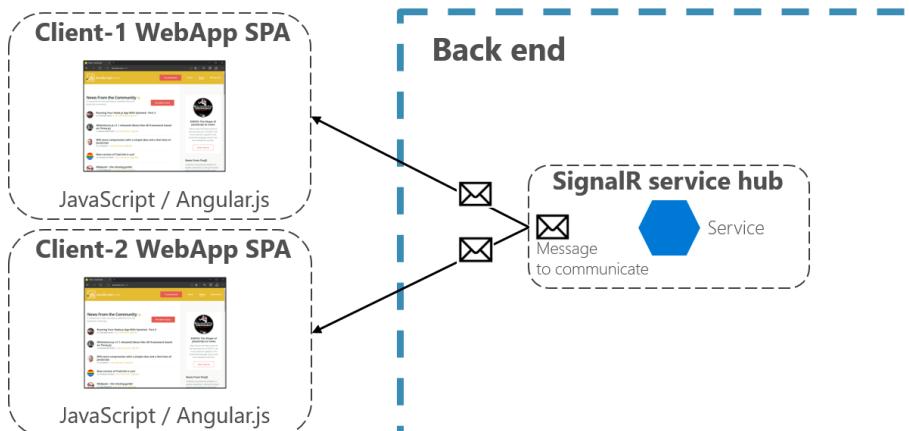


Figure 4-16. One-to-one real-time asynchronous message communication

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 WebSocket 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 when discussing microservices and Bounded Contexts, models (User, Customer, Product, Account, etc.) can mean different things to different microservices or Bounded Contexts. That means that you need some way to reconcile changes across the different models when changes occur. 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 is a message-based communication, the client assumes that the reply will not 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, very different than the typical SOA large brokers and orchestrators. 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 async messaging between the internal services and sync communication (such as Http) only from the front-end and client apps.

There are two kinds of messaging communication: Single receiver message-based communication and Multiple receivers message-based communication.

Single receiver message-based communication

A single receiver means there is a point to point communication which delivers a message to exactly one of the consumers that is reading from the channel, so the message should be processed just once. But there are special situations that you also need to take into account. 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.

Message-based asynchronous communication with a single receiver is especially well suited for sending an asynchronous *command* or *trigger* from one microservice to another. For example, if you want to trigger any action in another microservice (a single one) once you are already in an asynchronous and message based business process, you should avoid to mix message-based communication with synchronous Http communication. Therefore, that kind of trigger or command

should be based on asynchronous communication based on messages sent across microservices, like a message-based command. Figure 4-17 illustrates this approach.

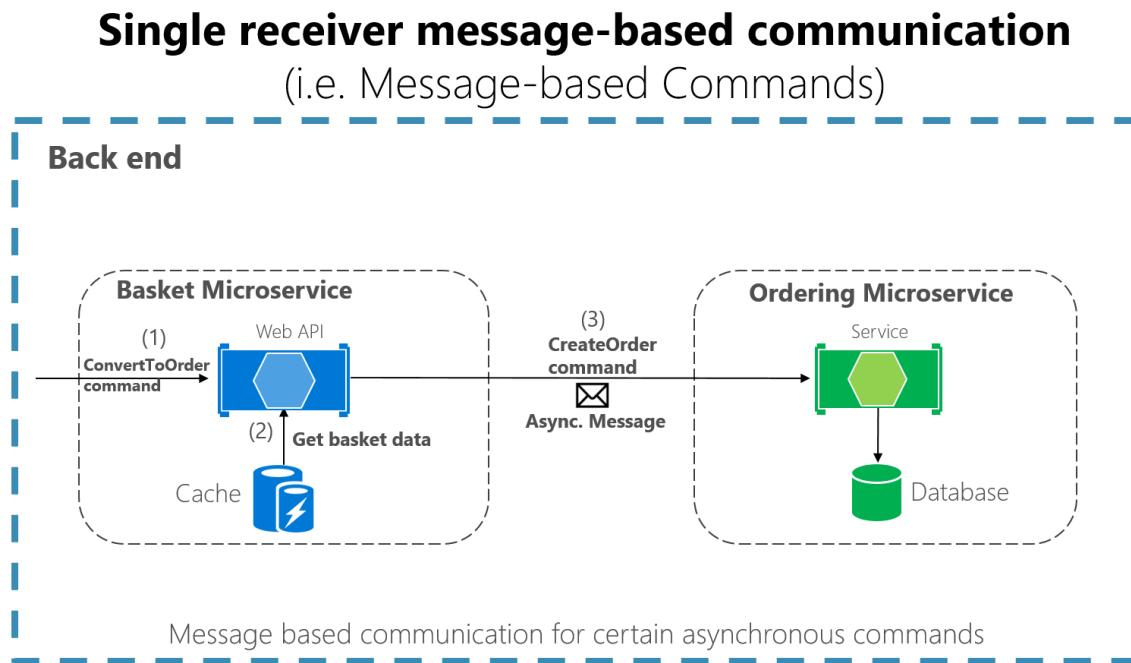


Figure 4-17. A single microservice receiving an asynchronous message

It is also worth noting that, when the commands are coming from the client applications, they can be implemented as Http synchronous commands. You should use message-based commands when you need higher scalability or when you are 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 this type of communication and architectural approach, a microservice publishes an integration event when something happens within its domain and any other microservice needs to be aware of, 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 to be published. This publish/subscribe system is usually performed by using an implementation of an event bus. The event bus can be designed as an abstraction or interface with the API 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.

A good recommendation is to make sure that this sort of eventual consistency driven by integration events has to be completely obvious to the end user and not try to fake it with push systems like SignalR, or polling refresh 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 is explicit.

As noted earlier in the "[Challenges and solutions for distributed data management](#)" section, you can use integration events to implement business tasks that span across multiple microservices. Thus, you will 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 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. For doing so, you can use the publish/subscribe messaging based on event-driven communication, as shown in Figure 4-18. This publish/subscribe mechanism is not exclusive to the microservice architecture. It is 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 [CQRS \(Command and Query Responsibility Segregation\)](#) architecture pattern. The goal is to have eventual consistency between multiple data sources across your distributed system.

Asynchronous event-driven communication

Multiple receivers

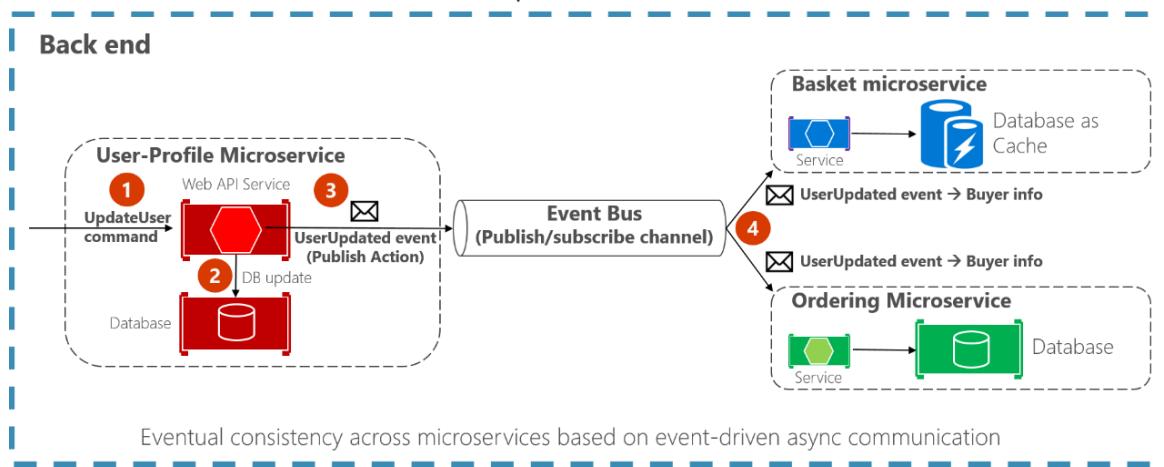


Figure 4-18. Asynchronous event-driven message communication

Your implementation will determine what protocol to use for event-driven, message-based communications. [AMQP](#) can help achieve reliable queued communication.

When using 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 available messaging technologies for implementing your abstract event bus are at different levels. For instance, products like RabbitMQ (a messaging broker transport) and even Azure Service Bus sit at a lower level than other products like, NServiceBus, MassTransit or Brighter which can work on top of either RabbitMQ and even on top of 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 we've 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, 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.

Finally, additional topics to consider when using asynchronous communication are message idempotence and messages deduplication, both topics covered in the section named *Implementing event-based communication between microservices (integration events)*, whtin this guide.

Additional resources

- **Event Driven Messaging**
http://soapatterns.org/design_patterns/event_driven_messaging
- **Publish/Subscribe Channel**
<http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- **Chris Richardson. Pattern: Command Query Responsibility Segregation (CQRS)**
<http://microservices.io/patterns/data/cqrs.html>
- **Command and Query Responsibility Segregation (CQRS)**
<https://msdn.microsoft.com/en-us/library/dn568103.aspx>
- **Communicating Between Bounded Contexts**
<https://msdn.microsoft.com/en-us/library/jj591572.aspx>
- **Eventual consistency**
https://en.wikipedia.org/wiki/Eventual_consistency

- **Philip Brown. Strategies for Integrating Bounded Contexts**
<http://culttt.com/2014/11/26/strategies-integrating-bounded-contexts/>
- **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 will be able to evolve a microservice independently only if you don't 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 are using. For instance, if you are using messaging (like [AMQP](#)), the API consists of the message types. If you are using HTTP and RESTful services, the API consists of the URLs and the request and response JSON formats.

However, even if you are 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 is 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 continue to 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 are using an HTTP-based mechanism such as REST, one approach is to embed the API version number in the URL or into a `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. For this a good approach is to use the [mediator pattern](#) (i.e. [MediatR library](#)) to decouple the different implementation versions into independent handlers.

Finally, it is worth to highlight that when using a REST architecture approach, [Hypermedia](#) is the best solution for versioning your services and it allows evolvable APIs.

Additional resources

- **Scott Hanselman. ASP.NET Core RESTful Web API versioning made easy**
<http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>
- **Versioning a RESTful web API**
<https://docs.microsoft.com/en-us/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 is used to resolve its location. Your microservice needs to be addressable wherever it is 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 are running on. This implies that there is an interaction between how your service is deployed and how it is 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 become unable to discover service instances. Consequently, 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, within an Azure Container Service environment, Kubernetes and DC/OS with Marathon can handle service instance registration and deregistration. They also run a proxy on each cluster host that plays the role of server-side discovery router. Another example is Azure Service Fabric, which also provides a service registry through its out-of-the-box Naming Service.

It is worth to mention that there is certain overlap between the service registry and the API gateway pattern which helps to solve this problem, as well. For example, the [Service Fabric Reverse Proxy](#) is a type of implementation of a Gateway API that based on the Service Fabric Naming Service helps to resolve address resolution to the internal services.

Creating composite UI based on microservices, including visual UI shape and layout generated by multiple microservices

Microservices architecture often starts with the server side handling data and logic. However, a more advanced approach is to design your app 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 together.

Figure 4-19 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/JS. The figure is a simplification that highlights that you have a single client UI (monolithic) consuming the microservices, which just focus on logic and data and not on the UI shape (HTML/JS).

Monolithic UI consuming microservices

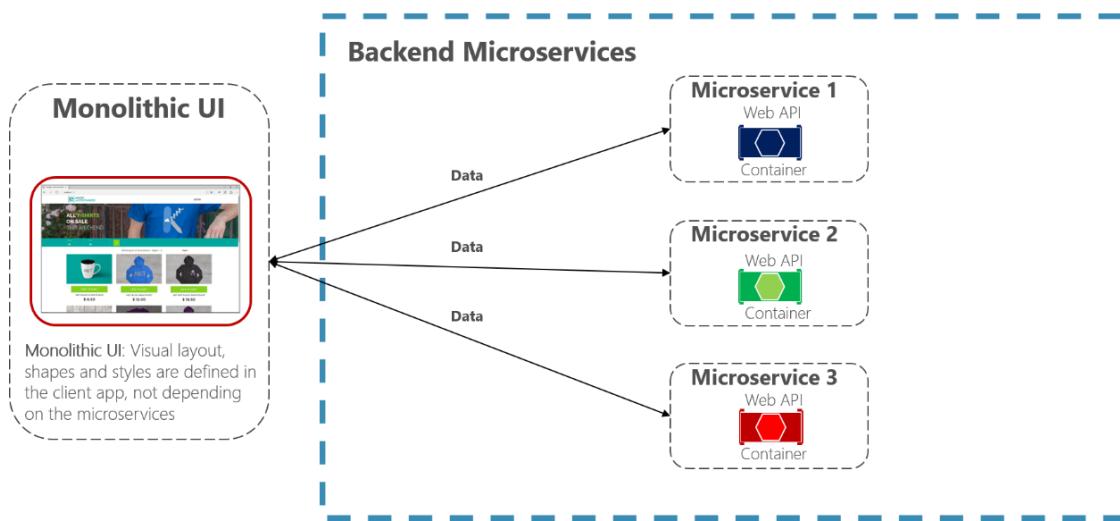


Figure 4-19. A monolithic UI application consuming backend microservices

In contrast, a composite UI is precisely generated and composed by actual microservices. Each microservice drives the visual shape of a specific area of the UI. The key difference is that you will have client UI components (TS 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 visually.

Figure 4-20 shows a version of this composite UI approach. It's simplified, because you might have other microservices that are aggregating granular parts based on different techniques—this depends on whether you are building a traditional web approach (ASP.NET MVC) versus a SPA (Single Page Application).

Composite UI generated by microservices

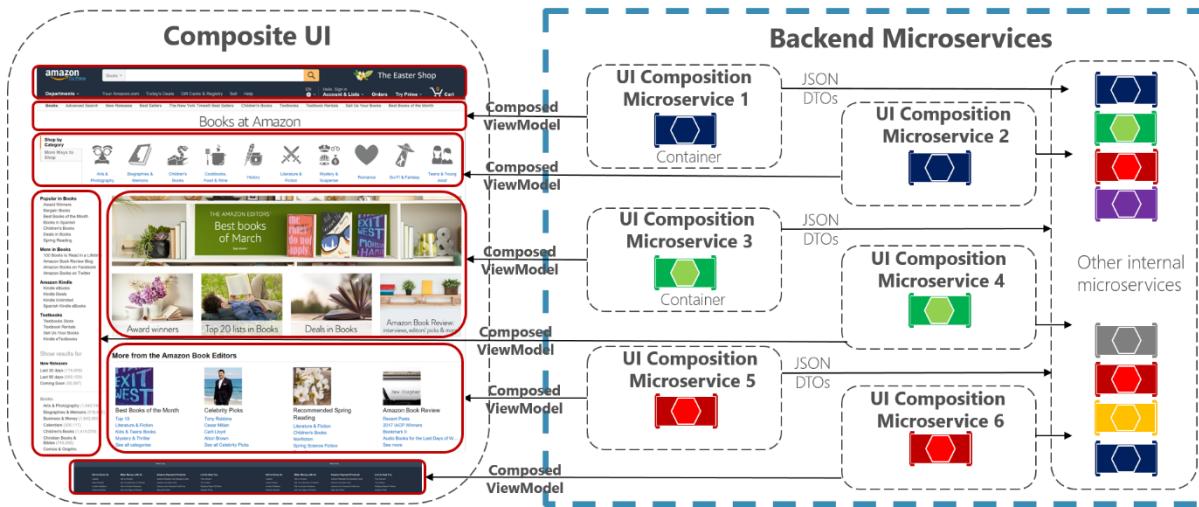


Figure 4-20. Example of a composite UI application shaped by backend microservices

Connecting back to the AP Gateway pattern, each of those UI composition microservices would be very similar to a small API Gateway but in this case responsible for a small UI area.

A composite UI approach that's driven by the microservices can be more or less challenging depending on what UI technologies you are using. For instance, you won't use the same techniques for building a traditional web application than for building a 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 is 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

- **Composite UI using ASP.NET (Particular's Workshop)**
<https://github.com/Particular/Workshop.Microservices/tree/master/demos/CompositeUI-MVC>
- **Ruben Oostinga. The Monolithic Frontend in the Microservices Architecture**
<http://blog.xebia.com/the-monolithic-frontend-in-the-microservices-architecture/>
- **Mauro Servienti. The secret of better UI composition**
<https://particular.net/blog/secret-of-better-ui-composition>
- **Viktor Farcic. Including Front-End Web Components Into Microservices**
<https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/>
- **Managing Frontend in the Microservices Architecture**
<http://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html>

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 restart often on another machine for availability reasons. This 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 resilience in the compute (the process restarts) 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 happen during an application upgrade. The microservice, working with the deployment system, needs to decide 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 requires the microservice to emit health information to be able to 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 tackles how to handle partial failure. It describes techniques like retries with exponential backoff or the Circuit Breaker pattern in .NET Core 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 is often overlooked, but a microservice must report its health and diagnostics. Otherwise, there is 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 is 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 is 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 may be currently 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 HealthCheck library in your microservices so they can report their state to a monitoring service to take appropriate actions.

Diagnostics, logs event streams

Logs provide information about how an application or service is running, including exceptions, warnings or simple informational messages. Usually, each log is based on text format, one line per event, although exceptions will also show the stack trace spanning to multiple lines.

In monolithic server-based applications you can simply write logs to a file on disk (a logfile) and then analyze it with any tool. Since the application execution is limited to a fixed server/VM, is not too complex to analyze the flow of events.

However, in a distributed application where multiple services are executed across many nodes in an orchestrator's cluster, being able to correlate the distributed events is a challenge.

A microservice based application should not try to store the output stream of events or logfiles by itself, 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 is running. An example of these event stream routers is [Microsoft.Diagnostic.EventFlow](#) which collects event streams from multiple sources and publish it on output systems like the simple standard output for development environment or cloud systems like [Application Insights](#), [OMS](#) (for on-premises applications) and [Azure Diagnostics](#). There are also very good third party log analysis platforms (search, alert, report and monitor), even in real time, like [Splunk](#).

Orchestrators managing health and diagnostics information

When creating a microservice-based application, you need to deal with complexity. Of course, a single microservice is simple to deal with, but tens or hundreds of types and thousands of instances of microservices is a complex problem. It's not 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 4-21. A Microservice Platform is fundamental for app's Health Management

The complex problems shown in Figure 4-21 are very hard to solve by yourself. Development teams should focus on solving business problems and building custom applications with microservices-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 reduces the complexities of building applications that use a microservices approach.

Different orchestrators might sound similar, but the capabilities on diagnostics and healthchecks offered by each of them can be different in terms of features and their state of maturity, sometimes depending on the OS platform, as explained in the next section.

Additional resources

- **12factor . Logs – Treat logs as event streams**
<https://12factor.net/logs>
- **Microsoft Diagnostic Event Flow Library**
<https://github.com/Azure/diagnostics-eventflow>
- **Microsoft. Azure Diagnostics**
<https://docs.microsoft.com/en-us/azure/azure-diagnostics>
- **Microsoft. OMS Agent**
<https://docs.microsoft.com/en-us/azure/log-analytics/log-analytics-windows-agents>
- **Microsoft. Semantic Logging Application block**
[https://msdn.microsoft.com/en-us/library/dn440729\(v=pandp.60\).aspx](https://msdn.microsoft.com/en-us/library/dn440729(v=pandp.60).aspx)
- **Splunk. Log analysis**
<http://www.splunk.com>
- **Microsoft. EventSource (for Windows ETW)**
[https://msdn.microsoft.com/en-us/library/system.diagnostics.tracing.eventsource\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/system.diagnostics.tracing.eventsource(v=vs.110).aspx)

Orchestrating 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 is composed of multiple services (like SOA), you will 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 4-22 illustrates deployment into a cluster of an application composed by multiple microservices (containers)3s3.

Composed Docker Applications in a Cluster

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

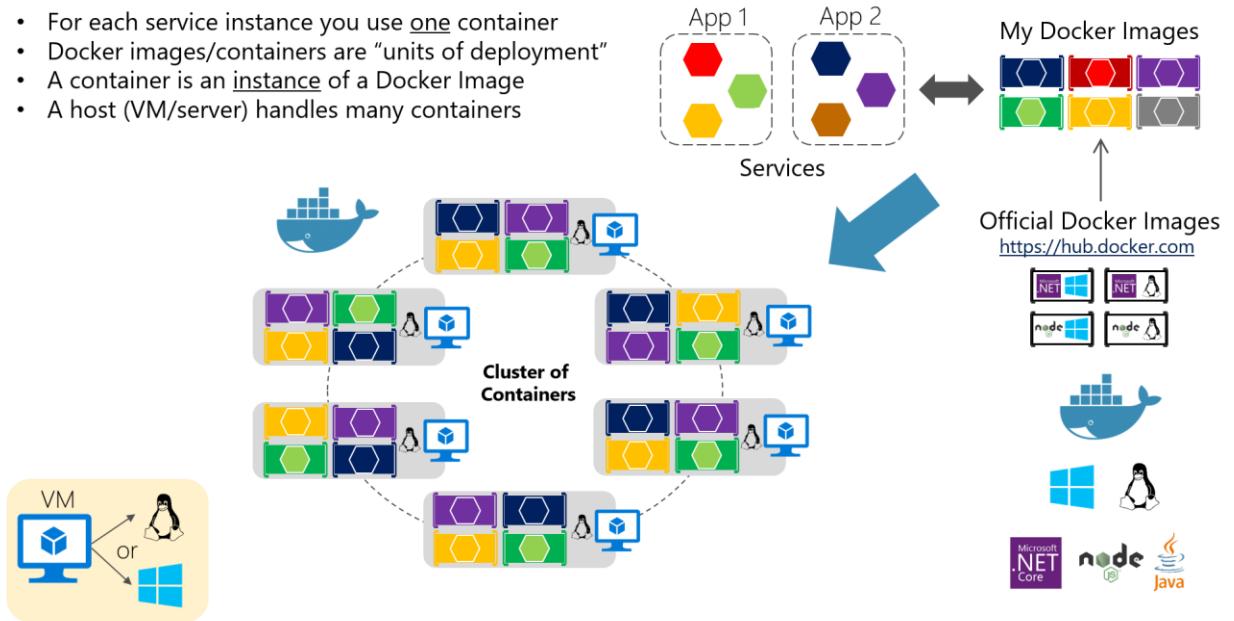


Figure 4-22. A cluster of containers

It looks like a logical approach. But how are you handling load-balancing, routing, and orchestrating these composed applications?

The Docker CLI meets the needs of managing one container 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 spin containers up, suspend them or shut them down when needed, and ideally also control how they access resources like network and data storage.

To go beyond the management of individual containers or very simple composed apps and move toward larger enterprise applications with microservices, you must turn to orchestration and clustering platforms. For Docker containers, these include Docker Swarm, Mesosphere DC/OS, and Kubernetes, which are all available as part of Microsoft Azure Container Service, or the microservices orchestrator that's part of Azure Service Fabric.

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

Clusters and Orchestrators. When you need to scale out applications across many Docker hosts like when dealing with a large microservice-based application, it is critical to have the ability to manage all those hosts as a single cluster by abstracting the complexity of the underneath platform. That is what the container clusters and orchestrators provide. Examples of orchestrators are Docker Swarm, Mesosphere DC/OS, Kubernetes and Azure Service Fabric.

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 have an efficient load balancing of containers across nodes/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 capabilities. The following list shows the most important platform and software choices you have for clusters and schedulers for containers. These clusters can be offered in public clouds like Azure.

| Software Platforms for Container Clustering, Orchestration, and Scheduling | |
|--|--|
| Docker Swarm  | <p>Docker Swarm lets you cluster and schedule Docker containers. By using Swarm, you can turn a pool of Docker hosts into a single, virtual Docker host. Clients can make API requests to Swarm the same way they do to hosts, meaning that Swarm makes it easy for apps to scale to multiple hosts.</p> <p>Docker Swarm is a product from Docker, the company.</p> <p>Docker v1.12 or later can run native and built-in Swarm Mode.</p> |
| Mesosphere DC/OS  | <p>Mesosphere Enterprise DC/OS (based on Apache Mesos) is a production ready platform for running containers and distributed applications.</p> <p>DC/OS works by abstracting a collection of the resources available in the cluster and making those resources available to components built on top of it. Marathon is usually used as a scheduler integrated with DC/OS.</p> |
| Google Kubernetes  | <p>Kubernetes is an open-source product that provides functionality from cluster infrastructure and container scheduling to orchestrating capabilities. It lets you automate deployment, scaling, and operations of application containers across clusters of hosts.</p> |

| | |
|---|--|
| | Kubernetes provides a container-centric infrastructure that groups application containers into logical units for easy management and discovery. |
| Azure Service Fabric  | <p>Service Fabric is a Microsoft microservices platform for building applications. It is an orchestrator of services and creates clusters of machines. By default, Service Fabric deploys and activates services as processes, but Service Fabric can deploy services in Docker container images. More importantly, you can mix both services in processes and services in containers together in the same application.</p> <p>The feature of Service Fabric deploying services as Docker containers is currently in preview state, as of April 2016.</p> <p>Service Fabric services can be developed in many ways, from using the Service Fabric programming models to deploying guest executables as well as containers. Service Fabric supports prescriptive application models like stateful services and Reliable Actors.</p> |

Using container-based orchestrators in Microsoft Azure

From a cloud offering perspective, several vendors are offering 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 Container Service (ACS), as explained in the next section.

Another choice is to use Microsoft Azure Service Fabric (a microservices platform), which also supports Docker support based on Linux and Windows containers. Service Fabric runs on Azure or any other cloud and also [on-premises](#).

Using Azure Container Service

A Docker cluster pools multiple Docker hosts and exposes them as a single virtual Docker host, so you can deploy multiple containers into the cluster. The cluster will handle all the complex management plumbing, like scalability, health, and so forth. Figure 4-23 represents how a Docker cluster for composed applications maps to Azure Container Service (ACS).

ACS provides a way to simplify the creation, configuration, and management of a cluster of virtual machines that are preconfigured to run containerized applications. Using an optimized configuration of popular open-source scheduling and orchestration tools, ACS 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 Container 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 choice of orchestrator tools, and Container Service handles everything else.

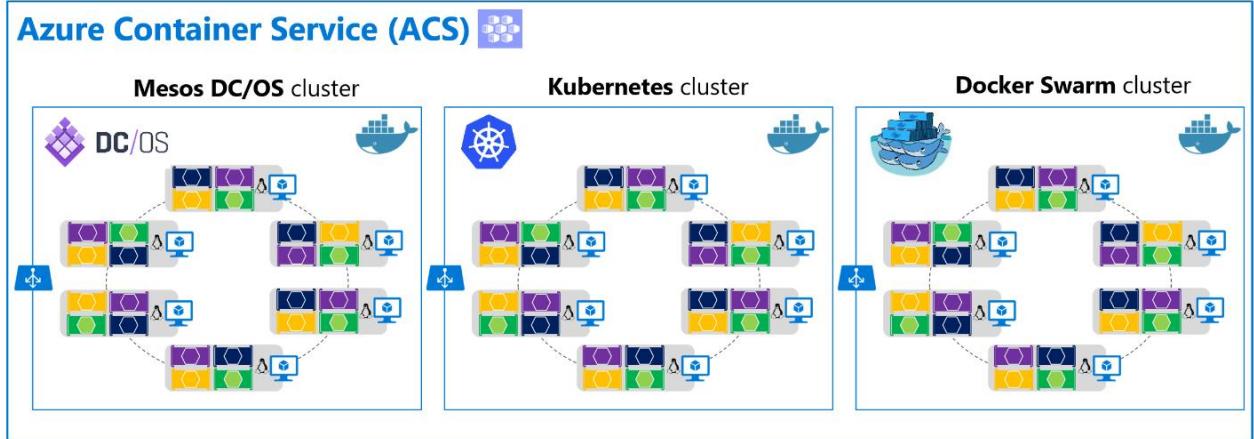


Figure 4-23. Clustering choices in Azure Container Service

ACS leverages Docker images to ensure that your application containers are fully portable. It supports your choice of open-source orchestration platforms like DC/OS (powered by Apache Mesos), Kubernetes (originally created by Google), and Docker Swarm, to ensure that these applications can be scaled to thousands or even tens of thousands of containers.

The Azure Container service enables you to take advantage of the enterprise-grade features of Azure while still maintaining application portability, including at the orchestration layers.

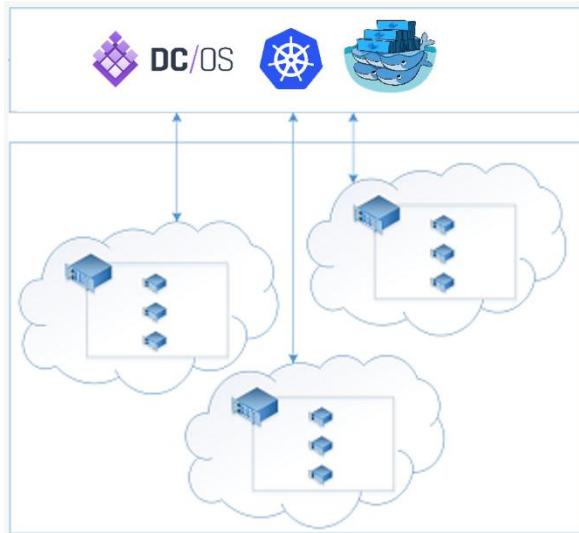


Figure 4-24. Orchestrators in ACS

As shown in figure 4-24, Azure Container Service is simply the infrastructure provided by Azure in order to deploy DC/OS, Kubernetes or Docker Swarm, but ACS does not implement any additional orchestrator. Therefore, ACS is not an orchestrator per se, only infrastructure leveraging existing OSS orchestrators for containers.

From a usage perspective, the goal of Azure Container Service is to provide a container hosting environment by using popular open-source tools and technologies. To this end, it exposes the standard API endpoints for your chosen orchestrator. By using these endpoints, you can leverage any software that can talk to those endpoints. For example, in the case of the Docker Swarm endpoint, you might choose to use the Docker command-line interface (CLI). For DC/OS, you might choose to use the DC/OS CLI.

Getting started with Azure Container Service

To begin using Azure Container Service, you deploy an Azure Container Service cluster from the Azure portal by using an Azure Resource Manager template or the [CLI](#). Available templates include [Docker Swarm](#), [Kubernetes](#), and [DC/OS](#). The provided quickstart templates can be modified to include additional or advanced Azure configuration. For more information on deploying an Azure Container Service cluster, see [Deploy an Azure Container Service cluster](#).

There are no fees for any of the software installed by default as part of ACS. All default options are implemented with open-source software.

ACS is currently available for Standard A, D, DS, G, and GS series Linux virtual machines in Azure. You are only charged for the compute instances you choose, as well as the other underlying infrastructure resources consumed such as storage and networking. There are no incremental charges for the ACS itself.

Additional resources

- **Introduction to Docker container hosting solutions with Azure Container Service**
<https://azure.microsoft.com/en-us/documentation/articles/container-service-intro/>
- **Docker Swarm overview**
<https://docs.docker.com/swarm/overview/>
- **Swarm mode overview**
<https://docs.docker.com/engine/swarm/>
- **Mesosphere DC/OS Overview**
<https://docs.mesosphere.com/1.7/overview/>
- **Kubernetes.** The official site.
<http://kubernetes.io/>

Using Azure Service Fabric

Azure Service Fabric arose from Microsoft's transition from delivering box products, which were typically monolithic in style, to delivering services. The experience of building and operating large services at scale, such as Azure SQL Database, Azure Document DB, Azure Service Bus, or Cortana's Backend, shaped Service Fabric. The platform evolved over time as more and more services adopted it. Importantly, Service Fabric had to run not only in Azure but also in standalone Windows Server deployments.

The aim of Service Fabric is to solve the hard problems of building and running a service and utilizing infrastructure resources efficiently, so that teams can solve business problems using a microservices approach.

Service Fabric provides two broad areas to help you build applications that use a microservices approach:

- A platform that provides system services to deploy, scale, upgrade, detect, and restart failed services, discover service location, manage state, and monitor health. These system services in effect enable many of the characteristics of microservices described previously.
- Programming APIs, or frameworks, to help you build applications as microservices: [reliable actors and reliable services](#). Of course, you can choose any code to build your microservice, but these APIs make the job more straightforward, and they integrate with the platform at a deeper level. This way you can get health and diagnostics information, or you can take advantage of reliable state management.

Service Fabric is agnostic with respect to how you build your service, and you can use any technology. However, it provides built-in programming APIs that make it easier to build microservices.

As shown in Figure 4-25, you can create and run microservices in Service Fabric either as simple processes or as Docker containers.

Azure Service Fabric – Types of clusters

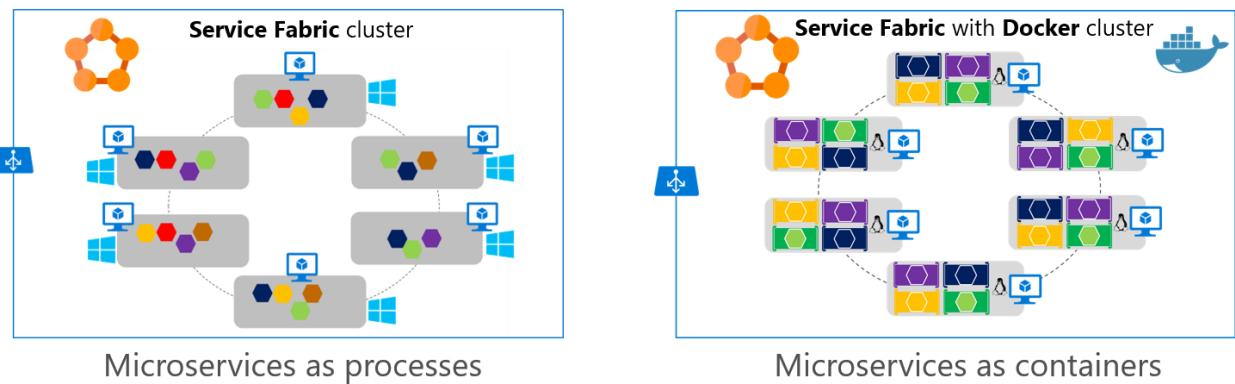


Figure 4-25. Deploying microservices as processes or as containers in Azure Service Fabric

Service Fabric clusters based on Linux and Windows hosts can run Docker Linux containers and Windows Containers.

For up-to-date information about containers support in Azure Service Fabric, see [Service Fabric and containers](#).

Service Fabric is a good example of a platform where you can define a different logical architecture (business microservices or Bounded Contexts) than the physical implementation that were introduced in the section "*Logical Architecture vs. Physical Architecture*". For example, if you implement [Stateful Reliable Services](#) in [Azure Service Fabric](#), which are introduced in the next section named "*Stateless versus stateful microservices*," you have a business microservice concept with multiple physical services.

As shown in figure 4-26, and thinking from a logical/business microservice perspective, when implementing a Service Fabric Reliable Service, you usually will need to implement two tiers of services. First, the backend stateful reliable service which handles multiple partitions, plus the frontend service or *Gateway service* in charge of routing and data aggregation across multiple partitions or stateful service instances. That Gateway service also handles the client-side communication libraries

with a retry loops accessing the backend service and retry policies used in conjunction with the Service Fabric [Reverse proxy](#).

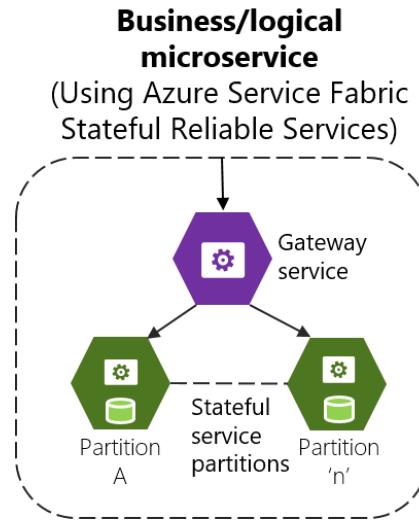


Figure 4-26. Busines microservice with several stateful/stateless services in Service Fabric

In any case, the fact here is that when using Service Fabric Stateful Reliable Services, you will also have a logical/business microservice (bounded context) usually composed by multiple physical services (each of them, gateway service or partition service, could be implemented as an ASP.NET Web API service, for instance), as shown in the image 4-9.

In Service Fabric, you can group and deploy groups of services under the concept of "[Service Fabric Application](#)" as that is the unit of packaging and deployment for the orchestrator/cluster. Therefore, the Service Fabric Application could be mapped to this autonomous "busines/logical microservice" boundary or Bounded Context, as well.

Service Fabric and containers

Going further and talking about containers in Service Fabric, you can also deploy services in container images within a Service Fabric cluster.

As shown in Figure 4-26, most of the times, there will only be one container per service.

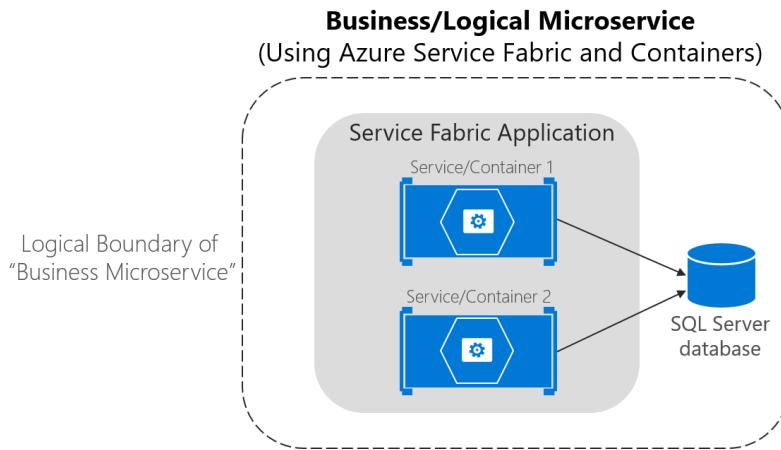


Figure 4-26. Busines microservice with several services/containers in Service Fabric

However, side-car containers (two containers that must be deployed together as part of a logical service) can also be possible in Service Fabric. The important point to highlight is that a business microservice is the logical boundary around several cohesive elements. In many cases, it might be a single service with a single data model, but in some other cases you might have physical several services, too.

It is important to note that as of April of 2017, in Service Fabric you still cannot deploy SF Reliable Stateful Services on containers but only deploy guest containers, stateless services or actor services in containers. But very importantly and as shown in the image 4-27, you can mix services in processes and services in containers in the same Service Fabric application.

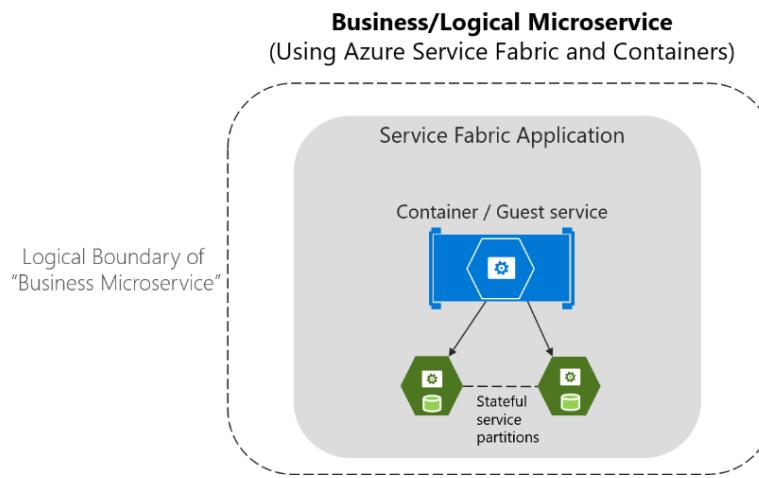


Figure 4-27. Busines microservice mapped to a Service Fabric application with containers & stateful services

Support is also different depending if using Docker containers on Linux or Windows Containers. Support for containers in Service Fabric will be increasing in upcoming releases. For up-to-date support of containers in Azure Service Fabric, check the page "[Service Fabric and Containers](#)".

Stateless versus stateful microservices

As mentioned earlier, each microservice (or logical bounded-context) must own its domain model (data and logic). In the case of stateless microservices, the databases will be external, employing relational options like SQL Server or NoSQL options like MongoDB or Azure Document DB.

But the services themselves can also be stateful, which means that the data resides within the microservice. This data might exist not just on the same server, but within the microservice's process, in memory and persisted on hard drive and replicated to other nodes. Figure 4-28 shows the different approaches.

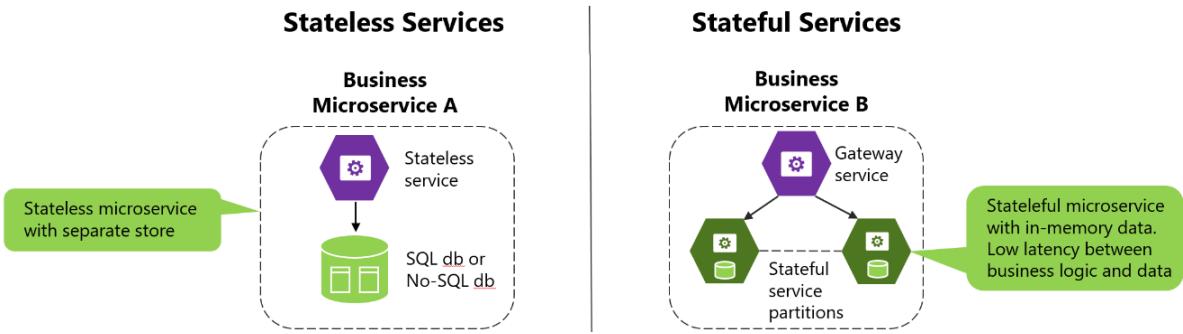


Figure 4-28. Stateless versus stateful microservices

A stateless approach is perfectly valid and is easier to implement than stateful microservices, since the approach is similar to traditional and well-known patterns. But stateless microservices impose latency between the process and data sources. They also involve more moving pieces when you are trying to improve performance via additional cache and queues. The result is that you can end up with complex architectures with too many tiers.

In contrast, [stateful microservices](#) can excel in advanced scenarios, because there is no latency between the domain logic and data. Heavy data processing, gaming back ends, databases as a service, and other low-latency scenarios all benefit from stateful services, which enable local state for faster access.

Stateless and stateful services are complementary. For instance, you can see in Figure 4-20 that a stateful service could be split into multiple partitions. To access those partitions, you might need a stateless service acting as a gateway service that knows how to address each partition based on partition keys.

Stateful services do have drawbacks. They impose a level of complexity to scale out. Functionality that would usually be implemented by external database systems must be addressed for tasks such as data replication across stateful microservices and data partitioning. However, this is one of the areas where an orchestrator like [Azure Service Fabric](#) with its [stateful reliable services](#) can help the most—by simplifying the development and lifecycle of stateful microservices using the [Reliable Services API](#) and [Reliable Actors](#).

Other microservice frameworks that allow stateful services, that support the Actor pattern, and that improve fault tolerance and latency between business logic and data are Microsoft [Orleans](#), from Microsoft Research, and Akka.NET. Both frameworks are currently improving their support for Docker, as well.

Note that Docker containers are themselves stateless. If you want to implement a stateful service, you need one of the additional, prescriptive and higher-level frameworks noted earlier. However, at the time of this writing, stateful services in Azure Service Fabric are not supported as containers, only as plain microservices. Reliable services support in containers will be available in upcoming versions of Service Fabric.

Development Process for Docker-Based Applications

Vision

Develop containerized .NET applications the way you like, either 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 tools choices: IDE or editor

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

Visual Studio with Docker Tools. If you're using Visual Studio 2015, you can install the [Visual Studio Tools for Docker](#) add-in. If you're using Visual Studio 2017, Docker Tools are already installed. In either case, the Docker tools 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 app without having to rebuild the container. This is the simplest and most powerful choice for Windows developers targeting Docker containers for Linux or Windows.

Visual Studio Code and Docker CLI If you prefer a lightweight and cross-platform editor that supports any development language, you can use Microsoft Visual Studio Code (VS Code) and the Docker CLI. This is a cross-platform development approach for Mac, Linux, and Windows.

These products provide a simple yet robust experience that streamlines the developer workflow. By installing [Docker Community Edition \(CE\)](#) tools, Docker developers can use a single Docker CLI to build apps for both Windows and Linux. Additionally, Visual Studio Code supports extensions for Docker such as intellisense for Dockerfiles and shortcut tasks to run Docker commands from the editor.

Additional resources

- [Visual Studio Tools for Docker](#)
- [Download Visual Studio Code](#)
- [Docker Community Edition \(CE\)](#) for Mac and Windows

.NET languages and frameworks for Docker containers

As introduced in earlier sections of this guide, you can use .NET Framework, .NET Core, or the OSS project Mono when developing Docker containerized .NET applications. You can develop in C#, F#, or Visual Basic targeting Linux or Windows containers, depending on which .NET framework is in use. Check the "[The .NET Language Strategy](#)" blog post for further details on .NET languages.

Development workflow for Docker apps

The application development lifecycle starts at each developer's machine, where the developer codes the app using their preferred language and tests it locally. No matter which language, framework, and platform the developer chooses, with this workflow, the developer is always developing and testing Docker containers, but doing so locally.

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

- An operating system selection (for example, a Linux distribution, Windows Nano Server, or Windows Server Core).
- Files added by the developer (app binaries, etc.).
- Configuration information (environment settings and dependencies).
- Instructions for the processes that Docker should run.

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 is not taking into account the broader DevOps workflow but still only focusing on the development work done at the dev machine. The initial steps to set up the environment are not included, since those are done only once.

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

Inner-Loop development workflow for Docker apps

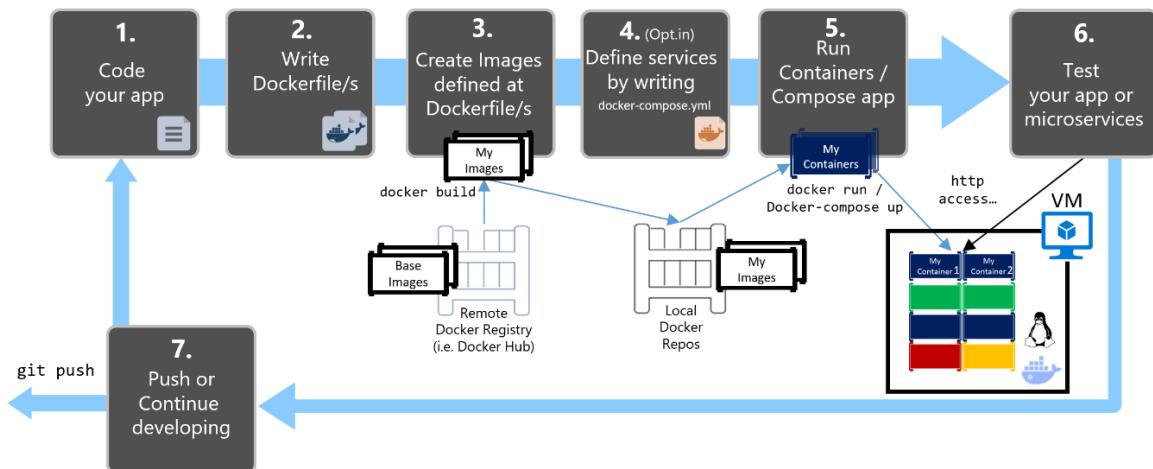


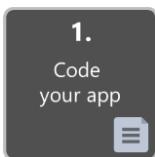
Figure 5-1. Step-by-step workflow for developing Docker containerized apps

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

When you are using an editor/CLI development approach (for example, Visual Studio Code plus Docker CLI in a macOS or Windows), you need to know every step, even in more detail than when using Visual Studio. In that case (CLI environment for macOS or Windows), refer to the eBook [Containerized Docker Application lifecycle with Microsoft Platforms and Tools](#) for these further details on a CLI environment.

When you are using Visual Studio 2015 or Visual Studio 2017, many of those steps are handled for you, which dramatically improves your productivity. This is especially true when you are using Visual Studio 2017 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 app. When you run the app 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, in the guidance that follows, we detail every step.



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

Developing a Docker app is similar to the way you develop an app without Docker. The difference is that while developing for Docker, you are deploying and testing your application or services running within Docker containers in your local environment (either a Linux VM or a Windows VM).

Set up your local environment with Visual Studio

To begin, make sure you have [Docker Community Edition \(CE\) for Windows](#) installed, as explained in the following instructions:

[Get started with Docker CE for Windows](#)

In addition, you'll need Visual Studio 2017 installed, as a preferred choice over Visual Studio 2015 with the Visual Studio Tools for Docker add-in because it has more advanced support for Docker, like support for debugging containers. Visual Studio 2017 includes the tooling for Docker if you selected the **.NET Core and Docker** workload during installation, as shown in Figure 5-2.



Figure 5-2. Selecting the **.NET Core and Docker** workload during Visual Studio 2017 setup

Additional resources

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

You can start coding your app in plain .NET (usually in .NET Core if you are planning to use containers) even before enabling Docker in your app and deploying and testing in Docker. However, it's 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, being the best example when debugging multi-container applications from Visual Studio.



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

You need a Dockerfile per custom image to be built and per container to be deployed, either performed automatically by Visual Studio when you hit F5 or manually by you if using Docker CLI (docker run and docker-compose commands). If your app contains a single custom service, you will need a single dockerfile. If your app contains multiple services (as in a microservices architecture), you need one Dockerfile per service.

The Dockerfile is placed in the root folder of your app or service. It contains the commands that tell Docker how to set up and run your app 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 2017, there's an option named **Enable Container (Docker) Support**, as shown in Figure 5-3.

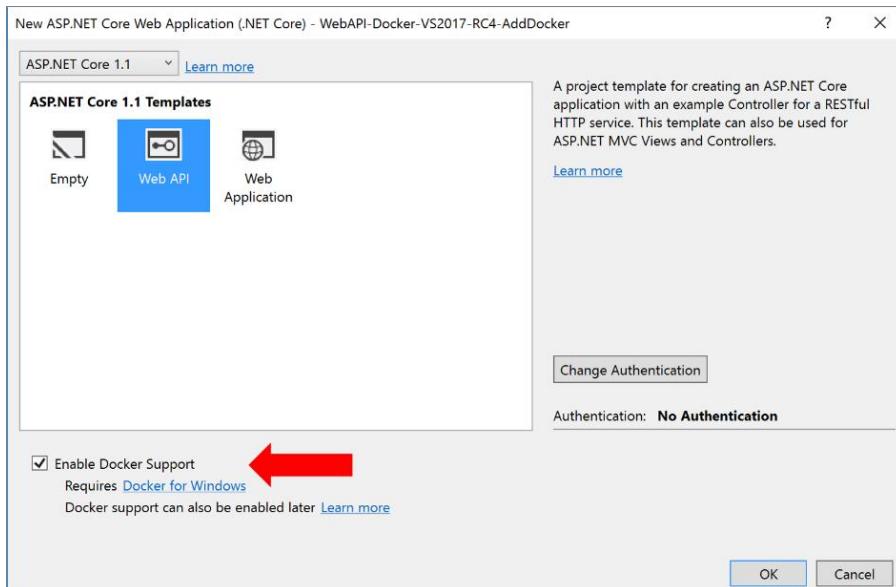


Figure 5-3. Enabling Docker Support when creating a new project in Visual Studio 2017

You can also enable Docker support on a new or existing project by right-clicking your project file in Visual Studio and selecting the option **Add-Docker Project Support**, as shown in figure 5-4.

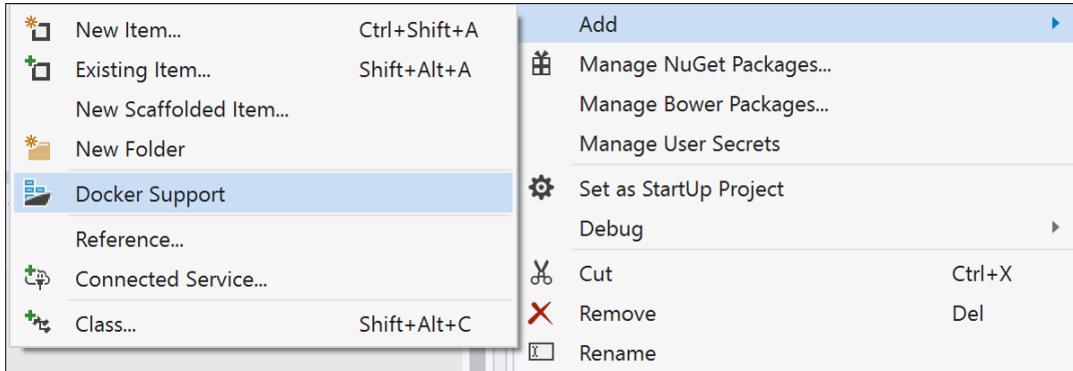


Figure 5-4. Enabling Docker support in an existing Visual Studio 2017 project

This action on a project (like an ASP.NET Web app or Web API service) adds a Dockerfile to the project with the required configuration, plus it will add a docker-compose.yml file for the whole solution. In the following sections, we describe the information that goes into each of those files. Visual Studio can do this work for you, but it's useful to understand what goes into a Dockerfile.

Option A: Creating a project using an existing official .NET Docker image

You usually build a custom image for your container on top of a base image you can get from an official repository at the [Docker Hub](#) registry. That is precisely what happens under the covers when enabling Docker support from Visual Studio. Your Dockerfile will use an existing aspnetcore image.

Earlier we explained which Docker images and repos you can use, depending on the framework and OS you've chosen. For instance, if you want to use ASP.NET Core and Linux, the image to use is `microsoft/aspnetcore:1.1`. Therefore, you just need to specify what base Docker image you'll be using for your container by specifying that in your Dockerfile, by adding `FROM microsoft/aspnetcore:1.1` to your Dockerfile. This will be automatically performed by Visual Studio, but if you were to update the version, you will do it that way.

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

Figure 5-5 shows a sample Dockerfile for an ASP.NET Core container.

```
FROM microsoft/aspnetcore:1.1
ARG source
WORKDIR /app
EXPOSE 80
COPY ${source:-obj/Docker/publish} .
ENTRYPOINT ["dotnet", "MySingleContainerWebApp.dll"]
```

Figure 5-5. Sample Dockerfile for a ASP.NET Core container (Image for .NET Core on Linux)

In this case, the container is based on version 1.1 of the official ASP.NET Core Docker image for Linux; this is the setting `FROM microsoft/aspnetcore:1.1`. (For further details about this base image, see the [ASP.NET Core Docker Image page](#) and the [.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 settings).

You can specify additional configuration settings in the Dockerfile, depending on the language and framework you are using. For instance, the ENTRYPOINT line with ["dotnet", "MySingleContainerWebApp.dll"] tells Docker to run a .NET Core app. If you are using the SDK and the .NET CLI (dotnet CLI) to build and run the .NET app, 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 .NET Core Applications**
<https://docs.microsoft.com/en-us/dotnet/articles/core/docker/building-net-docker-images>
- **Build your own image.** In the official Docker documentation.
<https://docs.docker.com/engine/tutorials/dockerimages/>

Using multi-platform 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. For example, the [microsoft/dotnet](#) repository available in the Docker Hub registry provides support for Linux and Windows Nano Server by using the same repo name with different tags, as shown in the following examples.

| | |
|---|---|
| microsoft/dotnet:1.1-runtime | .NET Core 1.1 runtime-only on Linux Debian |
| microsoft/dotnet:1.1-runtime-nanoserver | .NET Core 1.1 runtime-only on Windows Nano Server |

In the future, it will be possible to use the same repo name and tag targeting multiple operating systems. That 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.

Option B: 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

- **Create a base image** (in the Docker documentation)
<https://docs.docker.com/engine/userguide/eng-image/baseimages/>



Step 3. Create your custom Docker images and embed your app or service in it

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

You, as 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 5-6.

```
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
121d7ee6c20: waiting
eb57cf4f29ee: waiting
b2c5ae2d325b: waiting
```

Figure 5-6. Creating a custom Docker image

Optionally, instead of directly running `docker build` from the project's 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 representing a specific version. You can repeat this step for each custom image you need to create for your composed Docker application of several containers.

When an application is made by multiple containers (multi-container app), 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 5-7.

| REPOSITORY | TAG | IMAGE ID | CREATED | SIZE |
|--|--------|--------------|---------------|----------|
| cesardl/netcore-webapi-microservice-docker | first | 384c4ac1809b | 4 minutes ago | 579.8 MB |
| microsoft/dotnet | latest | 49aa5daa850 | 30 hours ago | 548.6 MB |
| ubuntu | latest | cf62323fa025 | 5 days ago | 125 MB |
| hello-world | latest | c54a2cc56ccb | 12 days ago | 1.848 kB |

Figure 5-7. Viewing existing images using the `docker images` command

Creating Docker Images with Visual Studio

When you are using 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 and 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 app

The [docker-compose.yml](#) file lets you define a set of related services to be deployed as a composed application with deployment commands.

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 shown in Figure 5-8.

```
version: '2'

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

  catalog.api:
    image: eshop/catalog.api
    environment:
      ConnectionString=Server=catalogdata;Port=5432;Database=postgres;...
    ports:
      - "81:80"
    depends_on:
      - postgres.data

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

  sql.data:
    image: mssql-server-linux:latest
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"

  postgres.data:
    image: postgres:latest
    environment:
```

```
POSTGRES_PASSWORD: tempPwd
```

Figure 5-8. Example docker-compose.yml file for a multi-container based app

Note that 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 always applies, plus configuration information that might depend on the deployment environment, like the connection string. In later sections, you will learn how you can 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 five services: the webmvc service (a web app); two microservices (ordering.api and basket.api); and two data source containers, sql.data based on SQL Server for Linux running as a container, and postgres.data with a Redis cache service. 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, on the webmvc container definition at the .yml file:

- Uses the pre-built eshop/web:latest image. Although you could also command the image to be built as part of the docker-compose execution with an additional configuration based on a section “build:” within 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 service to the basket 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 2017

When you add Docker solution support to a service project in a Visual Studio solution, Visual Studio adds a Dockerfile to your project, and it adds a service section (project) in your solution with the docker-compose.yml files. It is an easy way to start composing your multiple-container solution. You can then open the docker-compose.yml files and update them with additional features.

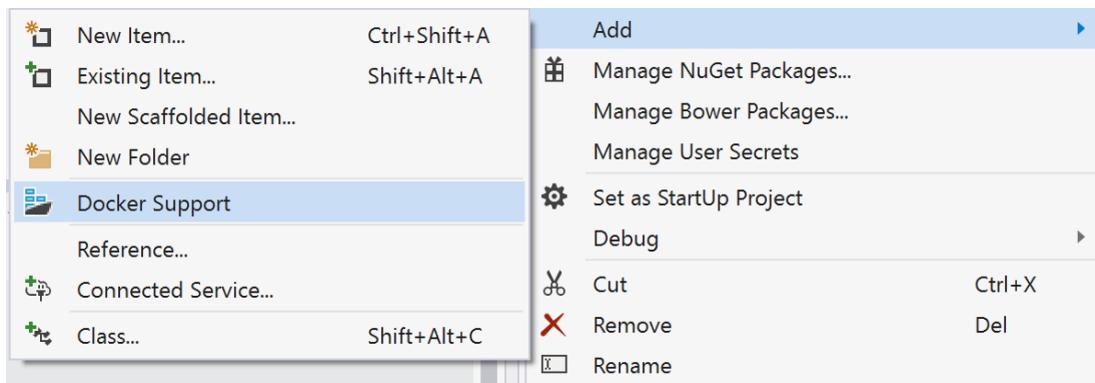


Figure 5-9. Adding Docker support in Visual Studio 2017 by right-clicking on an ASP.NET Core project

This action not only adds the Dockerfile to your project, but it adds the configuration information to several global docker-compose.yml files set at the solution level.

After you add Docker support to your solution in Visual Studio, you will also see a new node (docker-compose.dcproj project file) in the Solution Explorer that contains the added docker-compose.yml files, as shown in Figure 5-10.

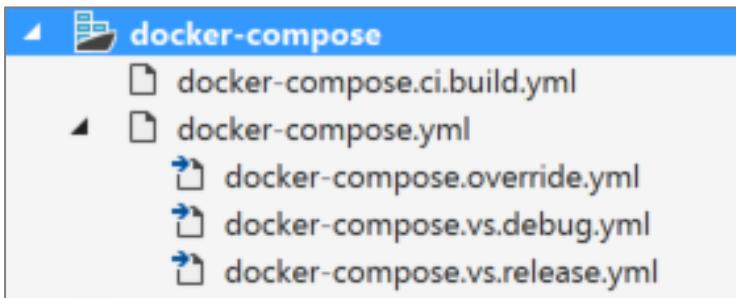


Figure 5-10. The docker-compose tree node added in Visual Studio 2017 Solution Explorer

You could deploy a multi-container application by using 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 (dev versus. production) and execution type (release versus debug). This capability will be explained in later sections.



Step 5. Build and run your Docker app

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

Option A: Running a single-container with Docker CLI

You can run a Docker container using the "docker run" command, as in the following example:

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

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

Figure 5-11. Running a Docker container using the "docker run" command

In this case, it 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.

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 5-12.

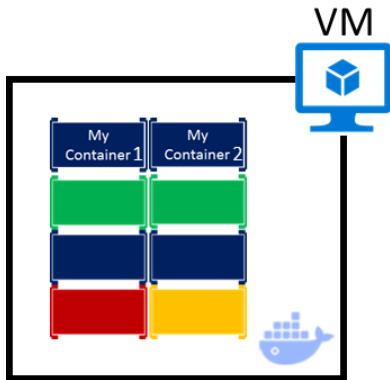


Figure 5-12. VM with Docker containers deployed

Running a multi-container application with Docker CLI

For this scenario, you can run 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 5-13 shows the results when running the command from your main project 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 5-13. 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 illustrated in the previous VM representation in Figure 5-12.

Running and debugging a multi-container application with Visual Studio

Running a multi-container application using Visual Studio 2017 cannot get simpler. You can not only run the multi-container application, but you're able to debug all its containers directly from Visual Studio by setting regular breakpoints.

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 spin up one container per project that has Docker solution support enabled, and perform all the internal steps for you (dotnet publish, docker build to build the Docker images, etc.).

The important point here is that, as shown in Figure 5-14, in Visual Studio 2017 there is an additional **Docker** command under the F5 key. This option lets you run or debug a multiple 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.

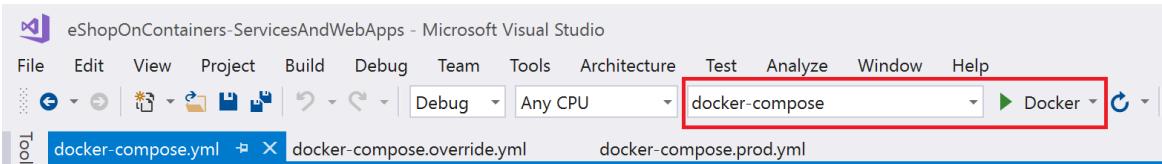


Figure 5-14. Running multi-container apps in Visual Studio 2017

Additional resources

- **Deploy an ASP.NET container to a remote Docker host**

<https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-docker-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 if you are targeting Docker clusters and orchestrators like Docker Swarm, Mesosphere DC/OS, or Kubernetes. If you are using a cluster like [Docker Swarm mode](#) (available in Docker CE for Windows and Mac since version 1.12), you need to deploy and test with additional commands like “[docker service create](#)” for single services. If you are deploying an app composed of several containers, you use “[docker compose bundle](#)” and “[docker deploy myBundleFile](#)” in order to deploy the composed app as a *stack*. For more information, see the article [Introducing Experimental Distributed Application Bundles in the Docker documentation](#).

For [DC/OS](#) and [Kubernetes](#) you would use different deployment commands and scripts as well.

6.
Test
your app or
microservices

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

This step will vary depending on what your app is doing. In a simple .NET Core Web app 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 5-15. (If the configuration in the Dockerfile maps the container to a port on the host that's anything other than 80, include the host port in the URL.)



Figure 5-15. 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.

Note that 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's how it was deployed with the "docker run" command, as explained in a previous step.

You can also test the app using curl from the terminal, as shown in figure 5-16. In a Docker installation on Windows, the default Docker Host IP is always 10.0.75.1 in addition to your real machine's IP.

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

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

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

Figure 5-16. Example of testing your Docker application locally using curl

Testing and debugging containers with Visual Studio 2017

When running and debugging the containers with Visual Studio 2017, you can debug the .NET application in much the same way as you would when running without containers.

Additional resources

- **Debugging apps in a local Docker container**
<https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-docker-edit-and-refresh/>

Testing and debugging without Visual Studio

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

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, because 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 the image 5-17.

VS development workflow for Docker apps

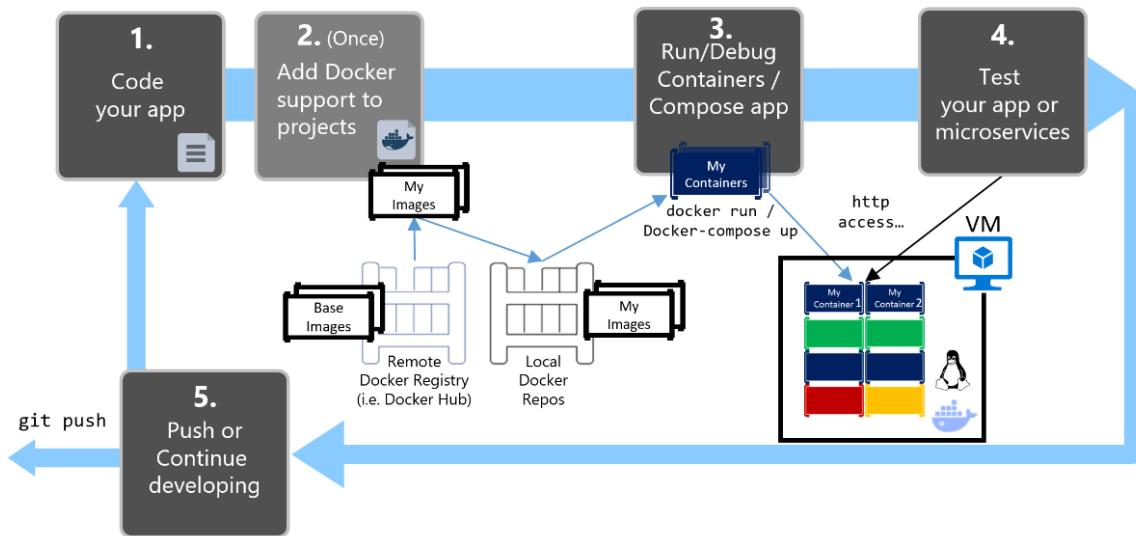


Figure 5-17. Simplified workflow when developing with Visual Studio

Moreover, you need to perform step 2 (Add Docker support to your projects) just once. Therefore, the workflow remains similar to your usual development tasks when using .NET for any other development. You need to know what's going on under the covers (the images build process, what base images you are 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.

Additional resources

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

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 Figure 5-18.

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

Figure 5-18. Example of running PowerShell commands in the Dockerfile

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 4.6, or any other Windows software. For example, following command in a Dockerfile sets up ASP.NET 4.5:

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

Additional resources

- **Example of powershell commands to run from dockerfiles to include Windows features**
<https://github.com/Microsoft/aspnet-docker/blob/master/4.6.2/Dockerfile>

Deploying Single-Container-Based .NET Core Web Applications on Linux or Windows Nano Server Hosts

Vision

You can use Docker containers for monolithic deployment of simpler web applications. This improves continuous integration/continuous deployment pipelines and helps achieve deployment-to-production success. No more "it works in my machine, why doesn't work in production?"

A microservices-based architecture has many benefits, but those benefits come at a cost of increased complexity. In some cases, the costs outweigh the benefits, and you will be better served with a monolithic deployment application running in a single container or in just a few containers.

An application might not be easily decomposable into well-separated microservices. You've learned that these should be partitioned by function: microservices should work independently of each other to provide a more resilient application. If you can't deliver feature slices of the application, separating it only adds complexity.

An application might not yet need to scale features independently. Let's suppose that early in the life of our reference application eShopOnContainers, the traffic did not justify separating features into different microservices. Traffic was small enough that adding resources to one service typically meant adding resources to all services. The additional work to separate the application into discrete services provided minimal benefit.

Also, early in the development of an application you might not have a clear idea where the natural functional boundaries are. As you develop a minimum viable product, the natural separation might not yet have emerged.

Some of these conditions might be temporary. You may start by creating a monolithic application, and later separate some features to be developed and deployed as microservices. Other conditions

might be essential to the application's problem space, meaning that the application might never be broken into multiple microservices.

Separating an application into many discrete processes introduces overhead. There is more complexity in separating features into different processes. The communication protocols become more complex. Instead of method calls, you must use asynchronous communications between services. As you move to a microservices architecture, you need to add many of the building blocks implemented in the microservices version of the *eShopOnContainers* application: event bus handling, message resiliency and retries, eventual consistency, and more.

A very much simplified version of *eShopOnContainers* (named [eShopWeb](#) and included in the same GitHub repo) runs as a monolithic MVC application, and as just described, there are advantages offered by that design choice. You can download the source for this application from GitHub and run it locally. Even this monolithic application benefits from being deployed in a container environment.

For one, the containerized deployment means that every instance of the application runs in the same environment. This includes the developer environment where early testing and development take place. The development team can run the application in a containerized environment that matches the production environment.

In addition, containerized applications scale out at lower cost. As you saw earlier, the container environment enables greater resource sharing than traditional VM environments.

Finally, containerizing the application forces a separation between the business logic and the storage server. As the application scales out, the multiple containers will all rely on a single physical storage medium. This would typically be a high-availability server running a SQL Server database.

Application tour

The [eShopWeb](#) application represents some of the *eShopOnContainers* application running as a monolithic application—an ASP.NET Core MVC based application running on .NET Core. It mainly provides the catalog browsing capabilities that we described in earlier sections.

The application uses a SQL Server database for the catalog storage. In container-based deployments, this monolithic application can access the same data store as the microservices-based application. The app is configured to run SQL Server in a container alongside the monolithic application. In a production environment, SQL Server would run on a high availability machine, outside of the Docker host. For convenience in a dev or test environment, we recommend running SQL Server in its own container.

The initial feature set only enables browsing the catalog. Updates would enable the full feature set of the containerized application. A more advanced monolithic web application architecture is described in the [ASP.NET Web Application architecture practices](#) eBook and related [eShopOnWeb sample app](#), although in that case it is not running on Docker containers because that scenario focuses on plain web development with ASP.NET Core.

However, the simplified version available at [eShopOnContainers \(eShopWeb\)](#) runs in a Docker container.

Docker support

The eShopOnWeb project runs on .NET Core. Therefore, it can run in either Linux-based or Windows-based containers. Note that for Docker deployment, you want to use the same host type for SQL Server. Linux-based containers allow a smaller footprint and are preferred.

Visual Studio provides a project template that adds support for Docker to a solution. You right-click the project, click **Add** followed by **Docker Support**. The template adds a Dockerfile to your project, and a new **docker-compose** project that provides a starter docker-compose.yml file. This step has already been done in the eShopOnWeb project downloaded from GitHub. You'll see that the solution contains the eShopOnWeb project and the docker-compose project as shown in Figure 6.1.

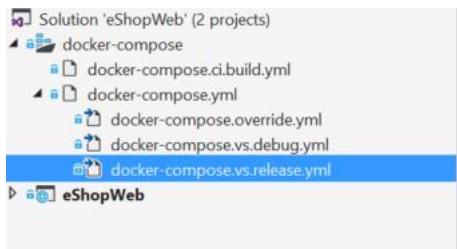


Figure 6-1. docker-compsoe project in a single container web app

These files are standard docker-compose files, consistent with any Docker project. You can use them with Visual Studio or from the command line. This application runs on .NET Core and uses Linux containers, so you can also code, build, and run on a Mac or on a Linux machine.

The docker-compose.yml file contains information about what images to build and what containers to launch. The templates specify how to build the eshopweb image and launch the app's containers. You need to add the dependency on SQL Server by including an image for it (for example, mssql-server-linux), and a service for the sql.data image for Docker to build and launch that container. These settings are shown in the following example.

```
version: '2'

services:
  eshopweb:
    image: eshop/web
    build:
      context: ./eShopWeb
      dockerfile: Dockerfile
    depends_on:
      - sql.data
  sql.data:
    image: microsoft/mssql-server-linux
```

The depends_on directive tells Docker that the eShopWeb image depends on the sql.data image. Lines below that are the instructions to build an image tagged sql.data using the microsoft/mssql-server-linux image.

The **docker-compose** project displays the other docker-compose files under the main docker-compose.yml node to provide a visual indication that these files are related. The docker-compose-

override.yml file contains settings for both services, such as connection strings and other application settings.

The following example shows the .docker-compose.vs.debug.yml file, which contains settings used for debugging in Visual Studio. In that file, the eshopweb image has the dev tag appended to it. That helps separate debug from release images so that you don't accidentally deploy the debug information to a production environment:

```
version: '2'

services:
  eshopweb:
    image: eshop/web:dev
    build:
      args:
        source: ${DOCKER_BUILD_SOURCE}
    environment:
      - DOTNET_USE_POLLING_FILE_WATCHER=1
    volumes:
      - ./eShopWeb:/app
      - ~/.nuget/packages:/root/.nuget/packages:ro
      - ~/clrdbg:/clrdbg:ro
    entrypoint: tail -f /dev/null
    labels:
      - "com.microsoft.visualstudio.targetoperatingsystem=linux"
```

The last file added is docker-compose.ci.build.yml. This would be used from the command line to build the project from a CI server. This compose file starts a Docker container that builds the images needed for your application. The following example shows the contents of the docker-compose.ci.build.yml file.

```
version: '2'

services:
  ci-build:
    image: microsoft/aspnetcore-build:1.0-1.1
    volumes:
      - .:/src
    working_dir: /src
    command: /bin/bash -c "dotnet restore ./eShopWeb.sln && dotnet publish ./eShopWeb.sln -c Release -o ./obj/Docker/publish"
```

Notice that the image is an ASP.NET Core build image. That image includes the SDK and build tools to build your application and create the required images. Running the **docker-compose** project using this file starts the build container from the image, then builds your application's image in that container. You specify that docker compose file as part of the command line to build your application in a Docker container, then launch it.

In Visual Studio, you can run your application in Docker containers by selecting the **docker-compose** project as the startup project, and then pressing Ctrl+F5 (F5 to debug), as you can with any other application. When you start the docker-compose project, Visual Studio runs **docker-compose** using the docker-compose.yml file, the docker-compose.override.yml file, and one of the docker-compose.vs.* files. Once the application has started, Visual Studio launches the browser for you.

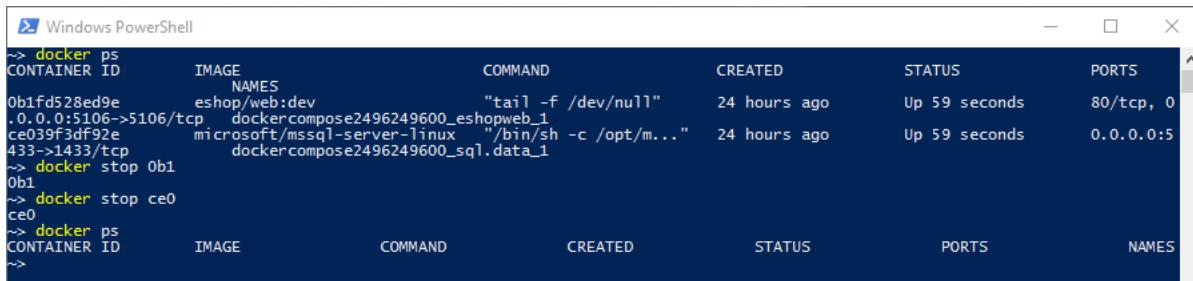
If you launch the application in the debugger, Visual Studio will attach to the running application in Docker.

Troubleshooting

This section describes a few issues that might arise when you run containers locally and suggests some fixes.

Stopping Docker containers

After you launch the containerized application, the containers continue to run, even after you've stopped debugging. You can run the `docker ps` command from the command line to see which containers are running. The `docker stop` command stops a running container, as shown in Figure 6-2.



```
Windows PowerShell
PS C:\> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
0b1fd528ed9e      eshop/web:dev       "tail -f /dev/null"   24 hours ago       Up 59 seconds      80/tcp, 0
ce039f3df92e      dockercompose2496249600_eshopweb_1   "/bin/sh -c /opt/m..."   24 hours ago       Up 59 seconds      0.0.0.0:5106->5106/tcp
433->1433/tcp
PS C:\> docker stop 0b1
0b1
PS C:\> docker stop ce0
ce0
PS C:\> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
PS C:\>
```

Figure 6-2. Listing and stopping containers with “`docker ps`” and “`docker stop`” CLI commands

You might need to stop running processes when you switch between different configurations. Otherwise, the container that is running the web application is using the port for your application (5106 in this example).

Adding Docker to your projects

The wizard that adds Docker support communicates with the running Docker process. The wizard will not run correctly if Docker is not running when you start the wizard. In addition, the wizard examines your current container choice to add the correct Docker support. If you want to add support for Windows Containers, run the wizard while you have Docker running with Windows containers configured. If you want to add support for Linux containers, run the wizard while you have Docker running with Linux containers configured.

Migrating Legacy Monolithic .NET Framework Applications to Windows Containers

Vision

Windows Containers can also be used as a way to improve dev/test environments and deployment to production of applications based on legacy technologies like WebForms or any other legacy technology available in the full .NET Framework. This is what is called a "lift and shift scenario".

Earlier sections of this guide have championed a microservices architecture where business applications are distributed among different containers, each running a small, focused service. That goal has many benefits. In new development, it's strongly recommended. Enterprise-critical applications will also benefit enough to justify the cost of a re-architecture and re-implementation.

But not every application will benefit enough to justify the cost. That doesn't mean those applications can't be used in container scenarios.

In this section, we'll explore an application for eShopOnContainers, shown in Figure 7-1. This application would be used by members of the *eShopOnContainers* enterprise to view and edit the product catalog.

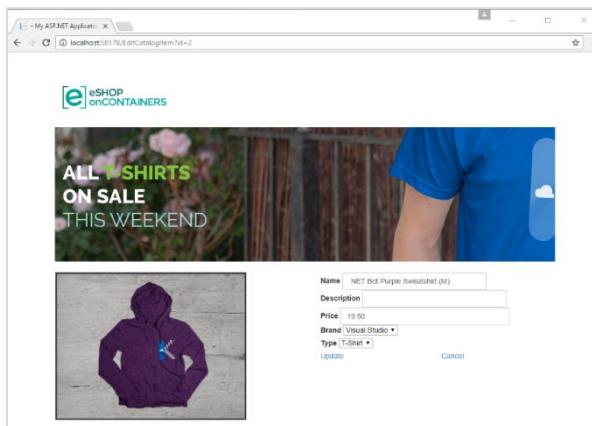


Figure 7-1. Web app implemented with ASP.NET Web Forms (legacy technology) on a Windows Container

This is a Web Forms application used to browse and modify the catalog entries. The Web Forms dependency means this application won't run on .NET Core unless it is rewritten without Web Forms and instead uses ASP.NET Core MVC. You'll see how you can run applications like these in containers without changes. You'll also see how you can make minimal changes to work in a hybrid mode where some functionality has been moved into a separate microservice, but most functionality remains in the monolithic application.

Benefits

The Catalog.WebForms application is available in the eShopContainers GitHub repository (<https://github.com/dotnet/eShopOnContainers>). This application is a standalone web application accessing a high availability data store. Even so, there are advantages to running the application in a container. You create an image for the application. From that point forward, every deployment runs in the same environment. Every container uses the same OS version, has the same version of dependencies installed, uses the same framework, and is built using the same process. You can see the application loaded in Visual Studio 2017 in Figure 7-2.

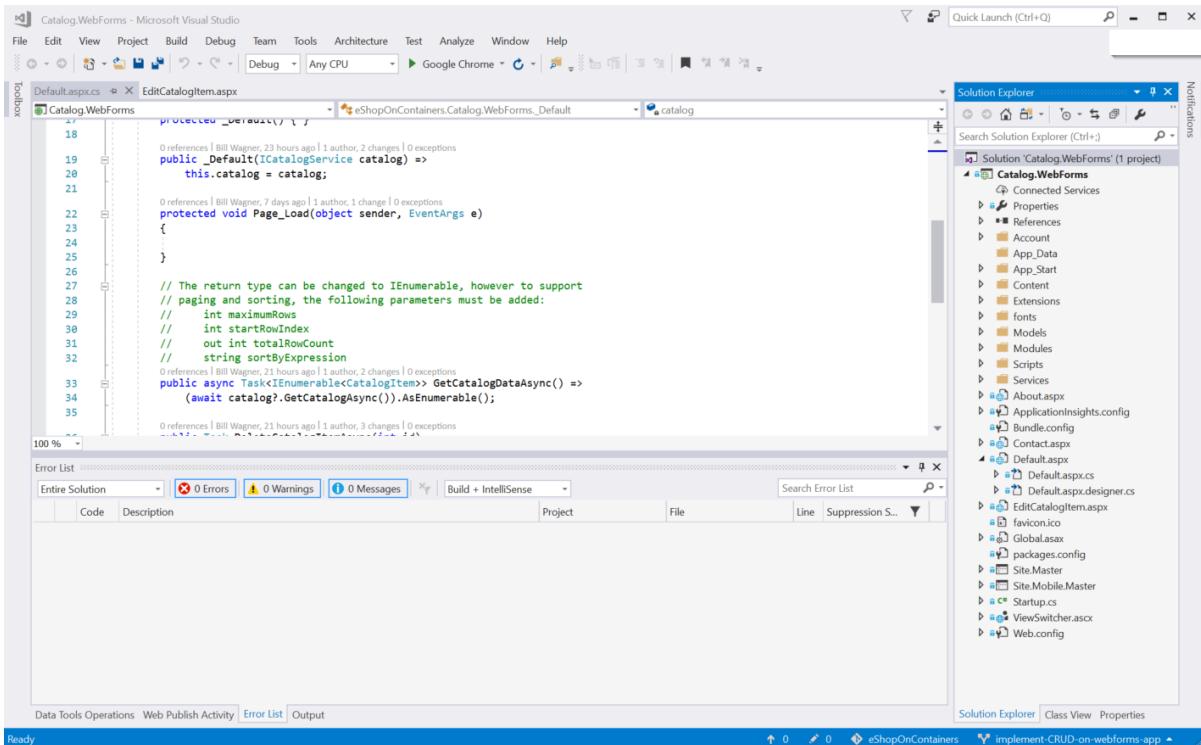


Figure 7-2. Catalog management Web Forms app in Visual Studio 2017

In addition, developers can all run the application in this consistent environment. Issues that only appear with certain versions will appear immediately for developers rather than surfacing in a staging or production environment. Differences between the development environments among the development team matter less once applications run in containers.

Finally, containerized applications have a flatter scale-out curve. You've learned how containerized apps enable more containers in a VM or more containers in a physical machine. This translates to higher density and fewer required resources.

For all of these reasons, consider running legacy monolithic apps in a Docker container using a “lift-and-shift” operation. The phrase “lift and shift” describes the scope of the task: you *lift* the entire application from a physical or virtual machine, and *shift* it into a container. In ideal situations, you don’t need to make any changes to the application code to run it in a container.

Possible migration paths

As a monolithic application, the `catalog.WebForms` app is one web application containing all the code, including the data access libraries. The database runs on a separate high availability machine. That configuration is simulated in the sample code by using a mock catalog service: you can run the `catalog.WebForms` application against that fake data to simulate a pure lift-and-shift scenario. This demonstrates the simplest migration path, where you move existing assets to run in a container without any code changes at all. This path is appropriate for applications that are complete and that have minimal interaction with functionality that you are moving to microservices.

However, the *eShopOnContainers* website is already accessing the data storage using microservices for different scenarios. Some small additional changes can be made to the catalog editor to leverage the catalog microservice instead of accessing the catalog data storage directly.

These changes demonstrate the continuum for your own applications. You can do anything from moving an existing application without change into containers, to making small changes that enable existing applications to access some of the new microservices, to completely rewriting an application to fully participate in a new microservice-based architecture. The right path depends on both the cost of the migration and the benefits from any migration.

Application tour

You can load the `Catalog.WebForms` solution and run the application as a standalone app. In this configuration, instead of a persistent storage database, the application uses a fake service to return data. The application uses AutoFac (<https://autofac.org/>) as an inversion of control (IOC) container. Using Dependency Injection (DI), you can configure the application to use the fake data or the live catalog data service. (We’ll explain more about DI shortly.) The startup code reads a `useFake` setting from the `web.config` files, and configures the AutoFac container to inject either the fake data service or the live catalog service. If you run the application with `useFake` set to false in the `web.config` file, you see the Web Forms application displaying the catalog data.

Most of the techniques used in this application should be very familiar to anyone who has used Web Forms. However, the catalog microservice introduces two techniques that might be unfamiliar: Dependency Injection (DI), which was mentioned earlier, and working with asynchronous data stores in Web Forms.

DI inverts the typical object-oriented strategy of writing classes that allocate all needed resources. Instead, classes request their dependencies from a service container. The advantage of DI is that you can replace external services with fakes (mocks) to support testing or other environments.

The DI container uses `web.config appSettings` configuration to control whether to use the fake catalog data or the live data from the running service. The application registers an `HttpModule` object that builds the container and registers a pre-request handler to inject dependencies. You can see that code in the `Modules/AutoFacHttpModule.cs` file, which looks like the following example:

```

private static.IContainer CreateContainer()
{
    // Configure AutoFac:
    // Register Containers:
    var settings = WebConfigurationManager.AppSettings;
    var useFake = settings["usefake"];
    bool fake = useFake == "true";
    var builder = new ContainerBuilder();
    if (fake)
    {
        builder.RegisterType<CatalogMockService>()
            .As<ICatalogService>();
    }
    else
    {
        builder.RegisterType<CatalogService>()
            .As<ICatalogService>();

        builder.RegisterType<RequestProvider>()
            .As< IRequestProvider>();
    }
    var container = builder.Build();
    return container;
}

private void InjectDependencies()
{
    if (HttpContext.Current.CurrentHandler is Page page)
    {
        // Get the code-behind class that we may have written
        var pageType = page.GetType().BaseType;

        // Determine if there is a constructor to inject, and grab it
        var ctor = (from c in pageType.GetConstructors()
                    where c.GetParameters().Length > 0
                    select c).FirstOrDefault();

        if (ctor != null)
        {
            // Resolve the parameters for the constructor
            var args = (from parm in ctor.GetParameters()
                        select Container.Resolve(parm.ParameterType))
                .ToArray();

            // Execute the constructor method with the arguments resolved
            ctor.Invoke(page, args);
        }

        // Use the Autofac method to inject any
        // properties that can be filled by Autofac
        Container.InjectProperties(page);
    }
}

```

The app's pages (Default.aspx.cs and EditPage.aspx.cs) define constructors that take these dependencies. Note that the default constructor is still present and accessible. The infrastructure needs this.

```
protected _Default() { }

public _Default(ICatalogService catalog) =>
    this.catalog = catalog;
```

The catalog APIs are all asynchronous methods. Web Forms now supports these for all data controls. The Catalog.WebForms application uses model binding for the list and edit pages; controls on the pages define SelectMethod, UpdateMethod, InsertMethod, and DeleteMethod properties that specify Task-returning asynchronous operations. Web Forms controls understand when the methods bound to a control are asynchronous. The only restriction you encounter when using asynchronous select methods is that you cannot support paging: the paging signature requires an out parameter. Asynchronous methods cannot have out parameters. This same technique is used on other pages that require data from the catalog service.

The default configuration for the catalog Web Forms application uses a mock implementation of the catalog.api service. This mock uses a hard-coded dataset for its data, which simplifies some tasks by removing the dependency on the catalog.api service in development environments.

Lifting and shifting

Visual Studio provides great support for containerizing an application. You right-click the project node and then select **Add** and **Docker Support**. The Docker project template adds a new project to the solution called **docker-compose**. The project contains the Docker assets that compose (or build) the images you need, and starts running the necessary containers, as shown in Figure 7-3.

In the simplest lift-and-shift scenarios, the application will be the single service that you use for the Web Forms application. The template also changes your startup project to point to the **docker-compose** project. Pressing Ctrl+F5 or F5 now creates the Docker image and launches the Docker container.

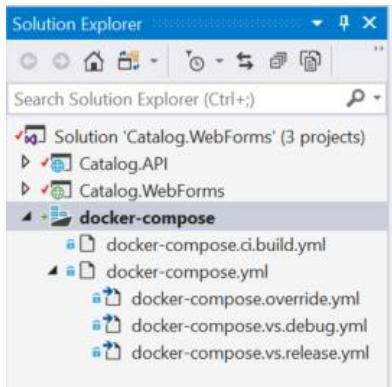


Figure 7-3. The docker-compose project in the Web Forms solution

Before you run the solution, you must make sure that you configure Docker to use Windows containers. To do that, you right-click the Docker taskbar icon in Windows and select **Switch to Windows containers**, as shown in Figure 7-4.

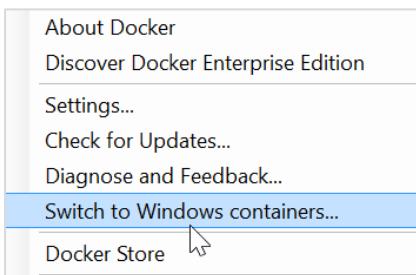


Figure 7-4. Switching to Windows Containers from Docker taskbar icon in Windows

If the menu item says **Switch to Linux containers**, you're already running Docker with Windows containers.

Running the solution restarts the Docker host. When you build, you build the application and the Docker image for the Web Forms project. The first time you do this, it takes considerable time. That's because the build process pulls down the base Windows Server image and the additional image for ASP.NET. Subsequent build and run cycles will be much faster.

Let's take a deeper look at the files added by the Docker project template. It created several files for you. Visual Studio uses these files to create the Docker image and launch a container. You can use the same files from the CLI to run Docker commands manually.

The following Dockerfile example shows the basic settings for building a Docker image based on the Windows ASP.NET image that runs an ASP.NET site.

```
FROM microsoft/aspnet
ARG source
WORKDIR /inetpub/wwwroot
COPY ${source:-obj/Docker/publish} .
```

This Dockerfile will look very similar to those created for running an ASP.NET Core application in Linux containers. However, there are a few important differences. The most important difference here is that the base image is `microsoft/aspnet`, which is the current Windows Server image that includes the .NET Framework. Other differences are that the directories copied from your source directory are different.

The other files in the `docker-compose` project are the Docker assets needed to build and configure the containers. Visual Studio puts the various `docker-compose.yml` files under one node to highlight how they are used. The base `docker-compose` file contains the directives that are common to all configurations. The `docker-compose.override.yml` file contains environment variables and related overrides for a developer configuration. The variants with `.vs.debug` and `.vs.release` provide environment settings that enable Visual Studio to attach to and manage the running container.

While Visual Studio integration is part of adding Docker support to your solution, you can also build and run from the command line, using the `docker-compose up` command, as you saw in previous sections.

Getting data from the existing catalog .NET Core microservice

You can configure the Web Forms application to use the *eShopOnContainers* catalog microservice to get data instead of using fake data. To do this, you edit the `web.config` file and set the value of the `useFake` key to `false`. The DI container will use the class that accesses the live catalog microservice instead of the class that returns the hard-coded data. No other code changes are needed.

Accessing the live catalog service does mean you need to update the **docker-compose** project to build the catalog service image and launch the catalog service container. Docker CE for Windows supports both Linux containers and Windows containers, but not at the same time. To run the catalog microservice, you need to build an image that runs the catalog microservice on top of a Windows-based container. This approach requires a different Dockerfile for the microservices project than you've seen in earlier sections. The `Dockerfile.windows` file contains the configuration settings to build the catalog API container image so that it runs on a Windows container—for example, to use a Windows Nano Docker image.

The catalog microservice relies on the SQL Server database. Therefore, you need to use a Windows-based SQL Server Docker image as well.

After these changes, the docker-compose project does more to start the application. The project now starts the SQL Server using the Windows based SQL Server image. It starts the catalog microservice in a Windows container. And, it starts the Web Forms catalog editor container, also in a Windows container. If any of the images need building, the images are created first.

Development and production environments

There are a couple of differences between the development configuration and a production configuration. In the development environment, you run the Web Forms application, the catalog microservice, and SQL Server in Windows containers, as part of the same Docker host. In earlier sections, you've seen the SQL Server images deployed in the same Docker host as the other .NET Core-based services on a Linux-based Docker host. The advantage of running the multiple microservices in the same Docker host (or cluster) is that there is less network communication and the communication between containers has lower latency.

In the development environment, you must run all the containers in the same OS. Docker CE for Windows does not support running Windows- and Linux-based containers at the same time. In production, you can decide if you want to run the catalog microservice in a Windows container in a single Docker host (or cluster), or have the Web Forms application communicate with an instance of the catalog microservice running in a Linux container on a different Docker host. It depends on how you want to optimize for network latency. In most cases, you'll want the microservices that your applications depend on running in the same Docker host (or swarm) for ease of deployment and lower communication latency. In those configurations, the only costly communications is between the microservice instances and the high-availability servers for the persistent data storage.

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

Vision

Developing containerized microservice applications means you are building multi-container applications, however, a multi-container application could also be simpler (like a 3-tier application) and not necessarily following 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 microservice-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 microservice- 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.

Designing 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'll say that the app must support a variety of clients, including desktop browsers running SPA (Single Page Applications), 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 are responsible for handling the UI and consuming remote services.
- Domain/business logic. This is the application's domain logic.
- Database access logic. This consists of data access components responsible for accessing databases (SQL or NoSQL).
- Application integration logic. This includes a messaging channel, mainly based on message brokers.

The application will require high scalability, while allowing its vertical subsystems to scale out autonomously, as 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, being 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 don't 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 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), asynchronously whenever possible, especially when propagating updates.

Microservices are developed and deployed as containers independently of one another. This 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 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 app for .NET Core and microservices deployed using containers

So you can focus on the architecture and technologies instead of thinking about a hypothetical business domain that you might not know, that's why we have selected a well-known business domain which is 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 at the [eShopOnContainers](#) Github repo.

The application consists of multiple subsystems, including several store UI front ends (a Web app and a native mobile app), along with the back-end microservices and containers for all the required server-side operations. Figure 8-1 shows the architecture of the reference application.

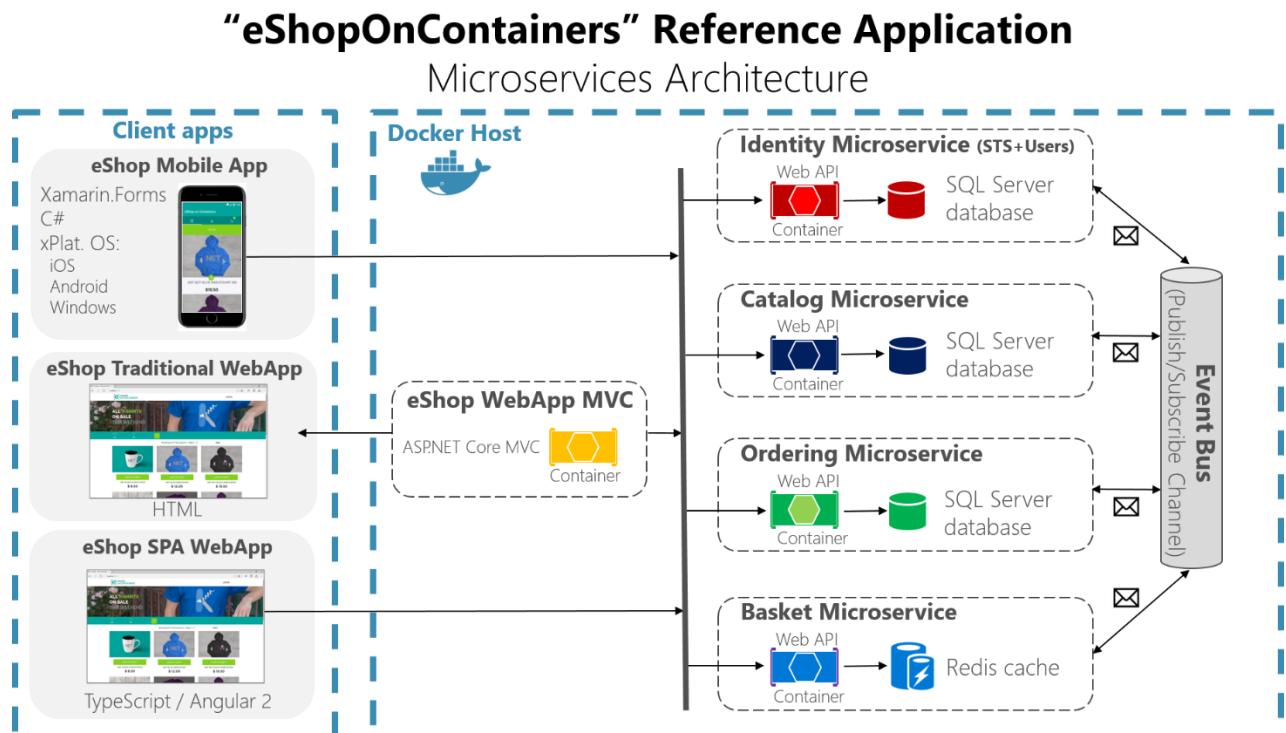


Figure 8-1. The eShopOnContainers reference app, showing the direct client-to-microservice communication and the event bus

Hosting environment. In Figure 8-1, 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're 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* app uses two communication types, depending on the kind of the functional action (queries versus updates and transactions):

- Direct client-to-microservice communication. This is used for queries and when accepting update or transactional commands from the client apps.
- Asynchronous event-based communication. This 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 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 containers as well as communicate between microservices. As mentioned, this initial architecture is using a direct client-to-microservice communication architecture, which means that a client app can make requests to each of the microservices directly. Each microservice has a public endpoint like `https://servicename.applicationname.companyname`. If required, each microservice can use a different TCP port. In production, that URL would map to the microservice's load balancer, which distributes requests across the available microservice instances.

As explained in the architecture section of this guide, the direct client-to-microservice communication architecture can have drawbacks when you are building a large and complex microservice-based application. But it can be good enough for a small application, such as in the *eShopOnContainers* application, where the goal is to focus on the microservices deployed as Docker containers.

However, if you are going to design a large microservice-based application with dozens of microservices, we strongly recommend that you consider the API Gateway pattern, as we explained in the architecture section.

Data sovereignty per microservice

In the sample application, each microservice owns its own database or data source, and each database or data source is deployed as another 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, to make it easy to compile the custom Docker images using .NET Core 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's 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 agility as it is easier to frequently deploy new versions of microservices.

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 in regards to the resources used when scaling out than a monolithic architecture.

You can divide the development work between multiple teams. Each service can be owned by a single dev 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 implementing wrong designs with direct dependencies between microservices), and other services can continue to handle requests. On the contrary, 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.

You can use the latest technologies. Because you can start developing services independently and run them side by side (thanks to containers and .NET Core), you can start using the latest technologies and frameworks 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. This 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 AMPQ, which adds complexity for testing and exception handling. It also adds latency to the system.

Deployment complexity. An application that has tens 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 the multiple microservices.

Increased global resource needs (total memory, drives, network for all the needed servers/hosts). In many cases, when you replace a monolithic application with a microservices approach, the amount of

global resources needed by the new microservice-based application will be larger than the infrastructure needs of the original monolithic application. This 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 just 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 app 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 app might need to make many separate requests to compose the UI, which can be inefficient over the public Internet and would be impractical over a mobile network. Therefore, requests from the client app 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 communication, 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 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 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 completely 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 service, 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 a different internal architecture, each based on different

patterns, for different microservices. The microservices can even use different technologies and programming languages, and different internal architecture and design patterns, as illustrated in Figure 8-2.

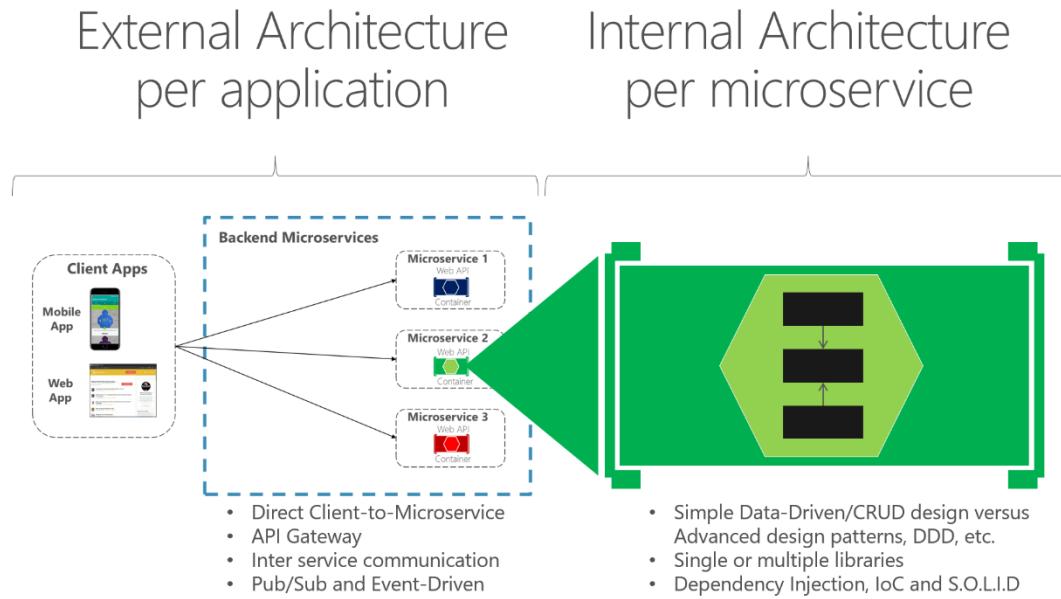


Figure 8-2. External versus internal architecture and design

For instance, in our eShop 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 eShop ordering microservice. (You will be able to review these DDD patterns in the section later that explains the implementation of the eShop 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](#) (Such as [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), 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 8-3 shows some approaches and technologies (although not in any particular order) that could be used in multiple and different microservices.

The Multi-Architectural-Patterns and polyglot microservices world



Figure 8-3. Multi-architectural patterns and the polyglot microservices world

As shown in Figure 8-3, in applications composed of many microservices (Bounded Contexts in Domain-Driven Design lingo, 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 app’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 sub-systems, you shouldn't apply a single top-level architecture based on a single architecture pattern. For instance, CQRS shouldn't 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.

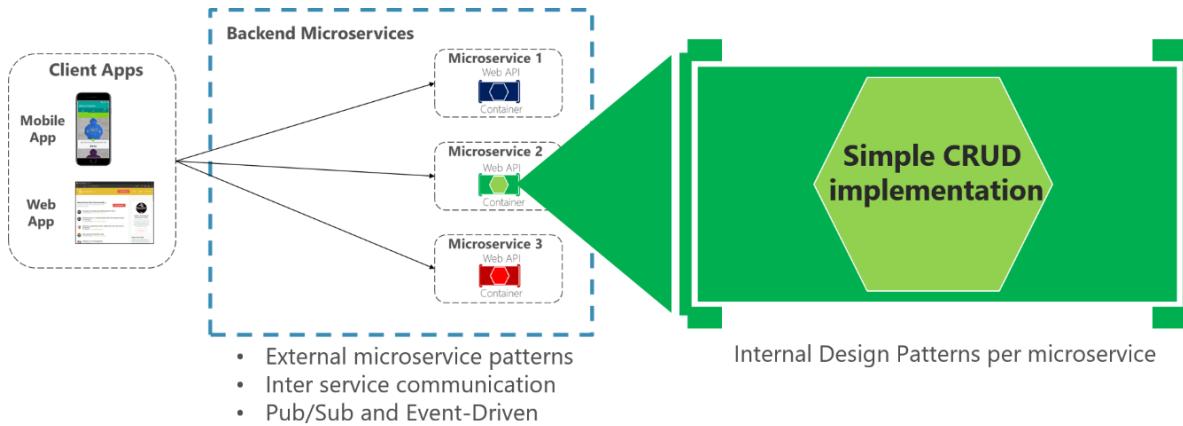


Figure 8-4. Internal Design for simpler CRUD microservices

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 8-5.

Data-Driven/CRUD microservice container

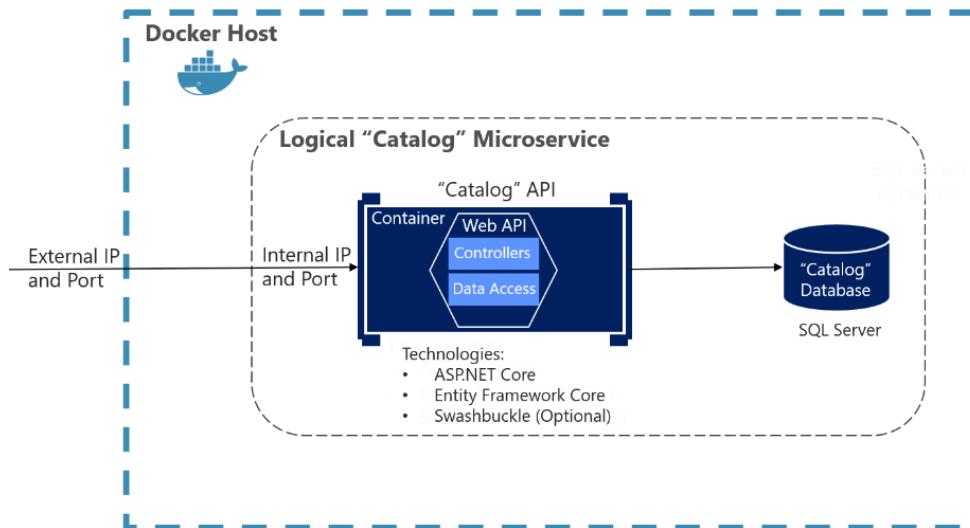


Figure 8-5. Simple data-driven/CRUD microservice design diagram

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 is very 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 to 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 DocumentDB.

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 Core and Visual Studio, you start by creating a simple ASP.NET Core Web API project (running on .NET Core so it can run on a Linux Docker host), as shown in Figure 8-6.

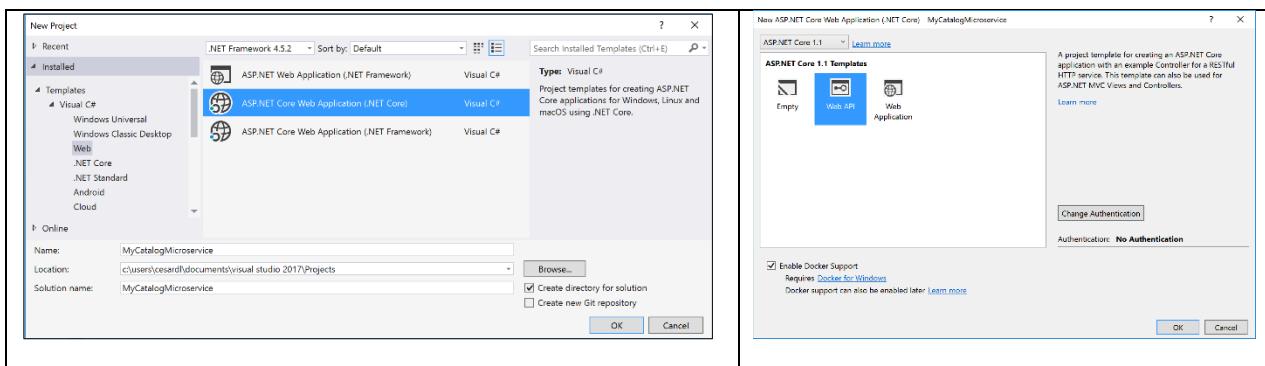


Figure 8-6. Creating an ASP.NET Core Web API project in Visual Studio

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 any other API. In the eShopOnContainers.Catalog.API project, you can see that the main dependencies for that microservice are just ASP.NET Core itself, Entity Framework, and Swashbuckle, as shown in Figure 8-7.

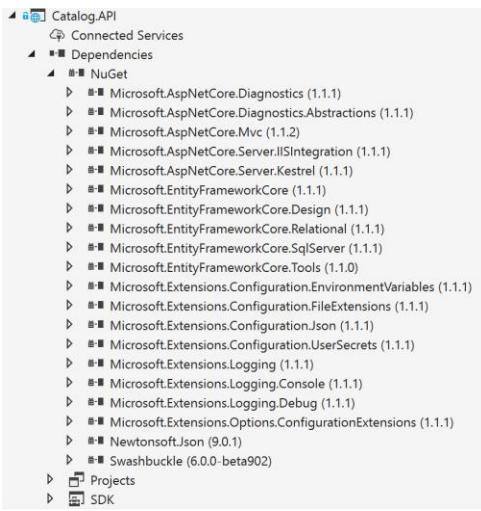


Figure 8-7. Dependencies in a simple CRUD Web API microservice

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.

Add Entity Framework Core to your dependencies

You can install the NuGet package for the database provider you want to use, in this case SQL Server, from within the Visual Studio IDE or with the NuGet console. Use the following command:

```
Install-Package Microsoft.EntityFrameworkCore.SqlServer
```

The data model

With EF Core, data access is performed by using a model. A model is made up of entity classes and a derived context 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 to create a database from your model (and evolve it as your model changes over time). For the catalog microservice we are using the last approach. You can see an example of the `CatalogItem` entity class in the following code example, which is a simple Plain Old CLR 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 PictureUri { get; set; }

    public int CatalogTypeId { get; set; }

    public CatalogType CatalogType { get; set; }

    public int CatalogBrandId { get; set; }

    public CatalogBrand CatalogBrand { get; set; }

    public CatalogItem() { }
}

```

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 code in the `DbContext` implementation. For example, in the sample application, we have a `OnModelCreating` method in the `CatalogContext` class that automatically populates the sample data the first time it tries to access the database. This method is useful for demo data. You can also use the `OnModelCreating` method to customize object/database entity mappings with many other [EF extensibility points](#).

You can see further details about `.OnModelCreating` in the *Implementing the Infrastructure-Persistence Layer with Entity Framework Core* section later in this book.

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;
    ((DbContext)context).ChangeTracker.QueryTrackingBehavior =
        QueryTrackingBehavior.NoTracking;
}

// GET api/v1/[controller]/items[?pageSize=3&pageIndex=10]
[HttpGet]
[Route("[action]")]
public async Task<IActionResult> Items([FromQuery]int pageSize = 10,
                                         [FromQuery]int pageIndex = 0)

{
    var totalItems = await _catalogContext.CatalogItems
        .LongCountAsync();
    var itemsOnPage = await _catalogContext.CatalogItems
        .OrderBy(c => c.Name)
        .Skip(pageSize * pageIndex)
        .Take(pageSize)
        .ToListAsync();
    itemsOnPage = ChangeUriPlaceholder(itemsOnPage);
    var model = new PaginatedItemsViewModel<CatalogItem>(
        pageIndex, pageSize, totalItems, itemsOnPage);
    return Ok(model);
} //...
}

```

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-code 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 have 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 Figure 8-10 above we are injecting an object of `CatalogContext` type plus other objects through the constructor `CatalogController()`.

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 `Startup` class by calling the `services.AddDbContext` method inside the `ConfigureServices` method, as shown in the following example:

```

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

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

```

Additional resources

- **Querying Data**
<https://docs.microsoft.com/en-us/ef/core/querying/index>
- **Saving Data**
<https://docs.microsoft.com/en-us/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=Pass@word",
  "ExternalCatalogBaseUrl": "http://localhost: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.

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=
      sql.data;Database=Microsoft.eShopOnContainers.Services.CatalogDb;
      User Id=sa;Password=Pass@word
    - ExternalCatalogBaseUrl=http://10.0.75.1:5101
    #- ExternalCatalogBaseUrl=
      http://dockerhoststaging.westus.cloudapp.azure.com:5101

  ports:
    - "5101:5101"
```

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. Consider that the Docker images that you build per microservice do not contain the docker-compose.yml files, only binary files and configuration files for each microservice, including the Dockerfile. But the docker-compose.yml file is not deployed along with your application; it is used only at deployment time. Therefore, placing environment variables values in those docker-compose.yml files (even without encrypting the values) is more secure than placing those values in regular .NET configuration files that are deployed with your code.

Finally, you can get that value from your code by using Configuration["ConnectionString"], as shown in the ConfigureServices method in Figure 8-12 earlier.

However, for production environments, you might want to explore additional ways on how to store secrets like the connection strings. Usually that will be managed by your chosen orchestrator, like you can do with [Docker Swarm secrets management](#).

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 app uses URI versioning.

With URI versioning, as in the eShopOnContainers sample app, 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 URLs 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, 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

- **Scott Hanselman. ASP.NET Core RESTful Web API versioning made easy**
<http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>
- **Versioning a RESTful web API**
<https://docs.microsoft.com/en-us/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 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 (<http://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.

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 to your ASP.NET Web API projects. It supports either 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, or plain Web API deployed on containers, as in this case.

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 very 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 8-8.

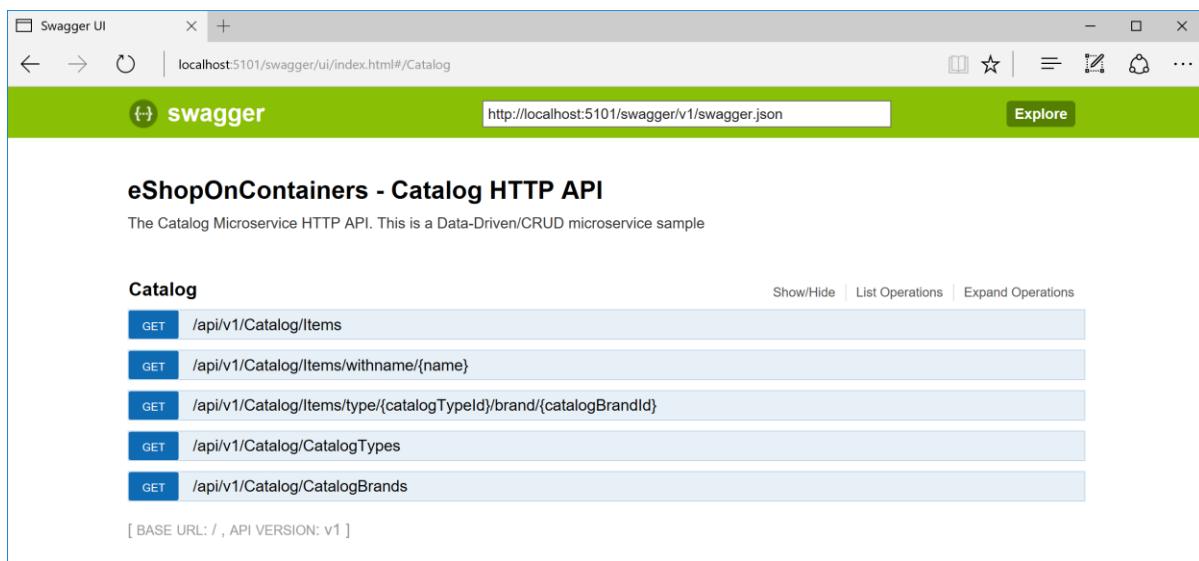


Figure 8-8. Swashbuckle UI (API Explorer) based on Swagger metadata—eShop Catalog microservice example

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 two NuGet packages: `Swashbuckle.SwaggerGen` and `Swashbuckle.SwaggerUi`. The former provides functionality to generate one or more Swagger documents directly from your API implementation and expose them as JSON endpoints. The latter provides an embedded version of the `swagger-ui` tool that can be served by your application and

powered by the generated Swagger documents to describe your API. However, the latest versions of Swashbuckle wrap these with the Swashbuckle.AspNetCore metapackage.

Note: For .NET Core Web API projects, you need to use [Swashbuckle.AspNetCore](#) version 1.0.0 or later.

Once you have installed these Nuget packages in your Web API project, you need to configure Swagger in the Startup class, as in the following code:

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

    // Other startup code...

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

        services.AddSwaggerGen();
        services.ConfigureSwaggerGen(options =>
        {
            options.DescribeAllEnumsAsStrings();
            options.SingleApiVersion(new Swashbuckle.Swagger.Model.Info()
            {
                Title = "eShopOnContainers - Catalog HTTP API",
                Version = "v1",
                Description = "The Catalog Microservice HTTP API",
                TermsOfService = "eShopOnContainers terms of service"
            });
        });

        // Other ConfigureServices() code...
    }

    public void Configure(IApplicationBuilder app,
                         IHostingEnvironment env,
                         ILoggerFactory loggerFactory)
    {
        // Other Configure() code...
        // ...
        app.UseSwagger()
            .UseSwaggerUi();
    }
}
```

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

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

You previously saw the generated UI created by Swashbuckle for a URL like `http://<your-root-url>/swagger/ui`. In Figure 8-9 you can also see how you can test any API method.

The screenshot shows the Swashbuckle UI for the eShopOnContainers - Catalog HTTP API. It displays the Catalog/Items endpoint. Parameters include pageSize (15) and pageIndex (1). Response messages show a 200 Success status. A 'Curl' section contains the command `curl -X GET --header 'Accept: application/json' 'http://localhost:5101/api/v1/Catalog/Items'`. The 'Request URL' is `http://localhost:5101/api/v1/Catalog/Items`. The 'Response Body' shows a JSON object representing a page of catalog items, with one item listed: { "pageIndex": 0, "pageSize": 10, "count": 4, "data": [{ "id": 4, "name": ".NET Red Black Sweatshirt" }] }.

Figure 8-9. Swashbuckle UI testing the Catalog/Items API method

Figure 8-10 shows the Swagger JSON metadata generated from the eShopOnContainer microservice (which is what the tools use underneath) when you request `<your-root-url>/swagger/v1/swagger.json` using [Postman](#).

The screenshot shows the Postman application's 'Builder' tab. It lists several API endpoints. One endpoint is selected: GET http://localhost:5101/swagger/v1/swagger.json. The 'Body' tab is open, showing the generated Swagger JSON. The JSON defines the API's schema, including its version (2.0), info (eShopOnContainers - Catalog HTTP API), and paths (including /api/v1/Catalog/Items).

```

{
  "swagger": "2.0",
  "info": {
    "version": "v1",
    "title": "eShopOnContainers - Catalog HTTP API",
    "description": "The Catalog Microservice HTTP API. This is a Data-Driven/CRUD microservice sample",
    "termsOfService": "Terms Of Service"
  },
  "paths": {
    "GET /api/v1/Catalog/Items": {
      "tags": [
        "Catalog"
      ],
      "responses": {
        "200": {
          "description": "Success"
        }
      }
    }
  }
}
  
```

Figure 8-10. 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://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger>

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

```

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

```

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 in <code>docker-compose.yml</code> | Description |
|--|--|
| <code>webmvc</code> | Container including the ASP.NET Core MVC app consuming the microservices from server-side C# |
| <code>catalog.api</code> | Container including the Catalog ASP.NET Core Web API microservice |
| <code>ordering.api</code> | Container including the Ordering ASP.NET Core Web API microservice |
| <code>sql.data</code> | Container running SQL Server for Linux, holding the microservices databases |
| <code>basket.api</code> | Container with the Basket ASP.NET Core Web API microservice |
| <code>basket.data</code> | 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=catalog.data;Initial Catalog=CatalogData;
                                              User Id=sa;Password=your@password
  expose:
    - "80"
  ports:
    - "5101:80"

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

  depends_on:
    - sql.data
```

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 per microservice database.
- The SQL Server name is `sql.data`, 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.

ImportantBecause 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 VSTS 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 `sql.data` service, (the SQL Server instance for Linux database running in a container). When you specify this dependency, the `catalog.api` container won't start until the `sql.data` container has already started; this is important because need 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 app 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 will 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, but other tools like orchestrators (for example, Docker Swarm) also understand that file.

Therefore, by using the `docker-compose` command you can target the following main scenarios.

Development environments

When developing applications, it is important to be able to run an application in an isolated development environment. You can use the `docker-compose` command line tool to create that environment or use 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" command line tool 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 at the same time are very 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.

```
docker-compose up -d  
./run_unit_tests  
docker-compose 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](#)), but it

could also be an entire [Docker Swarm](#) cluster, which also is compatible with the docker-compose.yml files.

If you are using any other orchestrator (Azure Service Fabric, Mesos DC/OS, Kubernetes, etc.), it is possible that you must 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 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 initial 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 8-11, when you are using Visual Studio and enabling Docker support, Visual Studio also creates those files plus some additional files used for debugging.

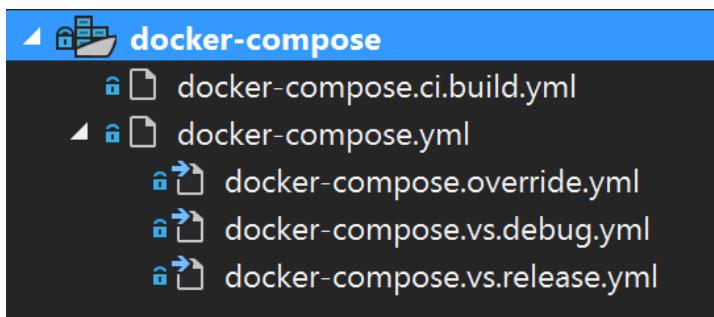


Figure 8-11. docker-compose files in Visual Studio 2017

You can edit the docker-compose files with any editor like Visual Studio Code or Sublime Text, and run the application with the `docker-compose up` command.

By convention, the `docker-compose.yml` file contains your base and more static configuration. 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. For instance, configuration that depends on the deployment environment. You can have multiple override files with different names, too. 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 8-12.

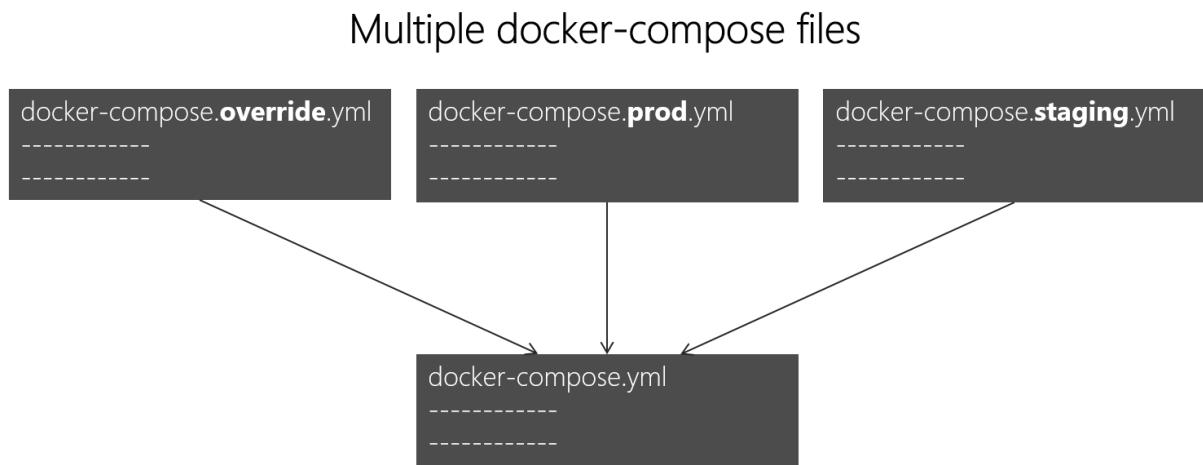


Figure 8-12. Multiple docker-compose files overriding values in the base docker-compose.yml file

You start with the base `docker-compose.yml` file. This base file has to contain the base or static configuration settings that do not change depending on the environment. For example, the eShopOnContainers has the following `docker-compose.yml` file as the base file.

```
#docker-compose.yml (Base)
version: '2'
services:
  basket.api:
    image: eshop/basket.api
    build:
      context: ./src/Services/Basket/Basket.API
      dockerfile: Dockerfile
    depends_on:
      - basket.data
      - identity.api
      - rabbitmq

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

  identity.api:
    image: eshop/identity.api
    build:
      context: ./src/Services/Identity/Identity.API
      dockerfile: Dockerfile
```

```

depends_on:
  - sql.data

ordering.api:
  image: eshop/ordering.api
  build:
    context: ./src/Services/Ordering/Ordering.API
    dockerfile: Dockerfile
  depends_on:
    - sql.data
    - rabbitmq

webspa:
  image: eshop/webspa
  build:
    context: ./src/Web/WebSPA
    dockerfile: Dockerfile
  depends_on:
    - identity.api
    - basket.api

webmvc:
  image: eshop/webmvc
  build:
    context: ./src/Web/WebMVC
    dockerfile: Dockerfile
  depends_on:
    - catalog.api
    - ordering.api
    - identity.api
    - basket.api

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

basket.data:
  image: redis
  expose:
    - "6379"

rabbitmq:
  image: rabbitmq
  ports:
    - "5672:5672"

webstatus:
  image: eshop/webstatus
  build:
    context: ./src/Web/WebStatus
    dockerfile: Dockerfile

```

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 useDockerfile
- 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: '2'

services:
# Simplified number of services here:
catalog.api:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://0.0.0.0:5101
    - ConnectionString=Server=sql.data; Database =
      Microsoft.eShopOnContainers.Services.CatalogDb; User Id=sa;Password=Pass@word
    - ExternalCatalogBaseUrl=http://localhost:5101
  ports:
    - "5101:5101"

identity.api:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://0.0.0.0:5105
    - SpaClient=http://localhost:5104
    - ConnectionStrings__DefaultConnection =
      Server=sql.data;Database=Microsoft.eShopOnContainers.Service.IdentityDb;User
      Id=sa;Password=Pass@word
    - MvcClient=http://localhost:5100
  ports:
    - "5105:5105"

webspa:
  environment:
    - ASPNETCORE_ENVIRONMENT=Development
    - ASPNETCORE_URLS=http://0.0.0.0:5104
    - CatalogUrl=http://localhost:5101
    - OrderingUrl=http://localhost:5102
    - IdentityUrl=http://localhost:5105
    - BasketUrl=http:// localhost:5103
  ports:
    - "5104:5104"

sql.data:
  environment:
```

```

- SA_PASSWORD=Pass@word
- ACCEPT_EULA=Y
ports:
- "5433:1433"

```

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 is merging both files.

Suppose that you want another Compose file for the production environment, with different configuration values. You can create another override file, like the following. (This file might be stored in a different Git repo or managed and secured by a different team.)

```

#docker-compose.prod.yml (Extended config for PRODUCTION env.)
version: '2'

services:
# Simplified number of services here:
catalog.api:
environment:
- ASPNETCORE_ENVIRONMENT=Production
- ASPNETCORE_URLS=http://0.0.0.0:5101
- ConnectionString=Server$sql.data; Database =
Microsoft.eShopOnContainers.Services.CatalogDb; User Id=sa;Password=Prod@Pass
- ExternalCatalogBaseUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5101
ports:
- "5101:5101"

identity.api:
environment:
- ASPNETCORE_ENVIRONMENT=Production
- ASPNETCORE_URLS=http://0.0.0.0:5105
- SpaClient=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5104
- ConnectionStrings__DefaultConnection =
Server$sql.data;Database=Microsoft.eShopOnContainers.Service.IdentityDb;User
Id=sa;Password=Pass@word
- MvcClient=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5100
ports:
- "5105:5105"

webspa:
environment:
- ASPNETCORE_ENVIRONMENT= Production
- ASPNETCORE_URLS=http://0.0.0.0:5104
- CatalogUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5101
- OrderingUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5102
- IdentityUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5105
- BasketUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5103
ports:
- "5104:5104"

sql.data:
environment:

```

```
- SA_PASSWORD=Prod@Pass  
- ACCEPT_EULA=Y  
ports:  
- "5433:1433"
```

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're specified on the command line. The following example shows how to deploy with a 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've shown in previous examples. You 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 "by default" values but could be overridden by the values you might have defined in each of your environments (host OS or env vars coming from your cluster). You place this .env file in the same folder where the docker-compose command is executed from.

The following example shows an .env file like the [.env defined at eShopOnContainers](#).

```
# .env file  
ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost  
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>.

Note that the values set in the runtime environment will always override the values defined inside the .env file. In a similar way, values passed via command-line command arguments will also override the default values set at the .env file.

Additional resources

- **Overview of Docker Compose**
<https://docs.docker.com/compose/overview/>
- **Multiple Compose files**
<https://docs.docker.com/compose/extends/#multiple-compose-files>

Building optimized ASP.NET Core Docker images

If you are exploring Docker and .NET Core 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 app. The following example shows a simple Dockerfile in this vein.

```
FROM microsoft/dotnet
WORKDIR /app
ENV ASPNETCORE_URLS http://+:80
EXPOSE 80
COPY . .
RUN dotnet restore
ENTRYPOINT ["dotnet", "run"]
```

A Dockerfile like this will work, but you can substantially optimize your images, especially your production images.

In the container and microservices model, you are constantly starting containers. The common way of using containers does not restart a sleeping container, because the container is disposable. Orchestrators (like Docker Swarm, Kubernetes, DCOS or Azure Service Fabric) simply create new instances of images. What this means is that you need to optimize, pre-compile the app when built so the instantiation process will be faster. When the container is started, it should be ready to run, you shouldn't restore and compile at run time (like if using dotnet restore and dotnet build commands from the dotnet cli that you can see in many blog posts about .NET Core and Docker in the Internet).

The .NET team has been doing important work to make .NET Core and ASP.NET Core a container-optimized framework. Not only is .NET Core a lightweight framework with a small memory footprint, but the team has focused on startup performance and produced some optimized Docker images, like the [microsoft/aspnetcore](#) image available at Docker Hub, in comparison to the regular [microsoft/dotnet](#) or [microsoft/nanoserver](#) images. The [microsoft/aspnetcore](#) image provides automatic setting of aspnetcore_urls to port 80 and the pre-ngend cache of assemblies; both of these settings result in faster startup.

Additional resources

- **Building Optimized Docker Images with ASP.NET Core**
[https://blogs.msdn.microsoft.com/stevlasker/2016/09/29/building-optimized-docker-images-with-asp-.net-core/](https://blogs.msdn.microsoft.com/stevlasker/2016/09/29/building-optimized-docker-images-with-asp-net-core/)

Building the application from a build (CI) container

Another benefit of Docker is that you can build your application from a preconfigured container, so you do not need to create a build machine or VM to build your application. You can use or test that build container running it at your development machine. But what's even more interesting is that you can use the same build container from your CI (Continuous Integration) pipeline.

Building the Application's bits from a container

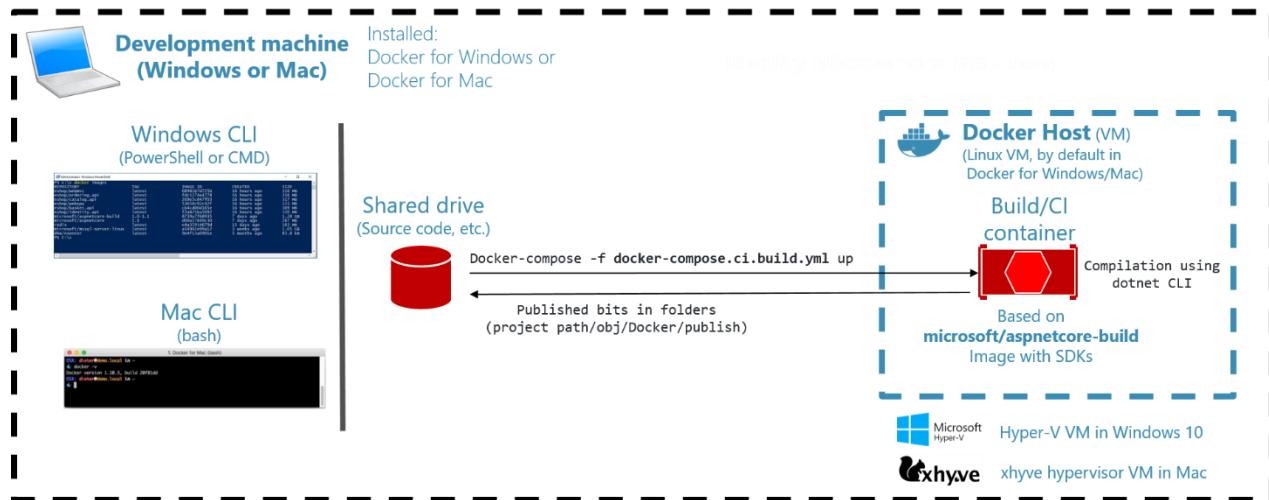


Figure 8-13. Components building .NET bits from a container

For this scenario we provide the [microsoft/aspnetcore-build](#) image, which you can use to compile and build your ASP.NET Core apps. The output is placed in an image based on the [microsoft/aspnetcore](#) image, which is an optimized runtime image, as previously noted.

The `aspnetcore-build` image contains everything you need in order to compile an ASP.NET Core app, including:

- .NET Core
- The ASP.NET SDK
- npm
- Bower
- Gulp

We need these dependencies at build time. But we do not want to carry these with the application at runtime, because it would make the image unnecessarily large. In the eShopOnContainers you can build the application bits from a container by just running the following `docker-compose` command.

```
docker-compose -f docker-compose.ci.build.yml up
```

Figure 8-14 shows this command running at the command line.

```
Administrator: Windows PowerShell
PS D:\GITREPOS\eshoponContainers> Docker-compose -f docker-compose.ci.build.yml up
Creating network "eshoponcontainers_default" with the default driver
Creating eshoponcontainers_ci-build_1
Attaching to eshoponcontainers_ci-build_1
ci-build_1 | Restoring packages for /src/src/Services/Catalog/Catalog.API/Catalog.API.csproj...
ci-build_1 | Restoring packages for /src/src/Services/Basket/Basket.API/Basket.API.csproj...
ci-build_1 | Restoring packages for /src/src/Services/Identity/Identity.API/Identity.API.csproj...
ci-build_1 | Installing Microsoft.AspNetCore.DataProtection.Abstractions 1.1.0.
ci-build_1 | Installing Microsoft.AspNetCore.Cryptography.Internal 1.1.0.
ci-build_1 | Installing Microsoft.DotNet.PlatformAbstractions 1.1.0.
```

Figure 8-14. Building your .NET application bits from a container

As you can see, the container running is the `ci-build_1` container. This is based on the `aspnetcore-build` image so that it can compile and build your whole application from within that container instead of from your PC. That's why, in reality, it is building and compiling the .NET Core projects in Linux—because that container is running on the default Docker Linux host.

The [`docker-compose.ci.build.yml`](#) file for that image (at `eShopOnContainers`) contains the following code. You can see that will start a build container using the [`microsoft/aspnetcore-build`](#) image.

```
version: '2'

services:
  ci-build:
    image: microsoft/aspnetcore-build:1.0-1.1
    volumes:
      - .:/src
    working_dir: /src
    command: /bin/bash -c "pushd ./src/Web/WebSPA && npm rebuild node-sass && pushd ../../.. && dotnet restore ./eShopOnContainers-ServicesAndWebApps.sln && dotnet publish ./eShopOnContainers-ServicesAndWebApps.sln -c Release -o ./obj/Docker/publish"
```

Once the build container is up and running, it runs the .NET SDK `dotnet restore` and `dotnet publish` commands against all the projects in the solution to compile the .NET bits. In this case, because `eShopOnContainers` also has an SPA based on TypeScript and Angular for the client code, it also needs to check JavaScript dependencies with `npm`, but that action is not related to the .NET bits.

The `dotnet publish` command builds and publishes the compiled bits within each project's folder to the `./obj/Docker/publish` folder, as shown in Figure 8-15.

| Name | Date modified | Type | Size |
|--|-------------------|-----------------------|--------|
| Pics | 3/9/2017 11:32 AM | File folder | |
| refs | 3/9/2017 11:32 AM | File folder | |
| runtimes | 3/9/2017 11:32 AM | File folder | |
| Catalog.API.deps.json | 3/13/2017 5:58 PM | JSON Source File | 195 KB |
| Catalog.API.dll | 3/13/2017 5:58 PM | Application extension | 47 KB |
| Catalog.API.pdb | 3/13/2017 5:58 PM | Program Debug Data... | 8 KB |
| Catalog.API.runtimeconfig.json | 3/13/2017 5:58 PM | JSON Source File | 1 KB |
| Dockerfile | 3/13/2017 5:58 PM | File | 1 KB |
| Microsoft.AspNetCore.Antiforgery.dll | 3/13/2017 5:58 PM | Application extension | 52 KB |
| Microsoft.AspNetCore.Authorization.dll | 3/13/2017 5:58 PM | Application extension | 39 KB |
| Microsoft.AspNetCore.Cors.dll | 3/13/2017 5:58 PM | Application extension | 31 KB |

Figure 8-15. Binary files generated by the `dotnet publish` command

Creating the Docker images from the CLI

Once the application bits are published at the related folders (with each project), the next step is to actually build the Docker images. To do this, you use the `docker-compose build` and `docker-compose up` commands, as shown in Figure 8-16.

Building the Docker images and running the containers

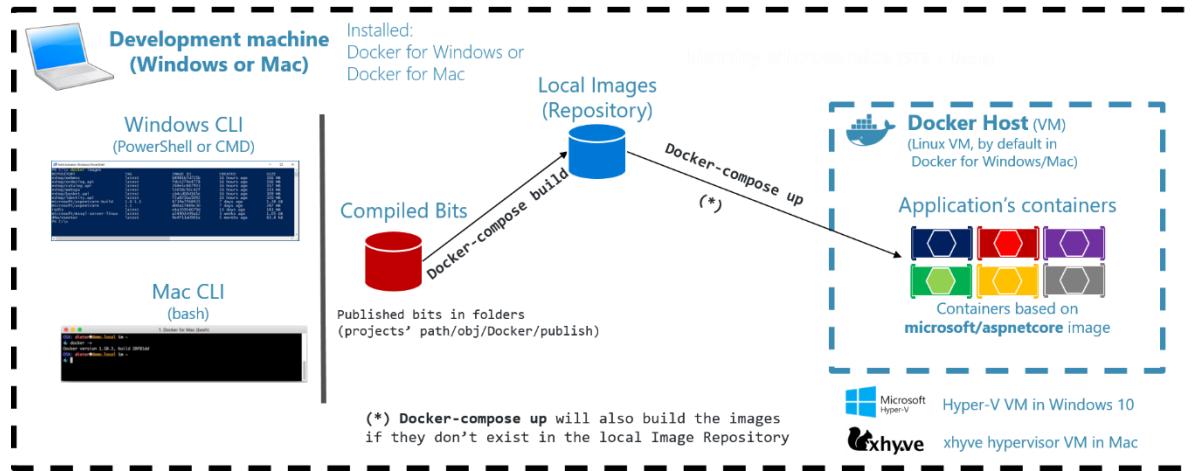


Figure 8-16. Building Docker images and running the containers

In Figure 8-17, you can see how the docker-compose build command runs.

```
Administrator: Windows PowerShell
PS D:\GITREPOS\eshopOnContainers> docker-compose build
basket.data uses an image, skipping
sql.data uses an image, skipping
Building identity.api
Step 1/6 : FROM microsoft/aspnetcore:1.1
--> d00a17849c30
Step 2/6 : ARG source
--> Running in b149b22c88c9
--> 16295b92d4ee
Removing intermediate container b149b22c88c9
Step 3/6 : WORKDIR /app
--> ac31d193b6dc
Removing intermediate container 3296c6a7c16f
Step 4/6 : EXPOSE 80
```

Figure 8-17. Building the Docker images with the docker-compose build command

The difference between the docker-compose build and docker-compose up commands is that docker-compose up both builds and starts the images.

When you use Visual Studio, all these steps are performed under the covers. Visual Studio compiles your .NET application bits, creates the Docker images, and deploys the containers into the Docker host. Visual Studio offers additional features, like the ability to debug your containers running in Docker, directly from Visual Studio.

The overall takeaway here is that you are able to build your application the same way your CI/CD pipeline should build it—from a container instead of from a local machine. After having the images created, then you just need to run the Docker images using the docker-compose up command.

Additional resources

- **Building bits from a container: Setting the eShopOnContainers solution up in a Windows CLI environment (dotnet CLI, Docker CLI and VS Code)**
[https://github.com/dotnet/eshopOnContainers/wiki/03.-Setting-the-eShopOnContainers-solution-up-in-a-Windows-CLI-environment-\(dotnet-CLI,-Docker-CLI-and-VS-Code\)](https://github.com/dotnet/eshopOnContainers/wiki/03.-Setting-the-eShopOnContainers-solution-up-in-a-Windows-CLI-environment-(dotnet-CLI,-Docker-CLI-and-VS-Code))

Using 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 do not have any external dependency, and a simply running the docker-compose 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 is a container named sql.data defined at the [docker-compose.yml](#) that runs SQL Server for Linux with all the SQL Server databases needed for the microservices. You could also have one SQL Server container for each database, but that would require more memory assigned to Docker. The important point in microservices is that each microservice owns its related data, therefore, its related SQL database, in this case. But the databases can be anywhere.

The SQL Server container in the sample application is configured with the following YAML code in the docker-compose.yml file that is executed when you run docker-compose up. Note that the YAML code has consolidated configuration information from the generic docker-compose.yml and the docker-compose.override.yml. (Usually you would separate the environment settings from the base or static information related to the SQL Server image.)

```
sql.data:
  image: microsoft/mssql-server-linux
  environment:
    - SA_PASSWORD=your@password
    - ACCEPT_EULA=Y
  ports:
    - "5434:1433"
```

The following docker run command could run that container:

```
docker run -e 'ACCEPT_EULA=Y' -e 'SA_PASSWORD= your@password' -p 1433:1433 -d
microsoft/mssql-server-linux
```

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://docs.microsoft.com/en-us/sql/linux/sql-server-linux-setup-docker>
- **Connect and query SQL Server on Linux with sqlcmd**
<https://docs.microsoft.com/en-us/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 Configure method in the Startup class of the Web API project:

```
public class Startup
{
    // Other Startup code...

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

        // Seed data through our custom class
        CatalogContextSeed.SeedAsync(app)
            .Wait();

        // Other Configure code...
    }
}
```

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 hold any 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 to add a duplicate value. But when using the in-memory database, you cannot handle that. Therefore, the in-memory database does not behave 100% the same as a real SQL Server database—it doesn't 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 `basket.data` container is defined as part of the multi-container `docker-compose.yml` file, as shown in the following example:

```
//docker-compose.yml file  
//...  
basket.data:  
  image: redis  
  expose:  
    - "6379"
```

This code in the docker-compose.yml defines a container named basket.data based on the redis image and publishing the port 6379 internally, meaning that it will be accessible only 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=basket.data  
    - EventBusConnection=rabbitmq
```

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 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 gives 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.

Implementing Asynchronous Event-Driven communication with an Event Bus

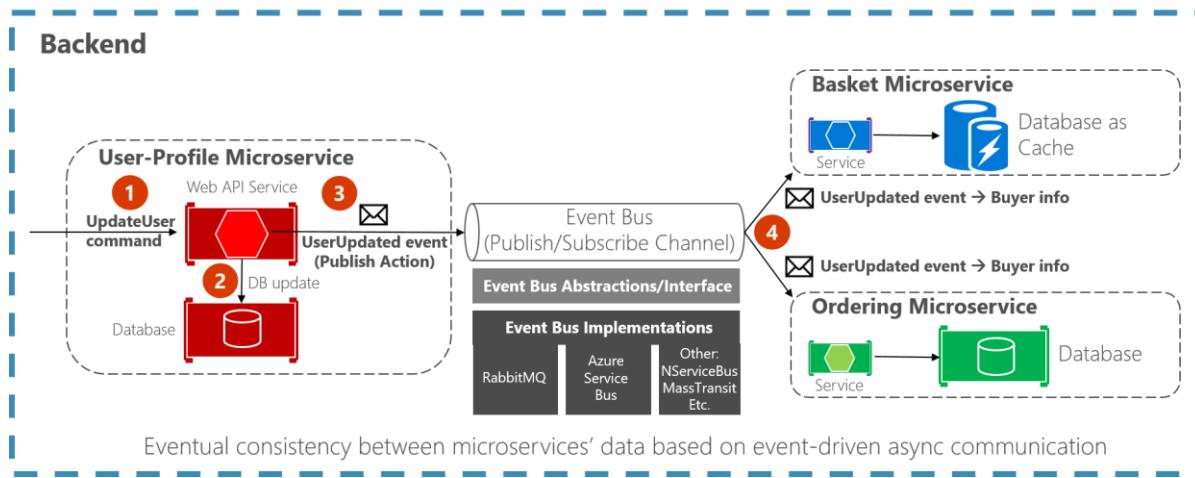


Figure 8-18. Event-driven communication based on an event bus

This section describes how you can implement this type of communication with .NET by using a generic event bus interface, as shown in Figure 8-18. 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 services 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 Fabric. 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. 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 present section, you should choose the service bus product in the market that best fits your needs.

Integration events

Integration events are used for bringing domain state in sync across multiple microservices or external systems. This 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 event class can be simple; for example, it might contain a GUID for its ID.

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 side. 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.

There are only a few kinds of libraries you should share across microservices. One is libraries that are final application blocks, like your [Event Bus client API](#), like 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 a publish/subscribe-style communication between microservices without requiring the components to explicitly be aware of each other, as shown in the Figure 8-19.

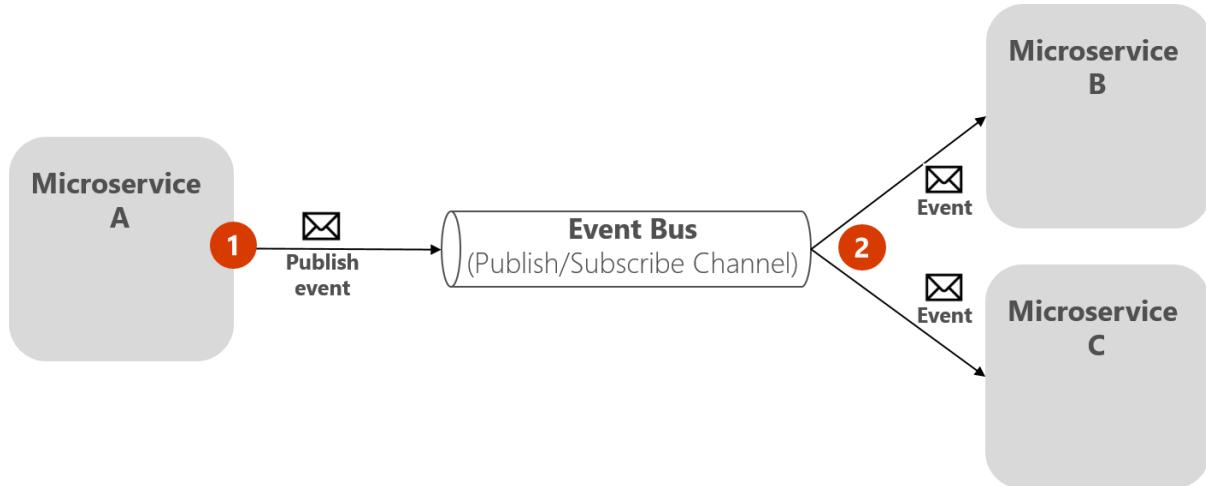


Figure 8-19. Publish/subscribe basics with an event bus

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 [Pub/Sub pattern](#) is the same as the Observer pattern: you want to notify other services when certain events take place. But there is an important semantic difference between the Observer and Pub/Sub patterns. In the Pub/Sub pattern, the focus is on broadcasting messages. In contrast, in the Observer pattern, the Observable does not want to know who the events are going to, just that they have gone out. In other words, the Observable (the publisher) does not want to know who the Observers (the subscribers) are.

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 8-19 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 could have multiple implementations so that you can swap between them, depending on the environment requirements (production versus development environments, for instance).

In Figure 8-20 you can see an abstraction of an event bus with multiple implementations based on infrastructure messaging technologies like RabbitMQ, Azure Service Bus, or other service buses like NServiceBus, MassTransit, etc.

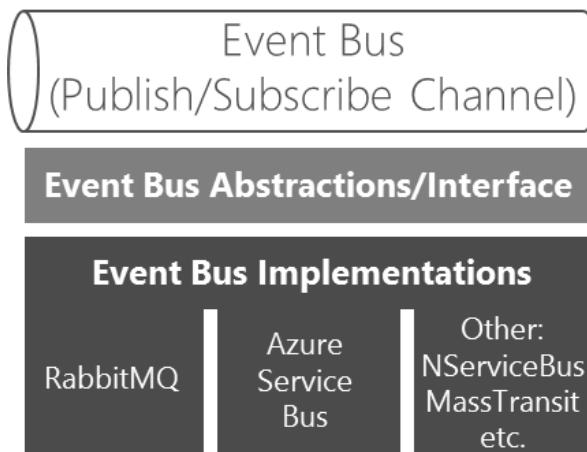


Figure 8- 20. Multiple implementations of an event bus

However, as highlighted previously, using abstractions (the event bus interface) is a possibility 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 provided by your preferred 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>(IIntegrationEventHandler<T> handler)
        where T: IntegrationEvent;
    void Unsubscribe<T>(IIntegrationEventHandler<T> handler)
        where T : IntegrationEvent;
}
```

The Publish method is straightforward. The event bus will broadcast the integration event passed to it to any microservice subscribed to that event. This method is used by the microservice that is publishing the event.

The Subscribe method is used by the microservices that want to receive events. This method has two parts. The first is the integration event to subscribe to ([IntegrationEvent](#)). The second part is the integration event handler (or callback method) to be called ([IIntegrationEventHandler<T>](#)) when the microservice receives that integration event message.

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. You should not use it for your production environment, unless you are building it as a part of a production-ready service bus. A simple custom event bus might be missing many production-ready critical features that a commercial service bus already has.

eShopOnContainer's custom implementation of an event bus is basically a library using RabbitMQ API. The implementation lets microservices subscribe to events, publish events, and receive events, as shown in Figure 8-21.

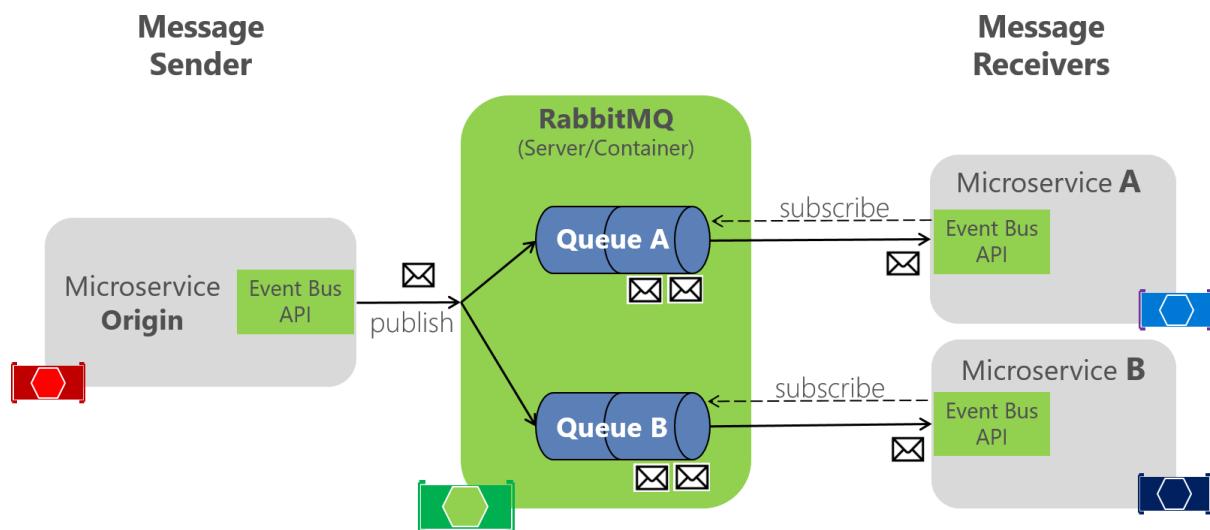


Figure 8-21. RabbitMQ implementation of an event bus

In the code, the `EventBusRabbitMQ` implements the generic `IEventBus` interface. This 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 8-21.

Implementing a simple publish method with RabbitMQ

The following code is part of the eShopOnContainers event bus implementation for RabbitMQ, so you usually do not need to code it unless you are making improvements. The code gets a connection and channel to RabbitMQ, creates a message, and then publishes the message into the queue.

```
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 at eShopOnContainers](#) is improved by using a Polly's retry policy in order to make a certain number of retries in case the RabbitMQ container is not ready for any reason, like when starting the containers with Docker-compose, when the RabbitMQ container might start slower 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>(IIntegrationEventHandler<T> handler)
        where T : IntegrationEvent
```

```

{
    var eventName = typeof(T).Name;
    if (_handlers.ContainsKey(eventName))
    {
        _handlers[eventName].Add(handler);
    }
    else
    {
        var channel = GetChannel();
        channel.QueueBind(queue: _queueName,
                           exchange: _brokerName,
                           routingKey: eventName);

        _handlers.Add(eventName, new List<IIIntegrationEventHandler>());
        _handlers[eventName].Add(handler);
        _eventTypes.Add(typeof(T));
    }
}
}

```

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 `IntegrationEventHandler` 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.

Subscribing to events

The first step for using the event bus is to subscribe the microservices to the events they want to receive. That 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. For instance, the `basket.api` microservice needs to subscribe to `ProductPriceChangedIntegrationEvent` messages. This makes the 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);

```

After this code runs, the subscriber microservice will be listening through RabbitMQ channels. When any message of type `ProductPriceChangedIntegrationEvent` comes, 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 is a simplified example not that does not take atomicity into account.) You would implement this code whenever an event has to be propagated across multiple microservices, usually right after committing data or transactions from the origin microservice.

First, the event bus implementation object (like based on RabbitMQ, or one 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("update")]
[HttpPost]
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem product)
{
    var item = await _context.CatalogItems.SingleOrDefaultAsync(
        i => i.Id == product.Id);
    // ...
    if (item.Price != product.Price)
    {
        var oldPrice = item.Price;
        item.Price = product.Price;
        _context.CatalogItems.Update(item);

        var @event = new ProductPriceChangedIntegrationEvent(item.Id,
                                                               item.Price,
                                                               oldPrice);
        // Commit changes in original transaction
        await _context.SaveChangesAsync();

        // Publish Integration Event to the event bus
        // (RabbitMQ or a service bus underneath)
        _eventBus.Publish(@event);
        // ...
    }
    return Ok();
}
```

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, it could be implemented in the `CommandHandler` or `DomainEventHandler` classes right after the original data is committed.

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. 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 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 database (or a microservice that owns its model) that is 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 deemphasize 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 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](#): 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 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

Document DB, MongoDB, Cassandra, CouchDB, or RavenDB. ES is a great approach for this problem, but not the most easiest solution unless you are already familiar with Event Sourcing.

The option to use transaction log mining initially looks very 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 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 Event Sourcing 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 won't 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: the application begins a local database transaction. It then updates the state of your domain entities and inserts an event into the integration event table. Finally, it commits the transaction. You get the desired atomicity.

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 happened 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 8-22 shows the architecture for the first of these approaches.

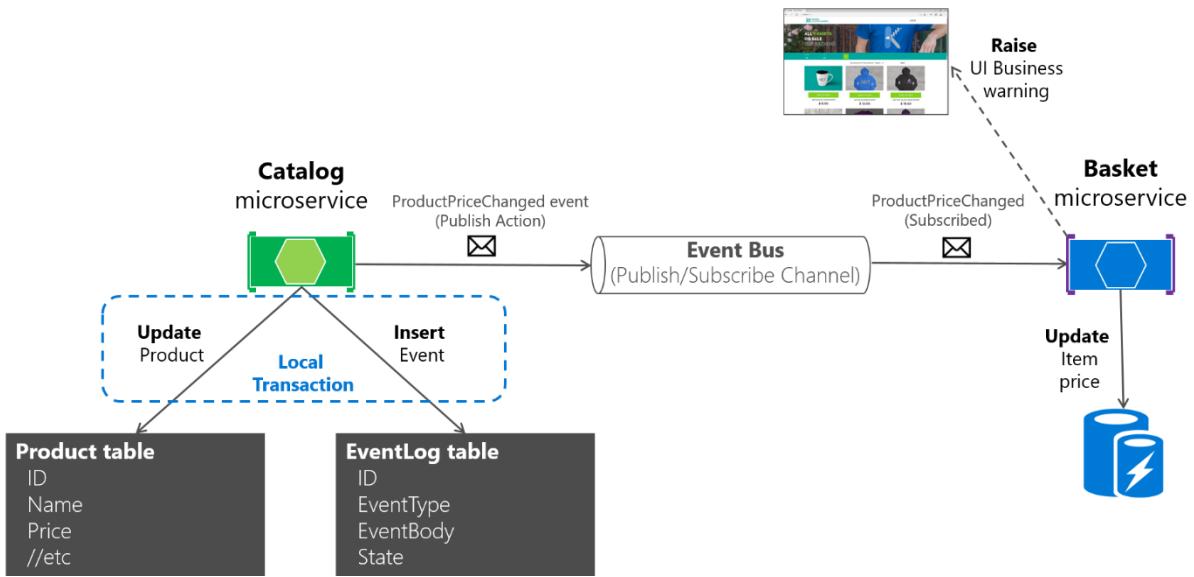


Figure 8-22. Atomicity when publishing events to the event bus

The approach illustrated in Figure 8-22 is missing an additional worker microservice that is in charge of checking/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.

About the second approach you would be using the EventLog table as a queue and always using a worker microservice to publish the messages, so the process would be like that shown in Figure 8-23. This shows an additional microservice, and the table is the single source when publishing events.

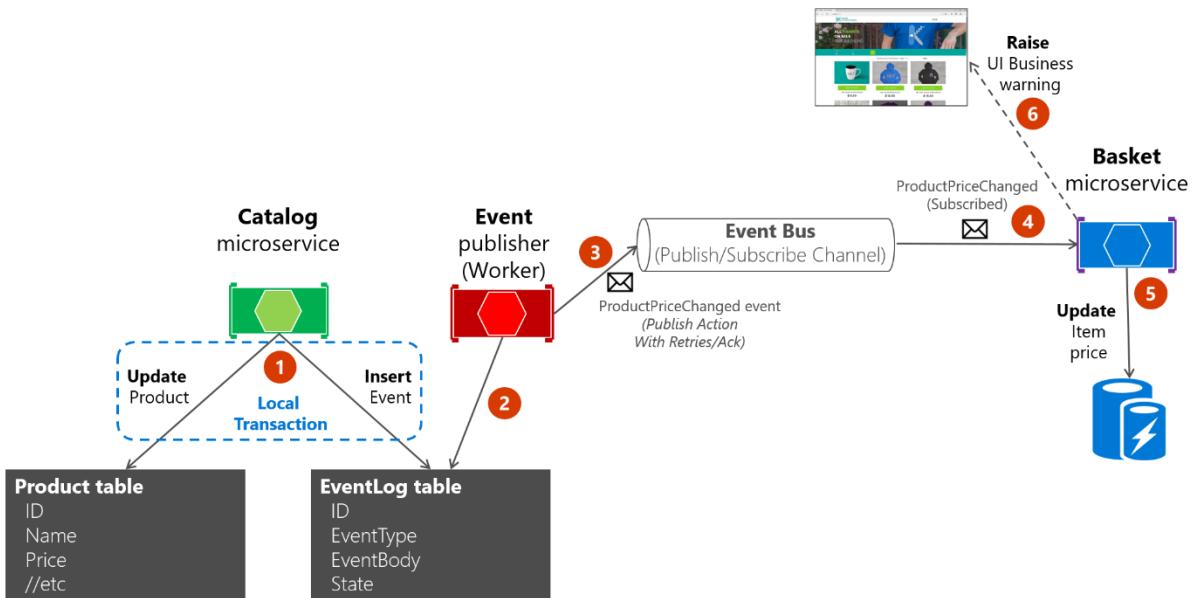


Figure 8-23. Atomicity publishing events to the event bus with a worker microservice

For simplicity, the eShopOnContainers sample uses the first approach (with no additional processes or checkers/workers microservices) plus the event bus. However, the eShopOnContainers is not handling all possible failure cases. In a real application implementation deployed to the cloud, you must

embrace the fact that issues will happen eventually and implement that check/validation and resend logic. Using the table as a queue can be more effective than the first approach by having that table as a single source of events when publishing them 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 being used.

Note that the transaction in the example code below will not be resilient if connections to the database have any issue at the time that the code runs. This is possible in cloud-based systems like Azure SQL DB, which might move databases across servers. For implementing resilient transactions across multiple contexts, check the “[Resilient Entity Framework Core Sql Connections](#)” section later in this guide.

For clarity, the example shows the whole process in a single piece of code. However, the eShopOnContainers implementation might be slightly refactored and split this logic into multiple classes so it is 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;

    // Achieving atomicity between original DB and the IntegrationEventLog
    // with a local transaction
    using (var transaction = _catalogContext.Database.BeginTransaction())
    {
        _catalogContext.CatalogItems.Update(catalogItem);
        await _catalogContext.SaveChangesAsync();

        // Save to EventLog only if product price changed
        if(raiseProductPriceChangedEvent)
```

```

        await _integrationEventLogService.SaveEventAsync(priceChangedEventArgs);

        transaction.Commit();
    }
    // Publish to Event Bus only if product price changed
    if (raiseProductPriceChangedEvent)
    {
        _eventBus.Publish(priceChangedEventArgs);

        integrationEventLogService.MarkEventAsPublishedAsync(
            priceChangedEventArgs);
    }
    return Ok();
}

```

Note how after the integration event "ProductPriceChangedIntegrationEvent" to be published 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, so it is a single transaction and you will always be able to check what event messages were properly sent or not.

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 sent.

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 originated in the catalog microservice, it will be handled in the basket microservice and change 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 :
        IIIntegrationEventHandler<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 8-24.

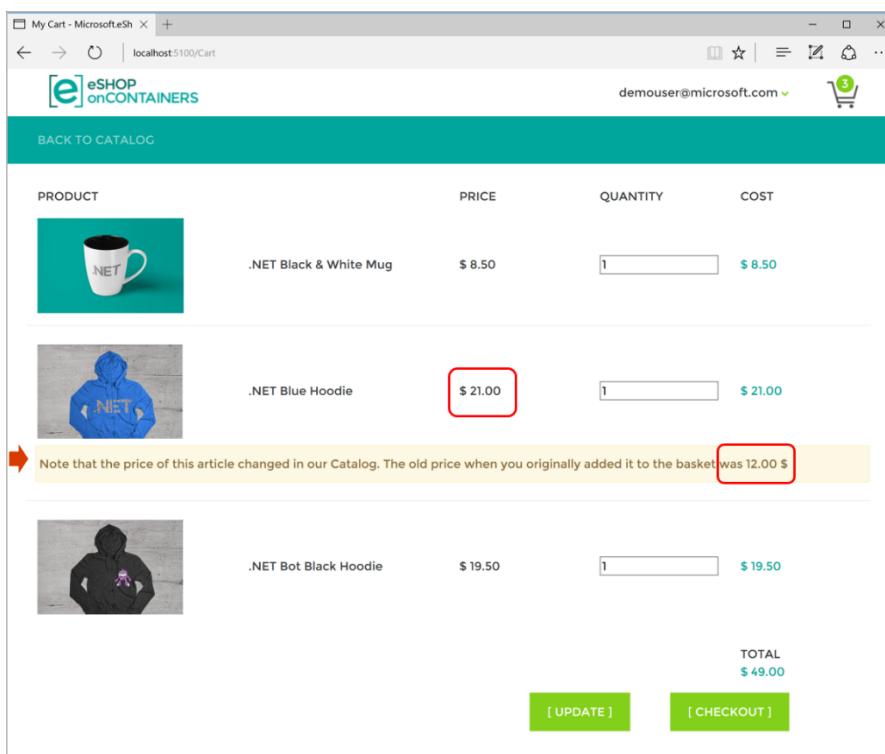


Figure 8-24. Displaying an item price change in a basket, as communicated by integration events

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 had already been published, creating a race condition. You need to 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 sentence that inserts data into a table only if that same data is not already stored in the table. It doesn't matter how many times you run that insert SQL sentence, 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 because of a retry logic the sender sends exactly the same message, you need to make sure that it will be 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, but not 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'd be losing that new price.

Another example would be an order-completed event being propagated to multiple subscribers. It is important that any order is propagated or updated in other systems just 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 based on that you can create logic enforcing that each event is processed only once per recipient.

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** (subhead on this page)
<https://msdn.microsoft.com/en-us/library/jj591565.aspx>

Deduplicating integration event messages

You can make sure that message events are sent and processed just 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.

De-duplicating message events at the EventHandler level

One way to do make sure that an event is processed just once by each destination recipient is by implementing certain logic when processing the message events at the event handlers. That approach is for instance the one chose by `eShopOnContainers` when receiving the [CreateOrderCommand](#), although in this case is an Http request command, not a message-based command, but in regards the logic you need to implement to make a command idempotent, can be pretty similar.

De-duplicating messages when using RabbitMQ

When intermittent network failures happen, messages can be duplicated and the message receivers must be ready to handle them. If possible, those receivers should handle messages in an idempotent way which would be better than explicitly handling with deduplication.

According to the [RabbitMQ documentation](#), “*if a message is sent to a consumer and the requeued (because it was not acknowledged before the consumer connection dropped, for example), RabbitMQ will set the redelivered flag on it when it is delivered again (whether to the same consumer or to a different one)*”.

If the “redelivered” flag is set, the receiver must take that into account as that message might has already been processed, but it’s not guaranteed, it might never have reached the receiver after it went out of the message broker because of any network issue.

On the other hand, if the “redelivered” flag is not set, it is for sure that the message has not been sent more than once. So, the receiver would only need to deduplicate messages or process messages in an idempotent way if the “redelivered” flag is set in the message.

Additional resources

- **Event Driven Messaging**
http://soapatterns.org/design_patterns/event_driven.messaging
- **Jimmy Bogard. Refactoring Towards Resilience: Evaluating Coupling**
<https://jimmybogard.com/refactoring-towards-resilience-evaluating-coupling/>
- **Publish-Subscribe channel**
<http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>
- **Communicating Between Bounded Contexts**
<https://msdn.microsoft.com/en-us/library/jj591572.aspx>
- **Eventual Consistency**
https://en.wikipedia.org/wiki/Eventual_consistency
- **Philip Brown. Strategies for Integrating Bounded Contexts**
<http://culttt.com/2014/11/26/strategies-integrating-bounded-contexts/>
- **Chris Richardson. Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 2**
<https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson>

- **Chris Richardson.** **Event Sourcing pattern**
<http://microservices.io/patterns/data/event-sourcing.html>
- **Introducing Event Sourcing**
<https://msdn.microsoft.com/en-us/library/jj591559.aspx>
- **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://msdn.microsoft.com/en-us/library/dn589800.aspx>
- **Rick Saling.** **The CAP Theorem: Why “Everything is Different” with the Cloud and Internet**
<https://blogs.msdn.microsoft.com/rickatmicrosoft/2013/01/03/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/>
- **Eric Brewer.** **CAP Twelve Years Later: How the "Rules" Have Changed**
<https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed>
- **Participating in External (DTC) Transactions (MSMQ)**
https://msdn.microsoft.com/en-us/library/ms978430.aspx#bdadotnetasync2_topic3c
- **Azure Service Bus. Brokered Messaging: Duplicate Detection**
<https://code.msdn.microsoft.com/Brokered-Messaging-c0acea25>
- **Reliability Guide** (RabbitMQ documentation)
<https://www.rabbitmq.com/reliability.html#consumer>

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 app. As such, you should have confidence they behave as intended for your app. 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 your microservices:

- *Unit tests.* These ensure that individual components of the app work as expected. Assertions test the component API.
- *Integration tests.* These 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 ensure that the app works as expected from the user's perspective.
- *Service tests.* These 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 app in isolation from its infrastructure and dependencies. When you unit test controller logic, only the contents 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. By focusing 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 void Add_new_Order_raises_new_event()
{
    // Arrange
    var street = "FakeStreet";
    var city = "FakeCity";
    // Other variables omitted for brevity ...

    // Act
    var fakeOrder = new Order(new Address(street, city, state, country, zipcode),
        cardTypeId, cardNumber,
        cardSecurityNumber, cardHolderName,
        cardExpiration);

    // Assert
    Assert.Equal(fakeOrder.DomainEvents.Count, expectedResult);
}
```

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, so you can run those tests faster. It is available in a NuGet component as `Microsoft.AspNetCore.TestHost` that can be added to integration test projects and used to host ASP.NET Core applications. It can serve test requests without the need for a real web host.

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 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://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing>
- **Steve Smith. Integration testing** (ASP.NET Core)
<https://docs.microsoft.com/en-us/aspnet/core/testing/integration-testing>
- **Unit testing in .NET Core using dotnet test**
<https://docs.microsoft.com/en-us/dotnet/articles/core/testing/unit-testing-with-dotnet-test>
- **xUnit.net.** Official site.
<https://xunit.github.io/>
- **Unit Test Basics.**
<https://msdn.microsoft.com/en-us/library/hh694602.aspx>
- **Moq.** GitHub repo.
<https://github.com/moq/moq>
- **NUnit.** Official site
<https://www.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 to run the whole application 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 (or similar ones like `docker-compose.ci.build.yml`), at the solution level you can expand the entrypoint 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.

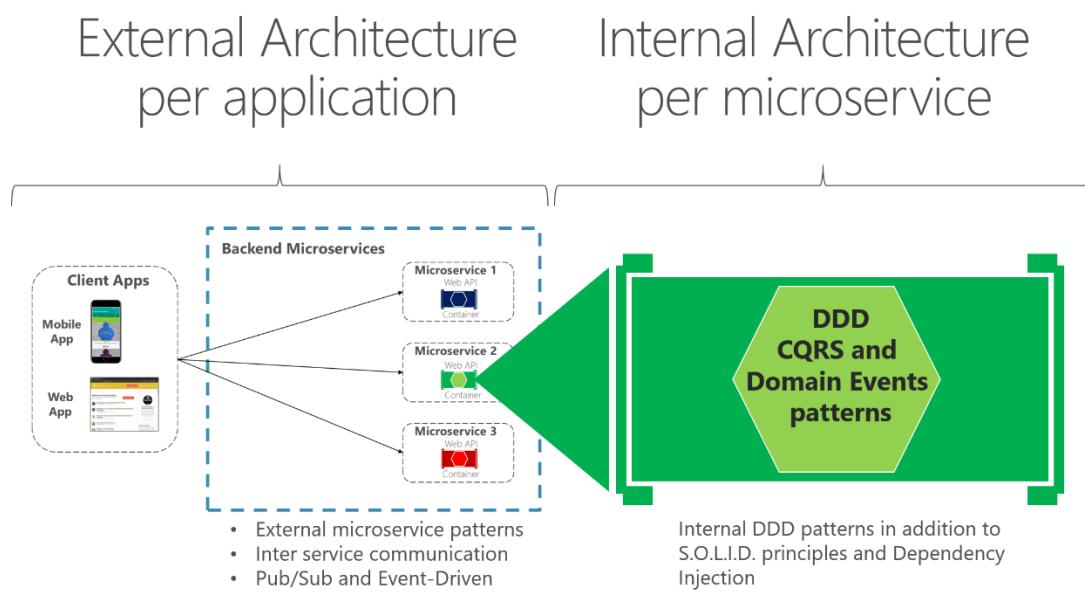
Once the compose application is up-and-running, you can take advantage of breakpoints and exceptions if you are running Visual Studio or you can run those integration tests automatically in your CI pipeline in Visual Studio Team Services or any other CI/CD system that supports Docker containers.

Tackling Business Complexity in a Microservice with DDD and CQRS Patterns

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, are also applicable to the more advanced microservices implemented internally with DDD (Domain-Driven Design) patterns. This section is an extension of the previous sections, as most of the practices explained earlier also apply here.

This section focuses on more advanced microservices that you implement when you need to tackle complexity of subsystems, or microservices derived from the knowledge of domain experts with ever-changing business rules.

This whole section focuses on the internal architecture, design and implementation of concrete microservices following more advanced patterns like the ones defined in DDD and CQRS, as illustrated in figure 9-1.



This chapter provides details on the simplified CQRS patterns used in the eShopOnContainers reference application. Later, you'll get an overview of the DDD techniques that enable you find common patterns that you can reuse in your applications.

DDD is a large topic with a rich set of resources you can use to learn. You can start by reading books like "[Domain-Driven Design](#)" by Eric Evans and many other literature from people like 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 of your concrete business domain.

References – Domain-Driven Design (DDD)

DDD (Domain-Driven Design)

<http://domainlanguage.com/>

<http://martinfowler.com/tags/domain%20driven%20design.html>

<https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/>

DDD Books

[Domain-Driven Design: Tackling Complexity in the Heart of Software](#) – Eric Evans

[Domain-Driven Design Reference: Definitions and Pattern Summaries](#) - Eric Evans

[Implementing Domain-Driven Design](#) - Vaughn Vernon

[Domain-Driven Design Distilled](#) - Vaughn Vernon

[Applying Domain-Driven Design and Patterns](#) - Jimmy Nilsson

[Domain-Driven Design Quickly](#)

DDD Training

Domain-Driven Design Fundamentals – Julie Lerman and Steve Smith

<http://bit.ly/PS-DDD>

Applying simplified CQRS and DDD patterns within a microservice

CQRS (Command Query Responsibility Separation) is an architectural pattern that separates the models for reading and writing data.

The related term [CQS \(Command Query Separation\)](#) 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:** Return a result and do not change the state of the system (and are free of side effects).
- **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 according with that definition. CQS could be It can be considered as a foundational principle for CQRS.

[CQRS \(Command and Query Responsibility Segregation\)](#) was introduced by Greg Young and also strongly promoted by Udi Dahan and other advocates. It is based on the CQS principle, although it is more detailed and can be considered a pattern based on commands and events plus optionally based on asynchronous messages. In many cases, CQRS is related to more advanced scenarios like having a

different physical database for the Reads/Queries than for the Writes/Updates. Going even further, a more evolved CQRS system would implement [Event-Sourcing \(ES\)](#) for your Updates/Writes database, so you would only store events in the Domain Model instead of the current state data. However, and as mentioned, this is not the case of this approach used in this guidance where we are using the simplest CQRS approach which is just separating the queries from the commands.

The separation pursued by CQRS is achieved by grouping query operations in one layer and commands in another layer. Each layer has its own model of data and is built using its own combination of patterns and technologies. More important, the two layers may be within the same tier or microservice (like the simplified chosen example approach in this guide) or they could even be on two distinct tiers/microservices/processes and be optimized separately without affecting each other.

CQRS means two objects for read/write where once there was one. There are reasons why you would want to have a de-normalized “reads-database” and you can learn about that in more advanced CQRS literature, but this is not the case for this more simplified approach where the main goal is to have higher flexibility in the queries instead of limiting the queries by constraints from DDD patterns like aggregates.

An example of this kind of service is the Ordering microservice from the *eShopOnContainers* reference application. This type of service implements a microservice based on a simplified CQRS (using a single data source or database, but two logical models) plus DDD patterns implementation for the transactional Domain, as shown in the design diagram in figure 9-2.

Simplified CQRS and DDD microservice

High level design

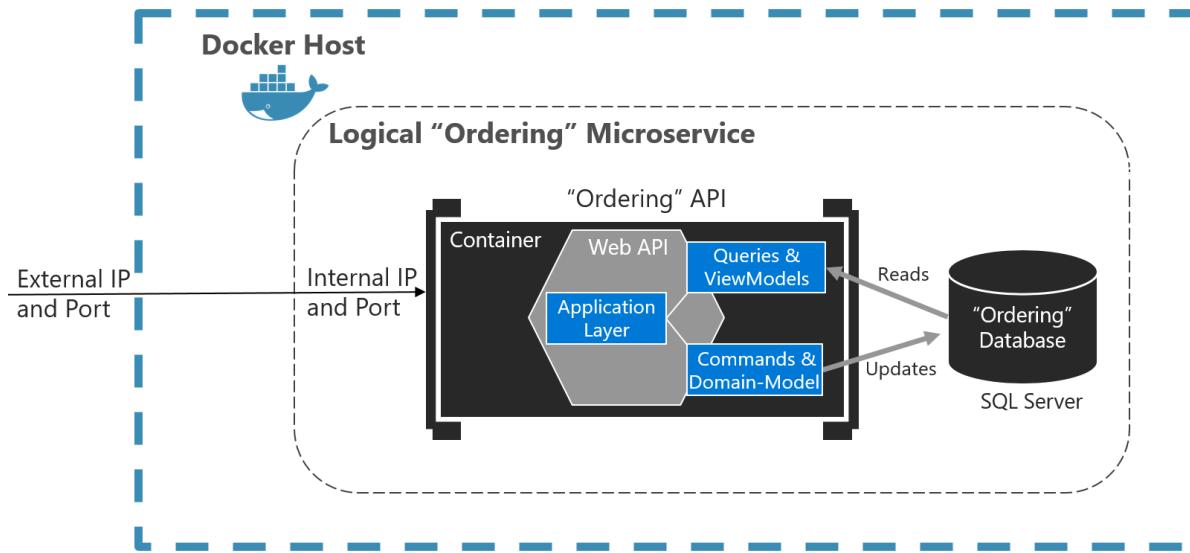


Figure 9-2. Simplified CQRS and DDD based microservice

The Application Layer can be the Web API itself. The important design decision here is that the microservice has split the Queries and ViewModels (Data models especially made for the client applications) from the Commands, Domain Model and transactions following a ([CQRS or Command and Query Responsibility Segregation](#)). This approach keeps the queries independent from restrictions

and constraints coming from Domain-Driven Design patterns that only make sense to transactions and updates, as explained in later sections.

CQRS and CQS approaches in a DDD microservice

The reference app design is based on CQRS principles but using the simplest approach, which is just separating the queries from the commands/updates and initially 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 because of the querying. Therefore, you could use a different "reads-data-model" than the transactional logic "writes-domain-model".

On the other hand, commands (which will trigger transactions and data updates) are what change state in your system. The commands and updates are where you need to be careful when dealing with complexity and ever-changing business rules. This is the area where you want to apply Domain-Driven Design techniques to have a more better modelled system.

The DDD patterns presented here 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. In the Aggregate pattern, you treat many domain objects as a single unit, because of their relationship in the domain. You may not always gain advantages from this patterns in queries. It will increase the complexity of query logic. For read-only queries, you don't gain the advantages of treating multiple objects as a single entity. You only get the complexity.

This guide suggests, as shown in image 9-2, DDD patterns only to the transactional/updates area of your microservice (triggered by Commands). Queries can follow a simpler approach, but should be separated from commands and updates following a CQRS approach. You can do this by implementing straight queries using a Micro ORM like [Dapper](#) or any other Micro ORM. You could choose to implement any query based on SQL sentences to get the best performance, thanks to a very light framework with very little overhead.

Note that when using this approach, updates to your model that impact how entities are persisted to a SQL database will necessitate separate updates to SQL queries used by Dapper or other separate (non-EF) approaches to querying.

CQRS and DDD patterns are not top-level architectures

It's important to highlight that CQRS and most DDD patterns (like DDD Layers or a Domain Model with Aggregates) are not architectural styles but only architectural. Microservices, SOA, Event Driven Architecture are examples of architectural styles. They describe a system of many components (like an architecture composed by many microservices). CQRS and DDD patterns describe something inside a single system or component, in this case, something inside a microservice.

Different bounded contexts will employ different patterns. They have different responsibilities, and that leads to different solutions. It's worth emphasizing: forcing the same pattern everywhere leads to failure. Don't use CQRS and DDD patterns everywhere because many subsystems, Bounded Contexts or microservices are simpler and can be implemented in an easier way as simple CRUD services or any other approach depending on what you need to create.

There is only one architecture. It is the one of the system or end-to-end application you are designing. It has its own set of tradeoffs and decisions that have been made per Bounded Context, microservice or any boundary you can have per subsystems. Do not try to apply the same architectural patterns like CQRS or DDD everywhere.

References – CQRS

CQRS

<https://martinfowler.com/bliki/CQRS.html>

CQS vs. CQRS (by Greg Young)

<http://codebetter.com/gregyoung/2009/08/13/command-query-separation/>

CQRS Documents (Greg Young)

https://cqrss.files.wordpress.com/2010/11/cqrs_documents.pdf

CQRS, Task Based UIs and Event Sourcing (Greg Young)

<http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/>

Clarified CQRS (Udi Dahan)

<http://udidahan.com/2009/12/09/clarified-cqrs/>

CQRS

<http://cqrss.nu/Faq/command-query-responsibility-segregation>

Event-Sourcing (ES)

<http://codebetter.com/gregyoung/2010/02/20/why-use-event-sourcing/>

Implementing the Reads/Queries in a CQRS microservice

As the chosen Reads/Queries implementation example, the Ordering microservice from the *eShopOnContainers* reference application has implemented the queries independently from the Domain-Driven Design model and transactional area. Mainly because the demands for each are drastically different (Reads vs. Writes).

It is a very simple approach as show in figure 9-3 where the API interface would be implemented by the Web API controllers using any infrastructure (like a MicroORM like Dapper) and returning dynamic ViewModels depending on the needs from the UI applications.

High level “Queries-side” in a simplified CQRS

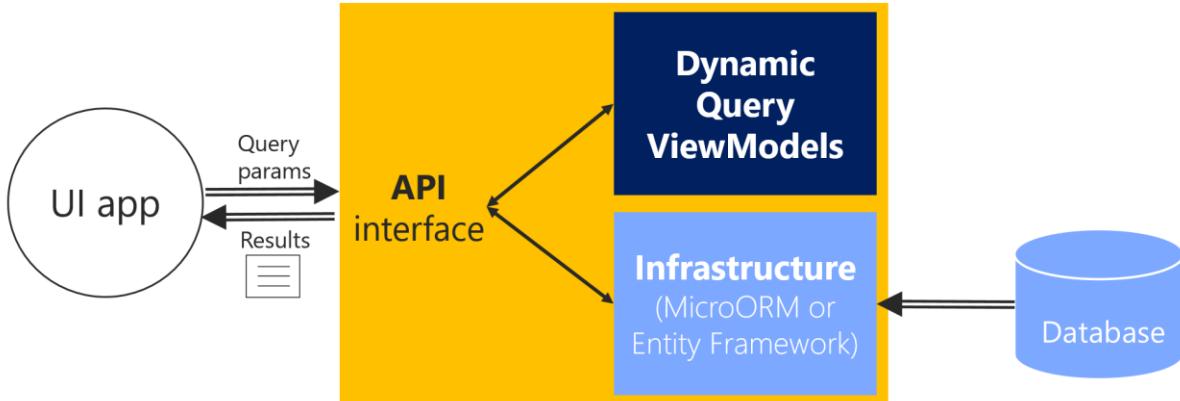


Figure 9-3. Simplest approach for queries in a CQRS microservice

This is the simplest possible approach for queries. The query definitions query the database and return a dynamic ViewModel built “on-the-fly” per each query. Since the queries are idempotent, you don’t need to be restricted by any DDD pattern used in the transactional side (like Aggregates and other patterns) but simply query the database for the data the UI needs and return that as a dynamic ViewModel that doesn’t need to be statically defined anywhere (no classes for the ViewModels) but in the SQL sentences themselves.

Since it is very simple approach, the required code for the “Queries side” like code using a MicroORM as [Dapper](#) can be implemented within the same Web API project as shown in figure 9-4 where the queries are defined in the Ordering.API microservice project within the eShopOnContainers solution.

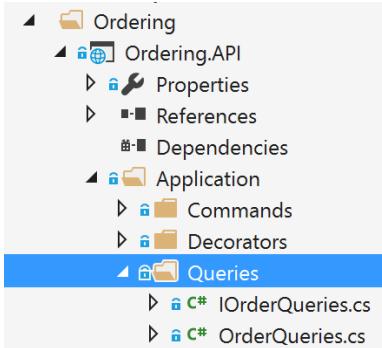


Figure 9-4. Queries in the Ordering microservice from eShopOnContainers

ViewModels specifically made for client apps, independent from the 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 them, based on the data returned by the queries. These specific models or DTOs (Data Transfer Object) are called ViewModels.

The returned data (ViewModel) can be the result of joining data from multiple entities or tables in the database 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 completely ignored and you are 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, or can also be created dynamically based on the queries performed, which is very agile for developers.

Dapper: selected Micro ORM as mechanism to query in the eShopOnContainers sample ordering microservice

You could use any Micro ORM, Entity Framework Core, or even plain ADO.NET for querying.

Dapper was selected for the Ordering microservice in the eShopOnContainers sample as a good example of a solid and popular Micro ORM. It can run plain and fast SQL queries with great performance, being a very light framework.

Dapper is an open source project (original created by Sam Saffron) and part of the building blocks used in [Stack Overflow](#).

Using Dapper, you can write a SQL query that could be accessing and joining multiple tables.

To use Dapper, you just need to install it through NuGet.



You will also need to add a using statement so your code has access to Dapper's extension methods.

When using Dapper in your code, you directly use the `SqlClient` class available in the `System.Data.SqlClient` namespace. Through the `QueryAsync<>()` method and other extension methods which extend the `SqlClient` class, you can simply run queries in a very straightforward and performant way.

Dynamic and static ViewModels

In the Ordering microservice, most of the ViewModels returned by the queries are implemented as dynamic. That means that the subset of attributes to be returned will be based on the query itself. If you add a new column to the query or join, that will be dynamically added to the returned ViewModel. This reduces the need to modify queries in response to updates to the underlying data model, making this design approach more flexible and tolerant of future changes.

```
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<dynamic> GetOrders()
    {
```

```

        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.StatusId = os.Id
        GROUP BY o.[Id], o.[OrderDate], os.[Name]");
    }
}

```

The important point to highlight is how by using a dynamic type, the returned collection of data will be dynamically assembled as the desired ViewModel.

For most of the queries you don't need to pre-define any DTO or ViewModel class so it is very straightforward code and very productive. However, you could also pre-define ViewModels (like pre-defined DTOs) if you want to have ViewModels with a more restricted definition as contracts.

References – Dapper

Dapper

<https://github.com/StackExchange/dapper-dot-net>

Data Points - Dapper, Entity Framework and Hybrid Apps (MSDN Mag. article by Julie Lerman)

<https://msdn.microsoft.com/en-us/magazine/mt703432.aspx>

Designing a domain-driven design-oriented microservice

Domain Driven Design advocates modeling based on the reality of business as relevant to your use cases. When 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*, *Aggregate* and *Aggregate-Root* rules to support the internal implementation. This section introduces the design and implementation of those internal patterns.

It is important to highlight that sometimes these DDD technical rules and patterns are perceived as hard barriers implementing DDD, but in the end, people tend to forget that the important part is to organize code artifacts in alignment with business problems and using the same common, ubiquitous language. Also, DDD approaches should be applied only when implementing complex microservices with ever-changing business rules. Simpler responsibilities, like a CRUD service can be managed with simpler solutions.

Where to draw the boundaries is the key task when designing, and defining a microservice. The Domain Driven Design patterns help you understand the complexity in the domain. You draw a bounded context around Entities, Value Objects, and Aggregates that model your domain. You build and refine a model that represents your domain and that model is contained within a boundary that defines your context. And that is very explicit in the form of a microservice. The components within those boundaries end up being your microservices. Microservices are about boundaries and so is DDD.

Keep the microservice context boundaries relatively small

Determining where to place boundaries between Bounded Contexts balances two competing goals: First, you want to create the smallest possible microservices. Second, you want to avoid chatty communications between microservices. These goals contradict each other. You should balance them by decomposing the system into as many small microservices as you can, until you start to see communication boundaries growing quickly with each additional attempt to separate a new Bounded Context.

Layers in domain-driven design microservices

All sufficiently complex enterprise applications consist of multiple layers. From a user's perspective, the layers are abstracted away and they exist solely to assist the programmer in managing all the emergent complexity. Distinct layers imply that translation must happen between some of the layers for information to propagate. For example, in a typical enterprise use case, an entity is loaded from the database, operated upon, persisted back to the database and information regarding the operation is returned to the user client app through a service/application layer, perhaps via a REST Web API service. The entity is contained within the domain layer and should not be forced into areas it doesn't belong, like in the presentation layer where a specific MVC view may require a user to enter information in several steps (basket, buying process, etc.). For instance, the user can enter the order's product item first, but the order might still have unspecified info about shipping or billing information. If the client application was using the Domain Entity, that target entity could be in invalid state. That is

not good. You need to have *Always-valid entities* (see the *Validations in Domain-Driven Design* section) controlled by *Aggregate-Roots*, so entities should not be bound to the client Views - this is what the *ViewModel* is for. The *ViewModel* is a building block of the presentation layer and the domain entity doesn't belong there. Instead, an appropriate domain layer entity should be created based on data contained in the view model. This can be done directly or by passing a *DTO* to a service. When tackling complexity, it is important to have a *Domain Model* controlled by *Aggregate-Roots* and following *Domain-Driven Design* patterns.

A service designed based on DDD patterns will usually be composed by several internal layers.

The following figure 9-5 shows how that design is implemented in the *eShopOnContainers* app.

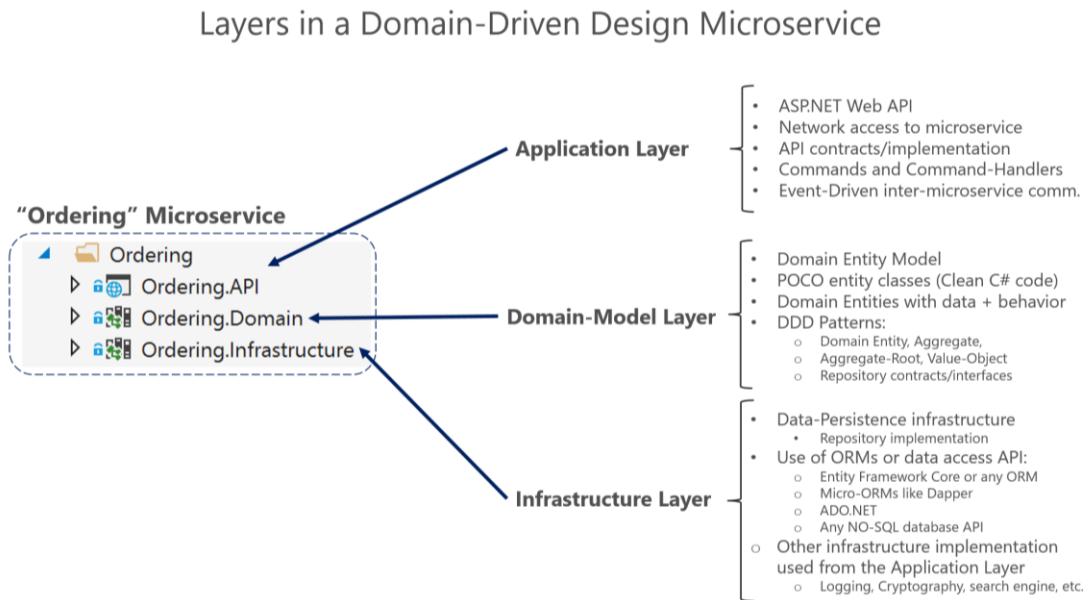


Figure 9-5. DDD Layers in the Ordering microservice from *eShopOnContainers*

Layers are abstractions. You want to design the system so that each layer communicates only with adjacent layers. That may be easier to enforce if layers are implemented as distinct class libraries. For instance, the Domain-Model Layer should not take any dependency on any other layer (the Domain Model classes should be [POCO](#) classes) as shown in figure 9-6 below about the *Ordering.Domain* layer library which only has dependencies with the .NET Core libraries.

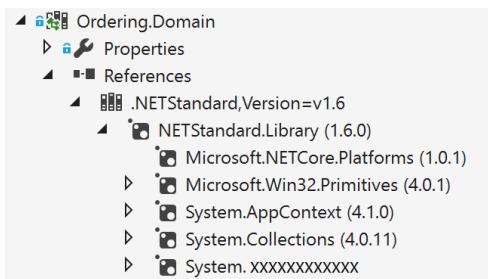


Figure 9-6. Layers implemented as libraries allow a better control of

Eric Evans's excellent book [Domain Driven Design](#) says the following about the Domain Model Layer and 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 Layer is where the business is expressed. When implementing a microservice's Domain Model Layer in .NET, that layer would be coded as a class library with the domain entities that will capture data plus behavior (methods).

Following the [Persistence Ignorance](#) and the [Infrastructure Ignorance](#) principles, this layer must completely ignore the 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 should be that your Domain Model entity classes should be [POCO](#) (Plain-Old CLR Objects). Domain Entities should not have any direct dependency with any data-access infrastructure framework like Entity Framework or NHibernate or any other data-access framework. Ideally, your Domain entities should not derive or implement any type defined in any infrastructure framework.

Luckily, most modern ORM frameworks like Entity Framework Core allow this approach so your domain model classes are not coupled to the infrastructure. However, having POCO entities is not always possible when using certain NoSQL persistence and frameworks like Actors and Reliable Collections in Azure Service Fabric.

“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 coded as an ASP.NET Core Web API project which implements the microservice's interaction, remote network access and external Web APIs to be 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. The ASP.NET Core Web API (representing the Application Layer) must not contain business rules or domain knowledge (especially domain rules for transactions or updates), which should be owned by the Domain Model class library. The Application Layer (in this case an ASP.NET Core Web API project) must only coordinate tasks and must not hold or define any domain state (domain model), but it will delegate the business rules execution to be run by 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, implementation for instance related to Web API or specific interfaces/contracts for your services front-end. The domain logic placed in the domain layer, however, is invariant to use cases and entirely reusable across all flavors of presentation and application layers you might have, and it must not depend on any infrastructure framework.

Infrastructure Layer: How the data initially held in domain entities in-memory will be persisted in databases or any other persistent store is a different matter. It will be implemented in the

Infrastructure Layer, as when using Entity Framework Core code to implement the Repository pattern classes that use DbContext to persist data in a relational database.

In accordance with the previously mentioned [Persistence Ignorance](#) and the [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 completely decoupled from invasive 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 the figure 9-7.

Dependencies between Layers in a Domain-Driven Design service

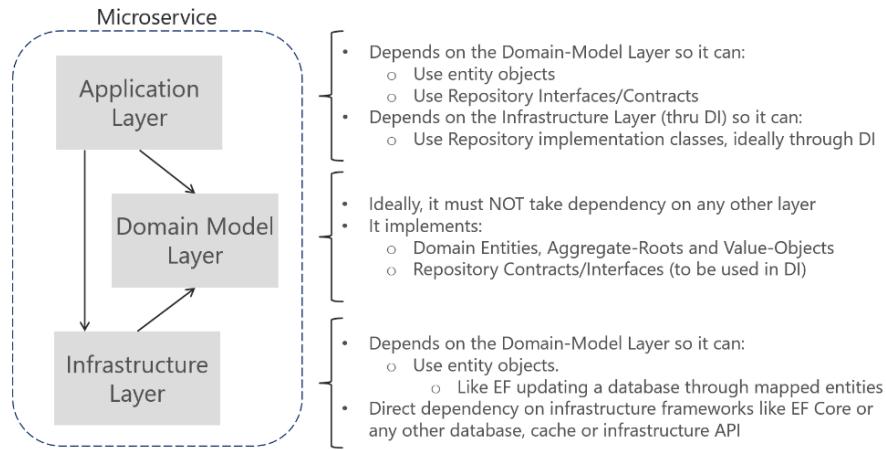


Figure 9-7. Dependencies between Layers in DDD

That layer's design should be independent per microservice, and as mentioned previously, you can implement your most complex microservices following DDD patterns, while implementing them in a much simpler way (simple CRUD in a single layer) for simpler data-driven microservices.

References – Persistence Ignorance principles

Persistence Ignorance principle

<http://deviq.com/persistence-ignorance/>

Infrastructure Ignorance principle

<https://ayende.com/blog/3137/infrastructure-ignorance>

Designing a microservice domain model

One rich Domain Model per Microservice

Your goal is to create a single cohesive domain model for each microservice. Each Bounded Context has its own Domain Model, and the implementation of that Domain Model is a Microservice.

The Domain model must capture the rules, behavior, business language and constraints of the single Bounded Context it represents.

The Domain Entity pattern

Entities represent domain objects and are primarily defined by their *identity, continuity, and persistence over time*, not only by the attributes that comprise them.

Per Eric Evans' definition, "*An object primarily defined by its identity is called an Entity*". Entities are very important in the Domain model and they should be carefully identified and designed.

Entities across multiple microservices or Bounded Contexts

The same identity may be modelled in multiple different bounded contexts. However, that does not imply that the same entity would be implemented in multiple bounded contexts. Rather, entities in each bounded context would limit its attributes and behaviors to those required in that bounded context. For instance, the Customer entity might have most of the person's attributes in the Profile or Membership microservice. However, the Buyer entity in the Ordering microservice (which shares its identity with the Customer entity) might have fewer attributes, because you only care about certain Buyer data related to the order process. The context of each microservice impacts the microservice's domain model.

Domain Entities must implement behavior in addition to data attributes

A Domain Entity in DDD must implement the domain logic 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 like adding an order item, data validation, or total calculation implemented as methods within the same entity class.

Figure 9-8 shows a diagram of a Domain Entity which clearly implements not only data attributes but also operations or methods with related domain logic.

Domain Entity pattern

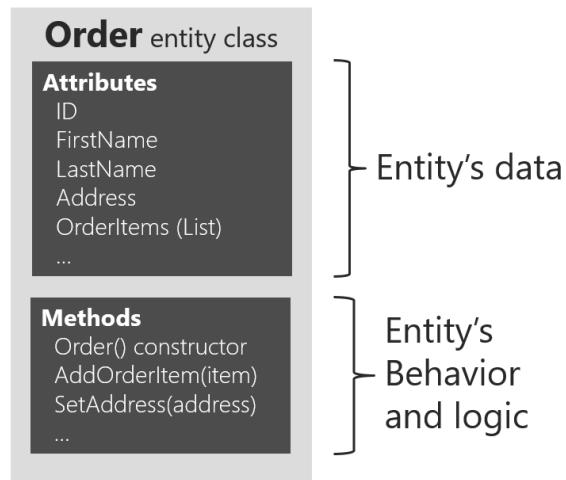


Figure 9-8. Example of Domain Entity Design implementing data plus

Of course, you could also have entities that do not implement any logic as part of the entity class, but this should only happen if that entity represents a DTO or other type with no domain logic. If you have a complex microservice that has a lot of logic implemented in the service classes instead of

within the domain entities, you could be falling into the Anemic Domain Model, explained in the following section.

Rich Domain Model vs. Anemic Domain Model

As Martin Fowler described in [Anemic Domain Model](#), an Anemic Domain Model is basically a data model implemented as a collection of classes with attributes or properties. There are entity objects, most of them based on the nouns in the domain space, and these objects related to the domain's logic. The catch comes when you look at the behavior of those entity objects, and you realize that there is hardly any behavior in these objects, making them little more than a DTO data class with getters and setters. Of course, these data models will be used from a set of service objects (typically named Business Layer) which capture all the domain or business logic. The Business Layer sits on top of the data-model and use that data-model just for 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 completely miss the point of what object-oriented design is all about. By putting all the behavior out into service objects (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, data-driven or CRUD, the anemic domain model (entity objects with just data properties) might be good enough and it might not be worth implementing more complex DDD patterns.

Some people might 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 and CRUD, probably it is not an anti-pattern. However, if you need to tackle the complexity of a specific microservice's Domain which has a lot of ever-changing business rules, then the Anemic Domain Model might be an anti-pattern for that particular microservice or Bounded Context and 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.

References – Domain Entity pattern , Domain Model and Anemic Domain Model

Domain Entity

<http://deviq.com/entity/>

The Domain Model

<https://martinfowler.com/eaaCatalog/domainModel.html>

The Anemic Domain Model

<https://martinfowler.com/bliki/AnemicDomainModel.html>

The Value-Object pattern

"Many objects do not have conceptual identity. These objects describe certain characteristics of a thing."
[Eric Evans]

There are many objects in a system that do not require an identity, whereas an Entity does.

The definition of Value-Object is: An object with no conceptual identity that describes a domain aspect. In short, these are objects that you instantiate to represent design elements which only concern you temporarily. You care about what they are, not who they are. Basic examples are numbers, strings, and such, but they also exist for higher level concepts like groups of attributes.

What may be an Entity in a microservice may not be an Entity in another microservice, because in the second case, Bounded Context might have a different meaning. For example, an address in some systems may not have an identity at all, since it may only represent a set of attributes of a person or company. That would be a Value-Object. That could be the case in an e-commerce application; the address may simply be a group of attributes of the customer's profile. In this case, the address doesn't have an identity per se and should be classified as a Value-Object pattern.

However, in other systems such as an application for an electric power utility company, the customer's 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 this case, an address should be classified as a Domain Entity.

References – Value-Object pattern

- <https://martinfowler.com/bliki/ValueObject.html>
- <http://deviq.com/value-object/>
- <https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects>
- Value-Object in "[Domain Driven Design](#)" Book - Eric Evans.

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 fulfilment or inventory. A more finely grained DDD unit is the Aggregate which describes a cluster or group of entities and behaviors that can be treated as a single 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 OrderItem 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 can't just pick some objects and say "this is an aggregate". You start with modelling a Domain concept and thinking about the entities that need to be used within your most common transactions, and then you can identify the aggregates in your model. Thinking about transaction operations is probably the best way to identify aggregates.

The Aggregate-Root or Root-Entity pattern

An aggregate will be composed of at least one entity: the Aggregate Root (AR), 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 placed 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, taking into account all the invariants and consistency rules you might need to comply with in your aggregate. If you change a child entity or VO (Value Object) independently, the Aggregate Root cannot ensure 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 9-9, 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.

Aggregate pattern

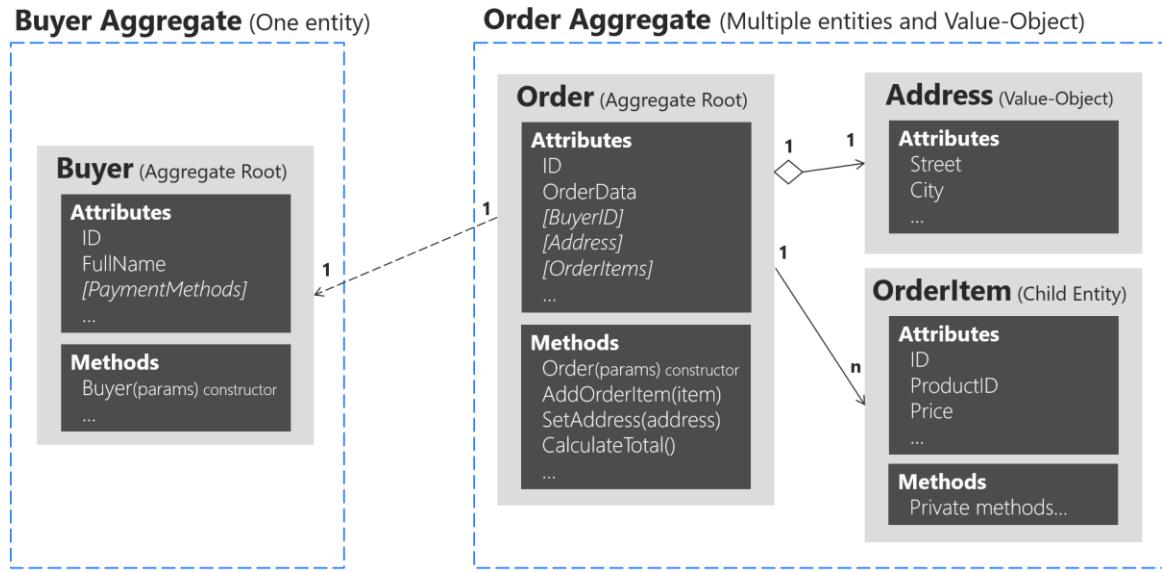


Figure 9-9. Aggregate pattern examples

Note that the Buyer aggregate could have additional child entities depending on your Domain, as it has in the sample Ordering microservice in the eShopOnContainers sample reference application. The figure 9-9 is just a case supposing that it could have a single entity, as an example of aggregate holding only an aggregate-root.

Identifying and working with aggregates requires research and experience. Below are a few articles and blog posts which drill down deeply into the subject and are very much recommended.

References – Aggregate related patterns

The Aggregate pattern

<http://deviq.com/aggregate-pattern/>

Effective Aggregate Design - Part I: Modeling a Single Aggregate

https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGRREGATES_PART_1.pdf

Effective Aggregate Design - Part II: Making Aggregates Work Together

https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGRREGATES_PART_2.pdf

Effective Aggregate Design - Part III: Gaining Insight Through Discovery

https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGRREGATES_PART_3.pdf

DDD Tactical Design Patterns

<https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part>

Developing Transactional Microservices Using Aggregates

<https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson>

Implementing a microservice's domain model with .NET Core

In the previous section, the fundamental design principles and patterns to design a domain model were explained. Now it's time to drill down into possible ways to implement the Domain Model by using .NET Core (plain C# code) and EF Core. (EF Core model requirements only. You shouldn't have hard dependencies or references to EF Core in your Domain Model).

Domain model structure in a .NET Core Standard Library

The folder organization used for the *eShopOnContainers* reference application demonstrates the DDD model for the application. You may find that a different folder organization more clearly communicates the design choices made for your application. As you can see in figure 9-10, in the Ordering Domain-Model there are two identified 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.

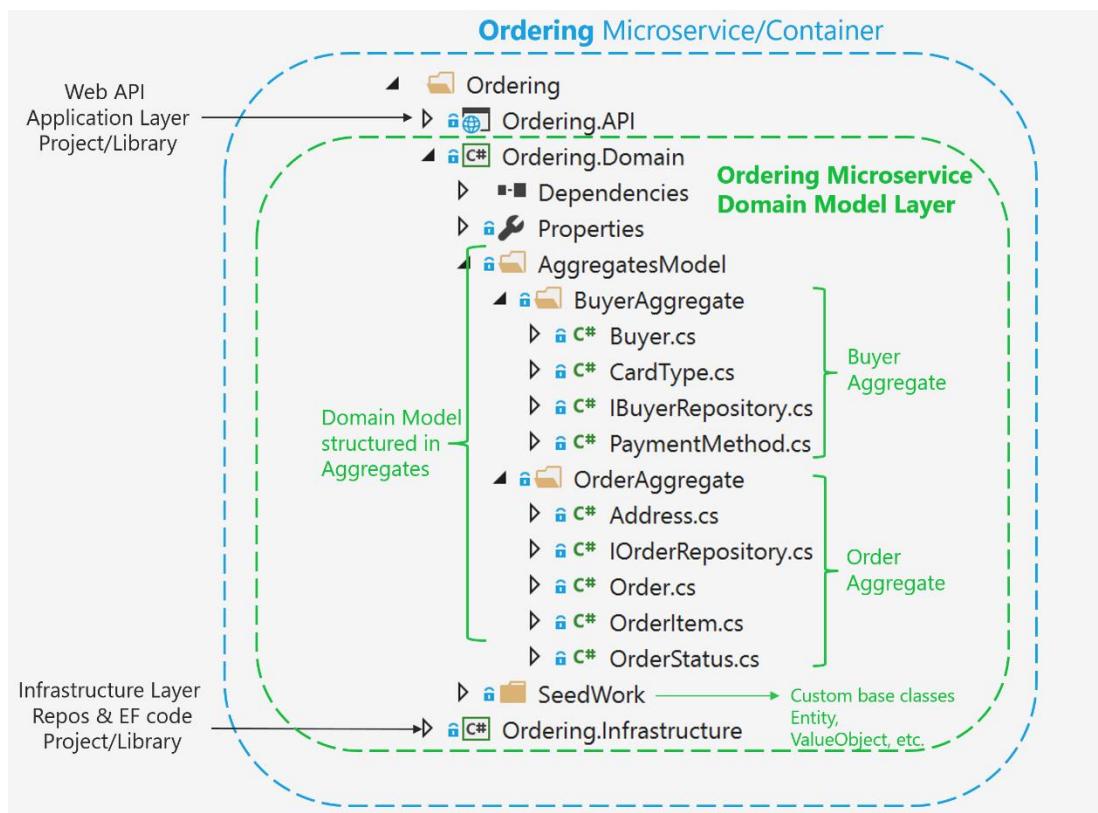


Figure 9-11. Domain Model structure for the Ordering

Additionally, in the Domain-Model layer includes the Repository contracts and interfaces that are the infrastructure requirements of your model, but not the infrastructure implementation of those repositories. They should be implemented outside of the domain model layer, in the infrastructure layer library.

You can also see a [SeedWork](#) folder which contains custom base classes that you can use as a base for your domain entities and value-objects, so you don't have to repeat redundant code in each domain's object class.

Structuring aggregates in a .NET Standard Library

The concept of 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 is composed of the following elements extracted from the eShopOnContainers Ordering microservice domain model, as shown in the figure 9-12.

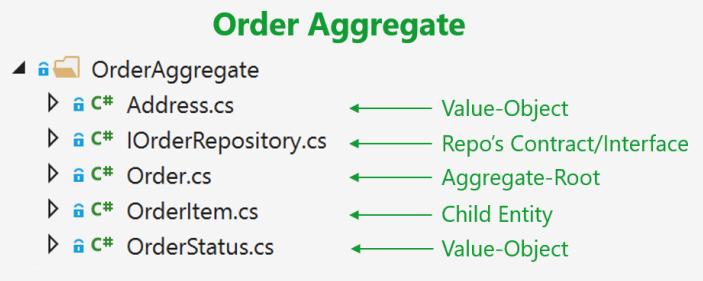


Figure 9-12. The "Order" aggregate in the VS solution

To see what kind of entity or object is contained in each class within an aggregate, you need to open its code and see how it is marked with the custom base classes or Interfaces implemented in the [SeedWork](#) folder.

Implementing domain Entities as POCO classes

The way you implement a domain model in .NET is by creating POCO classes that implement your domain entities. In the following code, the Order class is defined as an entity and also as an Aggregate Root. Because the Order class is deriving from the base class Entity, it can re-use common code related to entities. Keep in mind that these base classes and interfaces are defined by you here in the domain model project, so it is your code, not infrastructure code from any ORM like EF.

Entity Framework Core 1.0

```
// Entity is a custom base class with the Id
public class Order : Entity, IAggregateRoot
{
    public int BuyerId { get; private set; }
    public DateTime OrderDate { get; private set; }
    public int StatusId { get; private set; }
    public ICollection<OrderItem> OrderItems { get; private set; }
    public Address ShippingAddress { get; private set; }
    public int PaymentId { get; private set; }

    protected Order() { } //Needed only by EF Core 1.0

    public Order(int buyerId, int paymentId)
```

```

{
    BuyerId = buyerId;
    PaymentId = paymentId;
    StatusId = OrderStatus.InProcess.Id;
    OrderDate = DateTime.UtcNow;
    OrderItems = new List<OrderItem>();
}
public void AddOrderItem(string productName,
                        string pictureUrl,
                        decimal unitPrice,
                        decimal discount,
                        int units)
{
    //...
    // Domain Rules/Logic related to the OrderItem being added to the order
    // ...
    OrderItem item = new OrderItem(this.Id, ProductId, ProductName,
                                    PictureUrl, UnitPrice, Discount, Units);
    OrderItems.Add(item);
}

// ...
// Additional methods with Domain Rules/Logic related to the Order Aggregate
// ...

```

The important fact to highlight about the above code snippet is that this is a Domain Entity implemented as a POCO class. It doesn't have any direct dependency to Entity Framework Core or any other infrastructure framework. It is as it should be, just your C# code implementing your Domain Model.

In addition to that, it is also decorated with an interface named `IAggregateRoot`. That interface is an empty interface, sometimes called a *marker interface*, which is used just to say that this entity class is also an Aggregate-Root or the root entity of the aggregate. That means that most of the code related to the 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` to the Aggregate). You should not create or update `OrderItems` independently or directly; the `AggregateRoot` class must keep the control and consistency of any update operation against its child entities.

For example, you shouldn't do the following from any `CommandHandler` method or application layer class:

Wrong according to DDD patterns – Code at the application layer or Command Handlers

```

//My code in CommandHandlers 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() operation 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). Eventually you'll have spaghetti code, or a transactional script code implementation.

Following DDD patterns entities must not have public setters in any entity's property.

Going further, collections within the entity (like the order items) should be read-only properties (check the ".AsReadOnly()" pattern explained later) so you should be only able to update it from within the Aggregate root class methods.

As you can see in the code implementing the Order Aggregate-Root, all setters should be private, so any operation against the entity's data or its child entities will need to be performed through methods in the Aggregate-Root class. This will keep consistency in a more controlled and object-oriented way instead of doing a transactional script code implementation.

The following code snippet shows the proper code when adding an OrderItem to the Order aggregate.

Right according to DDD – Code at the application layer or Command Handlers

```
//My code in CommandHandlers or WebAPI controllers, only related to application stuff  
// NO code here related to OrderItem's business logic  
myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units);  
// The code related to OrderItem params validations or domain rules should be within AddOrderItem()  
//...
```

The important point here is that most of the validations or logic related to the creation of an OrderItem will be under the control of the Order aggregate-root, within the AddOrderItem() method, especially validations and logic related to other elements in the Aggregate. For instance, you might get the same product item as multiple AddOrderItem(params) invocations. In this method, you could check that out and consolidate the same product items in a single OrderItem 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.

In addition, the operation new OrderItem(params) will also be controlled and performed by the AddOrderItem() method from the Order aggregate-root, so most of the logic or validations related to that operation (especially if it 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 using Entity Framework 1.1, a DDD entity can be better expressed because one of the new features of Entity Framework Core 1.1 is that it allows [mapping to fields](#) in addition to properties. This is extremely useful when protecting collections of child entities or value objects.

Now, you can use simple fields instead of properties and implement any update to the field collection in public methods and providing read only access through the ".AsReadOnly()" pattern.

In DDD you want to update the entity only through methods in the entity (or the constructor) in order to control any invariant and consistency of the data, so properties with only a get accessor are defined. The properties are backed by private fields. Private members can only be accessed from within the class. However, there's one exception: EF Core needs to set these fields as well.

Entity Framework Core 1.1 or later

```
// Entity is a custom base class with the Id
public class Order : Entity, IAggregateRoot
{
    // DDD Patterns comment
    // Using private fields, allowed since EF Core 1.1, is a much better encapsulation
    // aligned with DDD Aggregates and Domain Entities (Instead of properties
    // and property collections)
    private bool _someOrderInternalState;
    private DateTime _orderDate;

    public Address Address { get; private set; }

    public Buyer Buyer { get; private set; }
    private int _buyerId;

    public OrderStatus OrderStatus { get; private set; }
    private int _orderStatusId;

    // DDD Patterns comment
    // Using a private collection field, better for DDD Aggregate's encapsulation
    // so OrderItems cannot be added from "outside the AggregateRoot" directly
    // to the collection, but only through the
    // OrderAggregateRoot.AddOrderItem() method which includes behavior.
    private readonly List<OrderItem> _orderItems;

    public IEnumerable<OrderItem> OrderItems => _orderItems.AsReadOnly();
    // Using List<>.AsReadOnly()
    // This will create a read-only wrapper around the private list so is protected
    // against "external updates". It's much cheaper than .ToList() because it will
    // not have to copy all items in a new collection.
    // (Just one heap alloc for the wrapper instance)
    // https://msdn.microsoft.com/en-us/library/e78dc75\(v=vs.110\).aspx

    public PaymentMethod PaymentMethod { get; private set; }
    private int _paymentMethodId;

    protected Order() { }

    public Order(int buyerId, int paymentMethodId, Address address)
    {
        _orderItems = new List<OrderItem>();
        _buyerId = buyerId;
        _paymentMethodId = paymentMethodId;
        _orderStatusId = OrderStatus.InProcess.Id;
        _orderDate = DateTime.UtcNow;
        Address = address;
    }

    // DDD Patterns comment
    // This Order AggregateRoot's method "AddOrderitem()" should be the only way
    // to add Items to the Order, so any behavior (discounts, etc.) and validations are
    // controlled by the AggregateRoot in order to maintain consistency
    // between the whole Aggregate.
}
```

```

public void AddOrderItem(int productId, string productName, decimal unitPrice,
                        decimal discount, string pictureUrl, int units = 1)
{
    // ...
    // Domain Rules/Logic related to the OrderItem being added to the order
    // ...
    OrderItem item = new OrderItem(this.Id, productId, productName,
                                   pictureUrl, unitPrice, discount, units);
    OrderItems.Add(item);
}

// ...
// Additional methods with Domain Rules/Logic related to the Order Aggregate
// ...
}

```

Mapping properties with only get accessors to the fields in the database table

When using EF 1.0, within the DbContext, you need to map the properties that you defined with only get accessors to the actual fields in the database table. This is done with the HasField method of the PropertyBuilder.

Mapping Fields without Properties

With this new feature in EF Core 1.1 to map columns to fields, it's also possible to not use properties, and instead just to map columns from a table to fields. A common use for that would be private fields for any internal state that doesn't need to be accessed from outside the entity.

For example, the `_someOrderInternalState` field has no related property for either setter or getter. That field will 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 1.1 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.

References – Implementing Aggregates and Domain Entities

Modeling Aggregates with DDD and Entity Framework (By Vaughn Vernon)

<https://vaughnvernon.co/?p=879> (Note that this is NOT Entity Framework Core)

Coding for Domain-Driven Design: Tips for Data-Focused Devs (Julie Lerman)

<https://msdn.microsoft.com/en-us/magazine/dn342868.aspx>

How to create fully encapsulated Domain Models (Udi Dahan)

<http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>

The SeedWork or reusable base classes and interfaces for your domain model

As mentioned, in the solution folder you can also see a SeedWork folder which contains custom base classes that you can use as a base for your domain entities and value-objects, so you don't have to repeat redundant code in each domain's object class.

It's called SeedWork instead of framework because it is just a small subset of reusable classes, but it cannot be considered a framework. [Seedwork](#) is a term introduced by Martin Fowler, but you could also name that folder "Common" or any other name.

Figure 9-12 shows the classes that form the SeedWork of the Domain Model in the Ordering microservice. It is just the custom “Entity” base class plus a few interfaces of the requirements asked to the implementation layer to have implemented. Those interfaces are also used through Dependency Injection from the application layer.

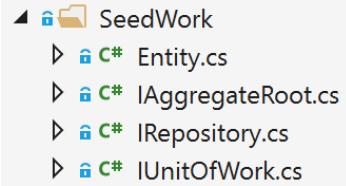


Figure 9-12. A sample Domain Model “Seedwork” with base classes and

This is the type of copy and paste reuse that many developers share between projects, not a formal framework. You can have SeedWorks within any layer or library, however, when it gets big enough, you might want to create a single class library just for itself.

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](#), etc.:

```
Entity Framework Core 1.1

public abstract class Entity
{
    int? _requestedHashCode;
    int _Id;

    public virtual int Id
    {
        get
        {
            return _Id;
        }
        protected set
        {
            _Id = value;
        }
    }

    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;

        Entity item = (Entity)obj;

        if (item.IsTransient() || this.IsTransient())
            return false;
    }

    public void RequestHashCode()
    {
        _requestedHashCode = hashCode();
    }

    protected int hashCode()
    {
        return _requestedHashCode ?? (_requestedHashCode = _Id);
    }
}
```

```

        else
            return item.Id == this.Id;
    }

    public override int GetHashCode()
    {
        if (!IsTransient())
        {
            if (!_requestedHashCode.HasValue)
                _requestedHashCode = this.Id.GetHashCode() ^ 31;
            // XOR for random distribution. See:
            // http://blogs.msdn.com/b/ericlippert/archive/2011/02/28/guidelines-
            // and-rules-for-gethashcode.aspx
            return _requestedHashCode.Value;
        }
        else
            return base.GetHashCode();
    }

    public static bool operator ==(Entity left, Entity right)
    {
        if (Object.Equals(left, null))
            return (Object.Equals(right, null)) ? true : false;
        else
            return left.Equals(right);
    }

    public static bool operator !=(Entity left, Entity right)
    {
        return !(left == right);
    }
}
}

```

Repository contracts and interfaces placed in the domain model layer

The Repository contracts are simply .NET interfaces that express the contract requirements of the Repositories to be used per each Aggregate. The Repositories themselves, with EF Core code or any other infrastructure dependencies and code, must not be implemented within the Domain Model; only the contracts or interfaces you demand to be implemented.

A pattern related to this practice (placing the Repository Interfaces in the Domain Layer) is the Separated Interface pattern defined by Martin Fowler as "*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 code snippet with the `IOrderRepository` interface defines what operations need to implement the `OrderRepository` in the infrastructure layer library. In the current implementation of the application it just needs to add the order to the database, since queries are split following the CQS approach and updates to Orders are not implemented in this implementation.

```
public interface IOrderRepository : IRepository<Order>
```

```

{
    Order Add(Order order);
}

public interface IRepository<T> where T : IAggregateRoot
{
    IUnitOfWork UnitOfWork { get; }
}

```

References – Repository Contracts

Separated Interface pattern (By Martin Fowler)

<http://www.martinfowler.com/eaaCatalog/separatedInterface.html>

Value objects

"Many objects do not have conceptual identity. These objects describe certain characteristics of a thing." [E.E.]

As shown in previous sections when drilling down on entities and aggregates, the identity is fundamental for the entities; however, there are many objects and data in a system that do not require such an identity and identity tracking.

The definition of Value Objects is: Objects that describe things; to be more accurate, an object with no conceptual identity that describes a domain aspect. In short, these are objects that we instantiate to represent design elements which only concern us temporarily. We care about what they are, not who they are. Basic examples are numbers, strings, etc. but they also exist in higher level concepts. For example, an "Address" in some systems/domains could be an entity because in that system an address is important as an identity, like in an electric power utility. But in most domain/systems, the "Address" can be simply a Value Object, a descriptive attribute of a company or person.

A Value Objects can also reference other entities. For example, in an application that generates a Route about how to get from one point to another, that route would be a Value Object (it would be a "snapshot" of points on how to go through a specific route, but this suggested route won't have an identity) even though internally it is referring to different entities (City, Roads, etc. if those were entities in that domain).

The following example shows a diagram of the Address Value Object within the Aggregate Order:

Value Object within Aggregate

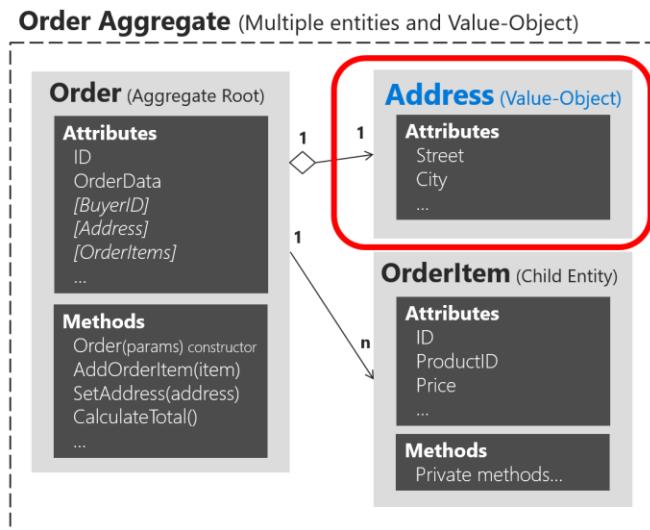


Figure 9-13. Value Object “Address” within the Order Aggregate

As shown in figure 9-13, an Entity is usually composed by multiple attributes. For example, the Order can be modeled as an Entity with an identity and composed internally by a set of attributes such as OrderId, date, Items, etc. But then, the address, which is simply a “complex value” composed by country, street, city, etc. must be modeled and treated as a Value Object.

Important characteristics of the Value Object

There are two main characteristics for the Value Objects:

- No Identity
- Immutable

The first characteristic was already introduced. In regards immutability, it is an important requirement. The values of a Value Object must be immutable once it is created. Therefore, at its construction, you must provide the required values but you must not allow them to change during the object’s lifetime.

Regarding performance, Value Objects allow you to perform certain “tricks”, thanks to their immutable nature. This is especially true in systems where there may be thousands of VALUE-OBJECT instances with many coincidences of the same values. Their immutable nature would allow us to reuse them; they would 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 another with good performance. Of course, all these recommendations 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 with basic utility methods like equality based on comparison between all the attributes (since it must not be based on identity) and

other fundamental characteristics, like the following 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, null) || left.Equals(right);
    }

    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }

    protected abstract IEnumerable<object> GetAtomicValues();

    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
        }
        ValueObject other = (ValueObject)obj;
        IEnumerator<object> thisValues = GetAtomicValues().GetEnumerator();
        IEnumerator<object> otherValues = other.GetAtomicValues().GetEnumerator();
        while (thisValues.MoveNext() && otherValues.MoveNext())
        {
            if (ReferenceEquals(thisValues.Current, null) ^ ReferenceEquals(otherValues.Current,
                null))
            {
                return false;
            }
            if (thisValues.Current != null && !thisValues.Current.Equals(otherValues.Current))
            {
                return false;
            }
        }
        return !thisValues.MoveNext() && !otherValues.MoveNext();
    }

    // Other utility methods
}
```

Then you can use it when implementing your actual Value Object, like the Address Value Object.

```
public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }
    public Address(string street, string city, string state,
                  string country, string zipcode)
    {
        Street = street;
        City = city;
        State = state;
        Country = country;
        ZipCode = zipcode;
    }

    protected override IEnumerable<object> GetAtomicValues()
```

```

    {
        yield return Street;
        yield return City;
        yield return State;
        yield return Country;
        yield return ZipCode;
    }
}

```

No Identity characteristic when using Entity Framework Core

A limitation when using EF Core is that in its current version (EF Core 1.1) you cannot use complex-types. Therefore, you must store your Value Object as an EF entity. However, you can hide its Id so you make clear that the identity is not important in the model of your value-object. The way you hide the Id is by using the ID as a "shadow property". Since that configuration is set up in the infrastructure level it will be transparent for your domain model and its infrastructure implementation could change in the future.

In *eShopOnContainers* that "hidden Id" needed by EF Core infrastructure is implemented in the following way in the DbContext level, using Fluent API, at the infrastructure project.

```

//Fluent API within the OrderingContext:DbContext at Ordering.Infrastructure project
void ConfigureAddress(EntityTypeBuilder<Address> addressConfiguration)
{
    addressConfiguration.ToTable("address", DEFAULT_SCHEMA);

    addressConfiguration.Property<int>("Id")
        .IsRequired();

    addressConfiguration.HasKey("Id");
}

```

Therefore, it is hidden from the domain model point of view and in the future, the Value Object infrastructure could also be implemented as a complex type or any other way.

References – Implementing Value-Objects in C#

Value Object pattern [Martin Fowler]

<https://martinfowler.com/bliki/ValueObject.html>

Value Object pattern [Eric Evans book]

[Value Object discussion \[Vaughn Vernon book\]](#)

Shadow Properties in EF Core

<https://docs.microsoft.com/en-us/ef/core/modeling/shadow-properties>

Complex types and/or value objects discussion for EF Core

<https://github.com/aspnet/EntityFramework/issues/246>

Base Value Object class at eShopOnContainers

<https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs>

Sample Address Value Object class at eShopOnContainers

<https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/AggregatesModel/OrderAggregate/Address.cs>

Using Enumeration classes instead of Enums

Enums are a thin language wrapper around an integral type. You should limit their use to when you are storing one value from a closed set of values. Classification based on gender (Male, Female,

Unknown), or sizes (S, M, L, XL) are good examples. Using enums for control flow, or more robust abstractions is a [code smell](#). It will lead to fragile code with many control flow statements checking values of the enum.

Instead, create enum classes that enable all the rich features of an object oriented language.

Implementing Enumeration classes

The eShopOnContainers application, within the Ordering microservice, provides a sample Enum base class implementation like the following.

```
public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }

    protected Enumeration()
    {
    }

    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }

    public override string ToString()
    {
        return Name;
    }

    public static IEnumerable<T> GetAll<T>() where T : Enumeration, new()
    {
        var type = typeof(T);
        var fields = type.GetTypeInfo().GetFields(BindingFlags.Public |
                                                    BindingFlags.Static |
                                                    BindingFlags.DeclaredOnly);

        foreach (var info in fields)
        {
            var instance = new T();
            var locatedValue = info.GetValue(instance) as T;

            if (locatedValue != null)
            {
                yield return locatedValue;
            }
        }
    }

    public override bool Equals(object obj)
    {
        var otherValue = obj as Enumeration;

        if (otherValue == null)
        {
            return false;
        }

        var typeMatches = GetType().Equals(obj.GetType());
    }
}
```

```

        var valueMatches = Id.Equals(otherValue.Id);

        return typeMatches && valueMatches;
    }

    // Other utility methods ...
}

```

Then you can use it as a type in any entity or value object like for the "CardType" enum class.

```

public class CardType : Enumeration
{
    public static CardType Amex = new CardType(1, "Amex");
    public static CardType Visa = new CardType(2, "Visa");
    public static CardType MasterCard = new CardType(3, "MasterCard");

    protected CardType() { }
    public CardType(int id, string name)
        : base(id, name)
    {
    }
    public static IEnumerable<CardType> List()
    {
        return new[] { Amex, Visa, MasterCard };
    }
    // Other util methods
}

```

References – Enumeration classes

Why Enums are dangerous for your Domain Model

<http://www.planetgeek.ch/2009/07/01/enums-are-evil/>

<https://codecraft.co/2012/10/29/how-enums-spread-disease-and-how-to-cure-it/>

Implementing Enumeration classes in .NET

<https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/>

Base Enumeration class at eShopOnContainers

<https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/SeedWork/Enumeration.cs>

Sample "CardType" enumeration class at eShopOnContainers

<https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/AggregatesModel/BuyerAggregate/CardType.cs>

Enum Alternatives in C#

<http://ardalis.com/enum-alternatives-in-c>

Designing validations in the domain model layer

From the DDD perspective, validation rules can be viewed as invariants. One of the central responsibilities of an aggregate is enforcement of 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 OrderItem object always has to have a quantity and a name. From that point of view, invariant enforcement is the responsibility of the domain entity itself (especially of the Aggregate-Root) and therefore an entity shouldn't 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 many bugs occur because objects are in a state they should never have been in. The following is a good and practical explanation from Greg Young:

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 ... 'wait if we never allowed name to become null we wouldn't have all of these tests'...

Implementing validations in the domain model layer

Validations are usually implemented in the Domain entities constructors, or within 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.

Validating conditions and throwing 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 more advanced implementations based on the previously mentioned patterns and others.

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
```

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.

Using validation attributes in the model based on Data Annotations

Another approach is to use validation attributes based on Data Annotations. Validation attributes provide a way to configure model validation, similar conceptually to validation on fields in database tables. This includes constraints such as assigning data types or required fields. Other types of validation include applying patterns to data to enforce business rules, such as a credit card number, phone number, or email address. Validation attributes make it easy to enforce requirements.

However, this approach might be too intrusive in a Domain-Driven Design Model, as it takes a dependency on `ModelState.IsValid` from `Microsoft.AspNetCore.Mvc.ModelState`, which you must call from your MVC controllers. The model validation occurs prior to each controller action being invoked, and it is the controller method's responsibility to inspect `ModelState.IsValid()` and react appropriately. The decision to use it depends on how tightly coupled you'd like your model to be with that infrastructure:

```

using System.ComponentModel.DataAnnotations;
// Other using statements ...

// Entity is a custom base class which has the Id
public class Product : Entity
{
    [Required]
    [StringLength(100)]
    public string Title { get; private set; }

    [Required]
    [Range(0, 999.99)]
    public decimal Price { get; private set; }

    [Required]
    [VintageProduct(1970)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; private set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; private set; }

    // Constructor...

    // Additional methods for entity logic and constructor...
}

```

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. Validation frameworks like Data Annotations in ASP.NET Core or any other validation frameworks like FluentValidation carry a requirement to invoke the application framework. For example, when calling the `ModelState.IsValid()` method in Data Annotations, you need to invoke ASP.NET controllers.

It can make sense to use DataAnnotations at the application layer on 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.

Validating 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 referenced articles below.

It is worth mentioning that you can also use just one of those patterns, for example validating manually with sentences of control but using the Notification pattern to be able to stack and return a list of validation errors.

Dealing with deferred validation in the domain

There are various approaches to deal with deferred validations in the domain, such as the [Implementing Domain-Driven Design book by Vaughn Vernon](#), from pages 208-215.

References – Validations in the Domain Model

Model Validation in ASP.NET Core

<https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation>

Adding Validation in ASP.NET Core

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation>

Using the Notification Pattern to replace 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>

Validation in Domain-Driven Design (DDD)

<http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/>

Domain Model Validation

<http://colinjack.blogspot.com/2008/03/domain-model-validation.html>

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 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 validation might be similar but serve a different purpose. If you're concerned about being 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.

You could also validate your commands or input DTOs in the server side code, especially if your system doesn't have a client UI application, for example, if you are only creating a public API. If you have a client application, from a UX perspective, it is best to be proactive and not allow the user to type in stuff that makes no sense.

Therefore, in the client side code you will typically be validating the ViewModels in the client app. You could also validate the client output DTOs or commands to be sent to the server 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, or a SPA web app with that validation being coded in JavaScript or TypeScript, or a mobile app coded with Xamarin and C#.

Below are a few references for various types of client apps and technologies.

References – Validation in the Client side (Presentation Layer apps)

Validation in Xamarin mobile apps

https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/

<https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/>

Validation in ASP.NET Core apps

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation>

Validation in SPA web apps (Angular 2 / TypeScript / Javascript)

<https://scotch.io/tutorials/angular-2-form-validation>

<https://angular.io/docs/ts/latest/cookbook/form-validation.html>

<http://breeze.github.io/doc-js/validation.html>

In summary, the following 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. Consider a Builder type to create the entity in a valid state. Or, consider the 'With' pattern for creating an entity that is similar to an existing 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 for this purpose.

Validation frameworks are best used in specific layers such as the presentation layer or the application/service layer, but usually not in the Domain Layer, as you would need to take a strong dependency on an infrastructure framework.

It is easier to duplicate validation logic than to keep it consistent across application layers, and in many cases having redundant validation in the client side is good, as you can be proactive.

Domain events

Use domain events to explicitly implement side effects of changes within your domain.

In other words, and using DDD lingo, 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, logically, something that happened in a particular domain and you wish other parts of the same domain could be aware and react based of that.

An important benefit from domain events is that side effects, after something happened in a domain, can be expressed explicitly instead of implicitly. For example, if you were using just Entity Framework and entities or even aggregates, if there is a change to the side effects of a use case, it will be implicit concept implemented by code after something happened. Sometimes you don't know if that side effect is part of the main operation or if it is really a side effect. When using domain events, it makes the concept explicit and part of the Ubiquitous Language; in the *eShopOnContainers* application, for example, creating an order is not just about that order, it updates or even creates a Buyer aggregate originated from the original user, because the user is not a buyer until and after he has bought. If

using domain events, we can explicitly express that domain rule based on the ubiquitous language provided by the domain/business experts.

Domain events are partially similar to messaging-style events, with one important difference. With true messaging, queuing and a service bus, a message is sent and always received 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 are currently running but you want any side effects of the domain event to occur within the same domain.

Independently of the chosen implementation, 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, since in many scenarios if this fails, the entire operation effectively never happened.

In addition, and as mentioned, 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 the image 9-14, 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.

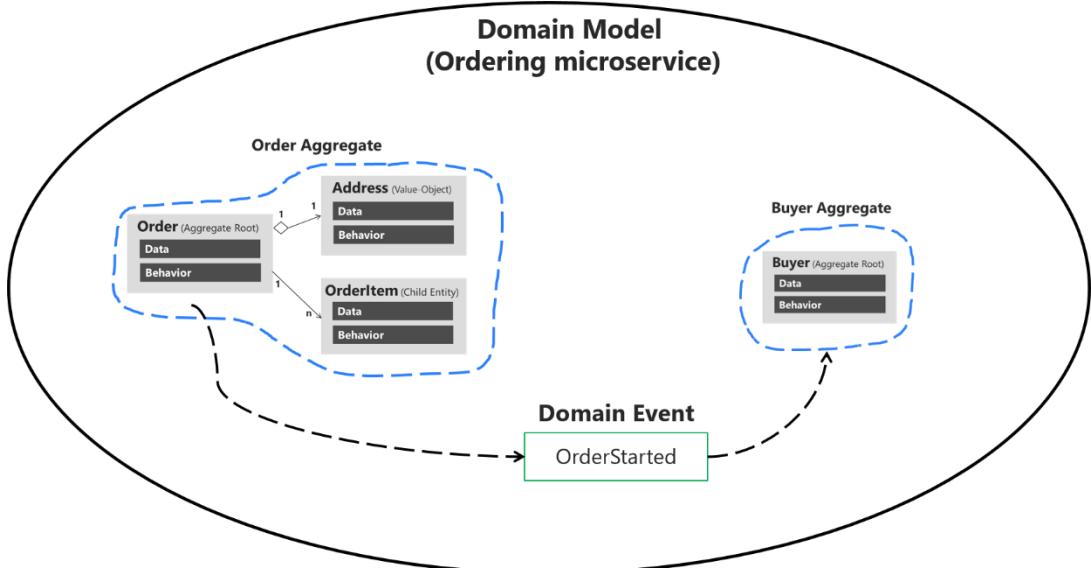


Figure 9-14. Domain Events to enforce consistency between multiple aggregates within the same domain

In the example above, the domain event "OrderStarted" might trigger a Buyer creation (if it doesn't exist) based on the original user's data when the user initiates an order. A buyer, in the ordering microservice, will be created based on the original user info from the identity microservice (info provided in the CreateOrderCommand). But the domain event is generated by the Order aggregate when it is created in the first place.

Alternately, you can also have the Aggregate Root subscribed for events raised by members of its Aggregate (child entities). For instance, each OrderItem child entity could be raising an event when the item price is higher than some amount or when the product item amount is too high, then having the aggregate root to receive those events and make any kind of global calculus or aggregation.

It is important to highlight that this event based communication is not implemented directly within the aggregates but you need to implement domain event handlers. Doing so, you could have any number of handlers triggering actions when a domain event happens.

Domain events can also be used to simply trigger an open number of application actions when that event happens. For instance, when the order is started, we might also want publish an "Integration Event" into an Event Bus and finally handled to propagate that info to other microservices or to send an email to the buyer/user saying that the order process has started. That action is not really related to any other aggregate, it is only a simple application action, but since it has to be performed **after** the transaction is committed, it is safer to use an integration event for that which are raised/published to any Event Bus only after the original transaction is performed. Therefore, this is a sample case of "connecting a Domain Event to an Integration Event" and publishing the Integration Event for "external actions" or to other microservices. Integration Events and the Event Bus are in a different subject, as introduced previously.

That "open number of actions" to be executed when a domain event happens is the key point. Eventually, the actions and rules in the domain and application will be growing. The complexity or number of actions "when something happens" will be growing and if your code is coupled with "glue", like just instantiating objects, every time you need to add a new action you will need to change the original code. At that moment, you could be introducing new bugs because with each new requirement you would need to change the original code flow which is going against the [Open/Closed Principle](#).

[principle](#) from [S.O.L.I.D.](#). Not only that, the original class that was orchestrating the operations will be growing and growing which is going 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 like in the following approach:

1. Send Command (CreateOrderCommand)
2. Command Handler
 - o Single Aggregate transaction
 - o Raise Domain Event (like OrderStarted)
3. Handle (within the current process) an open number of side effects in multiple aggregates or application actions
 - o Verify or create buyer and payment method
 - o 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
 - o Other side effects

As shown in the following image 9-15, 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.

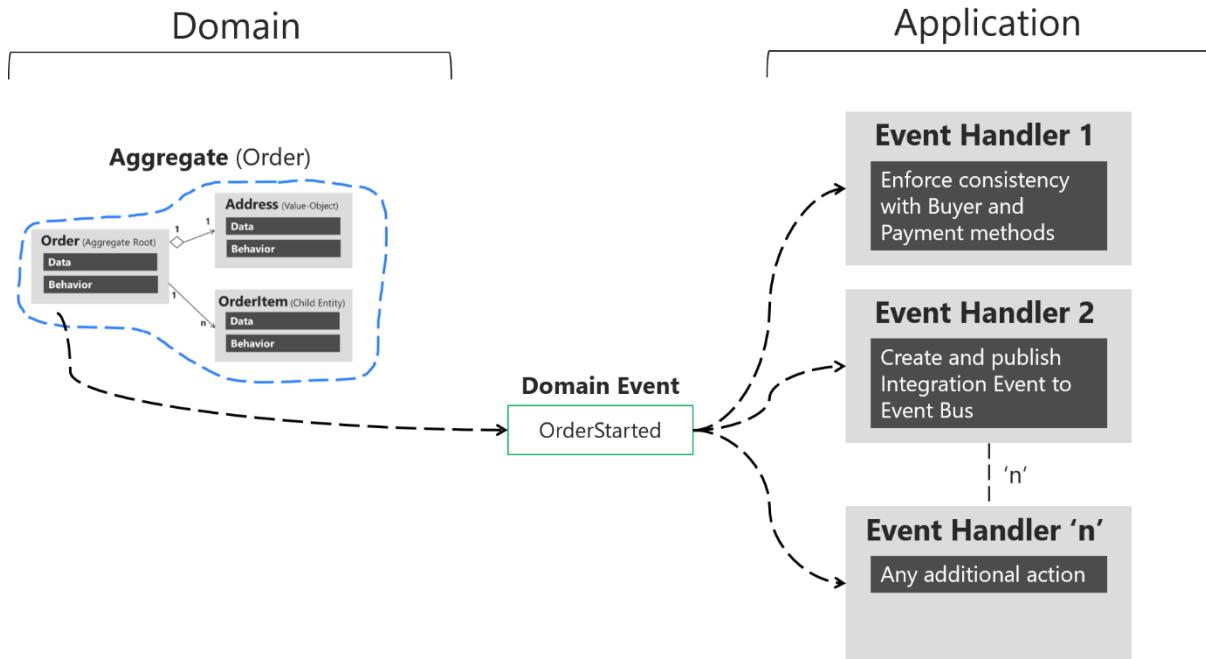


Figure 9-15. Handling multiple actions per domain

The event handlers are typically placed at the application layer as you will be using specific infrastructure objects like Repositories, or any application API for the microservice's behavior. From 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 just once. A domain event could be processed zero or 'n' times targeting multiple purposes.

Having the possibility of an open number of handlers per domain event would allow you to add many more domain rules without impacting/changing your current code. For instance, adding the following domain/business rule would be as easy as adding one or several new handlers for the following event:

"When a customer's total amount purchased in the store (including 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"

Implementing domain events

How to implement a domain event

In terms of C# code implementation, a domain event is simply a data-holding structure or class, like a DTO (Data Transfer Object) with all the information related to what just happened in the domain, like in the following code.

Regarding the ubiquitous language to be used, since an event is "*something that happened in the past*" it is very important that the class name of *the event must be represented as a verb in the past tense* such as `OrderStartedDomainEvent`, or `OrderShippedDomainEvent`. For example, the following domain event is how it is implemented in the Ordering microservice at the eShopOnContainers application.

```
public class OrderStartedDomainEvent : IAsyncNotification
{
    public int CardTypeId { get; private set; }
    public string CardNumber { get; private set; }
    public string CardSecurityNumber { get; private set; }
    public string CardHolderName { get; private set; }
    public DateTime CardExpiration { get; private set; }
    public Order Order { get; private set; }

    public OrderStartedDomainEvent(Order order,
        int cardTypeId, string cardNumber,
        string cardSecurityNumber, string cardHolderName,
        DateTime cardExpiration)
    {
        Order = order;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}
```

Basically, it is a class that holds all the data related to the `OrderStarted` event.

Events must be immutable. 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*, as you can notice in the previous code where the properties are read only from the outside of the object and the only way to update the object is through the constructor when you actually create the event object.

Raising domain events

The next question you might have is, "ok, this is cool, but how do I raise a domain event so it reaches its related event handlers?". Well, you could choose between multiple techniques or approaches for that.

Udi Dahan originally proposed in several related posts like [Domain Events – Take 2](#), to use a static class for managing and raising the events, like a static class named *DomainEvents* which would raise domain events immediately when calling the *DomainEvents.Raise(Event myEvent)*.

Jimmy Bogard also wrote a good post following a similar approach at [Strengthening your domain: Domain Events](#).

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 the side effects logic will be executed immediately right after raising the event. When you are testing and debugging you would want to focus and run just what's happening on the current aggregate classes instead of suddenly being redirected to other event handlers running side effects related to other aggregates or application logic. This is why other evolved approaches appeared, as explained in the next section.

The deferred approach for raising and dispatching events

Instead of dispatching to a domain event handler immediately, a better approach is to store/add the domain events in a collection and *right after or before* committing the transaction (like with *SaveChanges()* in EF), dispatch those domain events. That approach was neatly described also by *Jimmy Bogard* at the "[A better domain events pattern post](#)".

Deciding if you send the domain events right before or after committing the transaction is very important as depending on that you will include the side effects as part of the same transaction or in different transactions. In the last case, you would need to deal with eventual consistency implementations. This topic is precisely discussed in the next section.

The deferred approach is what the reference application [eShopOnContainers](#) uses. First, you add/store the events happening in your entities into a collection or list of events per entity. That list would be part of the entity object, better if coming from your base entity class, as shown in the code below.

```
public abstract class Entity
{
    // ...
    private List<IAsyncNotification> _domainEvents;
    public List<IAsyncNotification> DomainEvents => _domainEvents;
    public void AddDomainEvent(IAsyncNotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<IAsyncNotification>();
        _domainEvents.Add(eventItem);
    }
    public void RemoveDomainEvent(IAsyncNotification eventItem)
    {
        if (_domainEvents is null) return;
        _domainEvents.Remove(eventItem);
    }
    // ...
}
```

Thus, whenever you want to raise an event, you would just add it to the event collection like in the following code to be placed within any aggregate entity method.

```

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

this.AddDomainEvent(orderStartedDomainEvent);

```

Notice that the method "AddDomainEvent" the only thing is doing is "adding an event to the list", nothing more. It is still not reaching the event handler.

Later on, when committing the transaction into the database is when you really want to dispatch the events. If using Entity Framework Core, that means at the "SaveChanges" method level of your EF DbContext, as in the following code.

```

// EF Core DbContext
public class OrderingContext : DbContext, IUnitOfWork
{
    // ...
    public async Task<int> SaveEntitiesAsync()
    {
        // Dispatch Domain Events collection.
        // Choices:
        // A) Right BEFORE committing data (EF SaveChanges) into the DB. Will make 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. Will make 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 thought the DbContext will be committed
        var result = await base.SaveChangesAsync();
    }
}

```

With that code, you dispatch the entity events to their respective event handlers but you decouple the raising of a domain event (a simple add in memory) from dispatching to an event handler.

In addition to that, depending on what kind of dispatcher you are using, you could be dispatching the events synchronously or asynchronously.

Single transaction across aggregates versus eventual consistency across aggregates

This is an arguable topic. Many DDD authors like Eric Evans, Vaughn Vernon and others defend the rule of "1 Transaction = 1 Aggregate" and therefore, eventual consistency across aggregates, for instance:

E.E. DDD p128: *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.*

V.V. Effective Aggregate Design. Part II: Making Aggregates Work Together. p9: *...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 because in the second case the database locks amount will be pretty bad in large scale applications with a high scalability needs. Embracing the fact that high-scalable applications must not have instant transactional consistency between multiple aggregates helps accepting the concept of eventual consistency. Atomic changes are in many cases not needed by the business, and it is in any case responsibility of the domain experts to say that something really needs atomic transactions or not. Then, if some operation always needs an atomic transaction between multiple aggregates, you should at least wonder if your aggregate should be larger and was not correctly designed.

However, other developers and architects, like *Jimmy Bogard*, are okay by 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:

J.B. A better domain events pattern: ... *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 (DbContext SaveChanges()), we dispatch our events to their respective handlers.*

If you are dispatching the domain events right *before* committing the original transaction is because you want the side effects of those events to be included in the same transaction so, for instance, if the EF DbContext SaveChanges() fails the transaction will roll back all changes, including the rest of the side effect operations implemented by the related domain event handlers because the DbContext life scope is by default defined as "scoped", so the DbContext object is shared across multiple Repositories objects being instantiated within the same scope or object graph which also coincides with the HttpRequest scope when developing Web API or MVC apps.

In reality, both approaches (single atomic transaction vs. eventual consistency) can be right, it really depends on your domain/business requirements and what the domain experts tell you. Also, depending on how scalable you need it to be (more granular transactions will provoke less impact in regards database locks) and how much investment you are willing to do in your code, since eventual consistency will require a more complex code in order to detect possible inconsistencies across aggregates and the need to implement compensatory actions. Take into account that if you commit changes on the original aggregate in the first place and afterwards, when the events are being dispatched there is any issue and the events handlers cannot commit their side effects, you will have inconsistencies between aggregates.

A way to allow compensatory actions would be to store the domain events into additional database tables so it can be part of the original transaction. Afterwards, you could have batch processing detecting inconsistencies and running compensatory actions in case of issues by comparing the list of events with the current state of the aggregates.

In any case, you can choose the approach you might need, but the initial "deferred approach" implementation for raising and dispatching domain events would be pretty similar.

That is neat, but, how do you actually dispatch those events to their respective event handlers? What is that *_mediator* object that you see in the previous code? Well, that has to do with the techniques and artifacts you can use to map between events and their event handlers.

The domain event dispatcher: mapping from events to event handlers

Once you are able to dispatch or publish the events you need any kind of artifact that will publish the event so every related handler would get it and will process side effects based on that event.

One way to do it would be with a real messaging system or even an Event Bus possibly based on a Service Bus. However, that might be too much for processing domain events since you just need to process those events within the same process (same domain and application layer).

One way to map from events to multiple event handlers is by using types registration in an IoC container so you can dynamically infer where to dispatch the events. In other words, you need to know what event handlers need to get any specific event. You can see a simplified approach for that in the image 9-15.

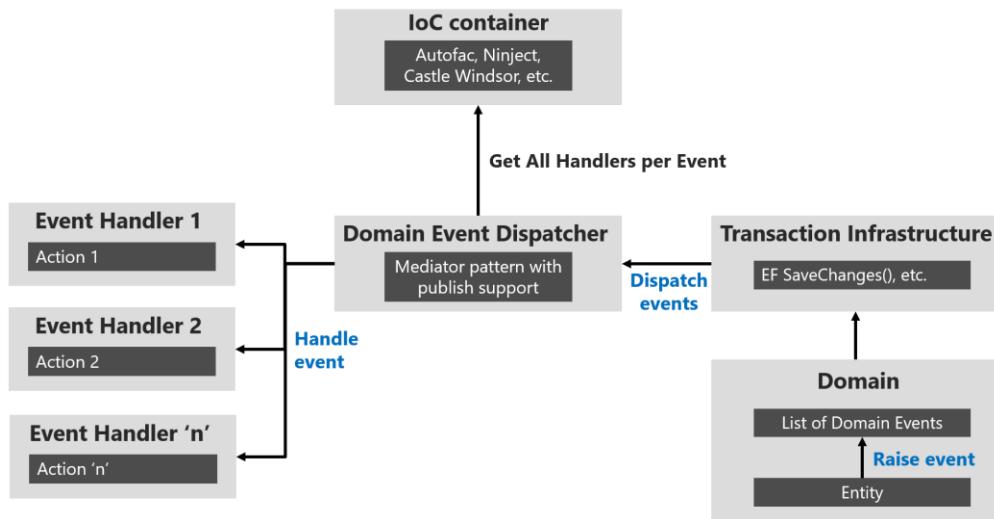


Figure 9-15. Domain Event Dispatcher

You can build all the “plumbing” and artifacts to implement that approach by yourself, however, you can also use already available libraries like [MediatR](#) which underneath uses your IoT container, so you can directly use the pre-defined interfaces and mediator’s publish/dispatch methods.

In terms of code, you first need to register the event handler types in your IoC container.

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        // Other registrations ...

        // Register the DomainEventHandler classes (they implement
        // IAsyncNotificationHandler<>) in assembly holding the Domain Events
        builder.RegisterAssemblyTypes()
            .Where(t => t.IsClosedTypeOf(typeof(IAsyncNotificationHandler<>)))
            .AsImplementedInterfaces();

        // Other registrations ...
    }
}
```

That code first identifies the assembly holding the domain event handlers based on the assembly that holds any of them. Then, since all the event handlers implement the interface `IAsyncNotificationHandler` it just searched for those types and registers all the event handlers.

How to subscribe to domain events

When using MediatR, each event handler is enforced to use an event type to be provided on the generic's parameter of the `IAsyncNotificationHandler` interface, as you can see in the following code.

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : IAsyncNotificationHandler<OrderStartedDomainEvent>
```

Based on that relationship between event and event handler (that can be considered the subscription), the mediator artifact is able to discover all the event handlers per event and trigger each of those event handlers.

How to handle domain events

Finally, the event handler will usually implement application layer code which will be using infrastructure repositories to obtain the required additional aggregates and execute side effect domain logic.

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
    : IAsyncNotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILoggerFactory _logger;
    private readonly IBuyerRepository<Buyer> _buyerRepository;
    private readonly IIdentityService _identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(ILoggerFactory logger,
    IBuyerRepository<Buyer> buyerRepository, IIdentityService identityService)
    {
        //Parameter's validations
        //...
    }

    public async Task Handle(OrderStartedDomainEvent orderStartedEvent)
    {
        var cardTypeId = (orderStartedEvent.CardTypeId != 0) ? orderStartedEvent.CardTypeId : 1;

        var userGuid = _identityService.GetUserIdentity();

        var buyer = await _buyerRepository.FindAsync(userGuid);
        bool buyerOriginallyExisted = (buyer == null) ? false : true;

        if (!buyerOriginallyExisted)
        {
            buyer = new Buyer(userGuid);
        }

        buyer.VerifyOrAddPaymentMethod(cardTypeId,
            $"Payment Method on {DateTime.UtcNow}",
            orderStartedEvent.CardNumber,
            orderStartedEvent.CardSecurityNumber,
            orderStartedEvent.CardHolderName,
            orderStartedEvent.CardExpiration,
            orderStartedEvent.Order.Id);

        var buyerUpdated = buyerOriginallyExisted ? _buyerRepository.Update(buyer) :
            _buyerRepository.Add(buyer);
    }
}
```

```

        await _buyerRepository.UnitOfWork
            .SaveEntitiesAsync();
    }
}
//Logging code using buyerUpdated info, etc.
}
}

```

The former event handler's code is considered application layer as it is using infrastructure repositories explained in the next section focusing on the infrastructure-persistence layer. Event Handlers could also use other infrastructure components.

Domain events could generate Integration events to be published outside of the microservice boundaries

Finally, is important to mention that sometimes you might want to propagate events across multiple microservices. That is considered 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.

In other words, and using DDD lingo, use domain events to explicitly implement side effects across one or multiple *aggregates*. Additionally, and for better scalability and less impact in database locks, use eventual consistency between aggregates within the same domain.

For additional information on domain events, read the following references.

References – Implementing Domain Events

What is a Domain Event? [Greg Young]

<http://codebetter.com/gregyoung/2010/04/11/what-is-a-domain-event/>

Domain Events [Jan Stenberg]

<https://www.infoq.com/news/2015/09/domain-events-consistency>

A Better Domain Events Pattern [Jimmy Bogard]

<https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/>

Effective Aggregate Design Part II: Making Aggregates Work Together [Vaughn Vernon]

http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf

Strengthening your domain: Domain Events [Jimmy Bogard]

<https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/>

Domain Events Pattern Example [Tony Truong]

<http://www.tonytruong.net/domain-events-pattern-example/>

Domain Events – Take 2 [Udi Dahan]

<http://udidahan.com/2008/08/25/domain-events-take-2/>

Domain Events – Salvation [Udi Dahan]

<http://udidahan.com/2009/06/14/domain-events-salvation/>

How to create fully encapsulated Domain Models [Udi Dahan]

<http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/>

Don't publish Domain Events, return them! [Jan Kronquist]

<https://blog.jayway.com/2013/06/20/dont-publish-domain-events-return-them/>

Domain Events vs. Integration Events in DDD and microservices architectures [Cesar de la Torre]

<https://blogs.msdn.microsoft.com/cesardelatorre/2017/02/07/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/>

Designing the infrastructure persistence layer

The data persistence components provide access to the data hosted within the boundaries of your microservice (i.e. your microservice's database). They contain the actual implementation of components such as Repositories and Unit of Work patterns that provide functionality to access data hosted within the boundaries of your microservice.

The Repository pattern

Repositories are classes/components 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 layer. If you use an ORM like Entity Framework, the code that must be implemented is highly 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 one of the well documented ways of working with a data source. Martin Fowler in his [PoEAA book](#) 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 that allows you to populate data in-memory, coming from the database in the form of the Domain Entities. This also allows you to persist updated data in the entities of the aggregate back into the database.

If you are using the CQS/CQRS architectural pattern, then most of the public methods you will have in a Repository will create/update/delete in the database from your Domain Model. You won't need any methods for queries in such a Repository.

It is important to re-emphasize that only one Repository should be defined for each Aggregate-Root. Following the goals of the aggregate-root to maintain transactional consistency between all the objects within an aggregate, you should never create a Repository for each table in the database, just one for each aggregate-root.

In a microservice based on DDD, the only channel you should use to update the database should be through 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 is okay to query the database through other channels (as you can do following a CQRS approach), because queries are idempotent and no matter how many queries you do, the database won't change. However, the transactional area, the updates, must always be controlled by the Repositories and the Aggregate-Roots.

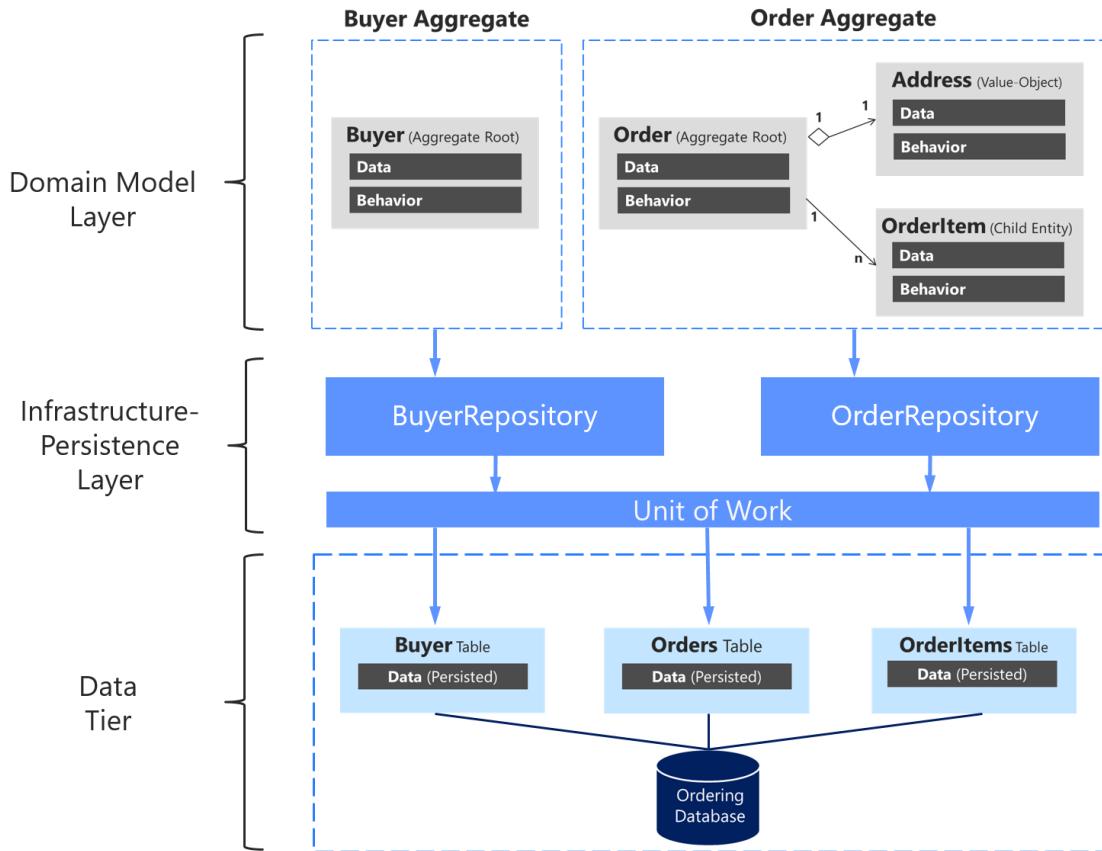


Figure X-XX. Relationship between Repositories, Aggregates and Database Tables

Enforcing 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** market interface.

Thus, each repository class implemented at the infrastructure layer implements its own contract or interface, like in the following code.

```
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
```

Going further, each specific repository interface is implementing the generic **IRepository**.

```
public interface IOrderRepository : IRepository<Order>
{
    Order Add(Order order);
    // ...

}
```

However, one way to have the code better enforce the DDD convention that each Repository should be related to a single Aggregate would be to implement a generic repository type so it is explicit that you are using a repository to target a specific aggregate. That can be easily done implementing that generic at the 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 our code, not infrastructure, so the repository abstractions will make it easier to achieve that goal.

As introduced in a previous section, it is recommended to define and place the Repository interfaces in the domain layer so the application layer (for instance, your Web API microservice) doesn't directly depend on the Infrastructure layer where you have implemented the actual Repository classes. By doing so and using Dependency Injection in the controllers of your Web API you could implement mock Repositories that would return fake hard-coded data instead of accessing the database. That decoupled approach allows you to create and run unit tests that can test just 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 a bad thing for two reasons. First, it might take a lot of time because of the large number of tests, and second, the database's records might change and impact on the results of your tests, so they might not be consistent. Testing against the database is not a Unit Test but an Integration Test. You should have many Unit Tests running fast but fewer Integration Tests against the databases.

The difference between the Repository pattern and the legacy Data Access class (DAL class)

It is important to differentiate between a Repository class and the legacy Data Access (DAL) class. A Data Access object directly performs data access and persistence operations against the storage. A repository marks the data with the operations you want to perform in the memory of a Unit of Work object (as in EF when using the DbContext), but these updates will not be performed immediately.

A Unit of Work is referred to as a single transaction that involves multiple insert/update/delete operations. In simple terms, it means that for a specific user action (for example, registration on a website), all the insert/update/delete transactions are handled in a single transaction. This is more efficient than handling multiple database transactions in a chattier way.

These multiple persistence operations will be 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 is implemented as the DbContext.

In many cases, this pattern or way of applying operations against the storage can increase the application performance and reduce the possibility of inconsistencies. Also, it reduces transaction blocking in the database tables because all the intended operations will be committed as part of one transaction. This is more efficient in comparison to executing many isolated operations against the

database. Therefore, the selected ORM will be able to optimize the execution against the database by grouping several update actions, as opposed to many small separate executions.

References – Infrastructure and Persistence patterns

The Repository pattern

<http://martinfowler.com/eaaCatalog/repository.html>
<https://msdn.microsoft.com/en-us/library/ff649690.aspx>
<http://deviq.com/repository-pattern/>

-- The Repository pattern. [By Eric Evans in his DDD book](#) --

The Unit of Work pattern

<http://martinfowler.com/eaaCatalog/unitOfWork.html>
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>

Implementing the infrastructure persistence layer with Entity Framework Core

When using 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 using .NET Core, you should also use Entity Framework Core, which runs on Windows or Linux in the same way as .NET Core. EF Core is a complete rewrite of Entity Framework, 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.

Since an introduction to EF Core is already available in Microsoft's documentation, this guidance is simply pointing to it with no further details:

References – Entity Framework Core

EF Core intro

<https://docs.microsoft.com/en-us/ef/core/>

Getting started with ASP.NET Core and Entity Framework Core

<https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/>

DbContext

<https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.dbcontext>

Compare EF Core & EF6.x

<https://docs.microsoft.com/en-us/ef/efcore-and-ef6/index>

Infrastructure in Entity Framework Core from a DDD perspective

From a Domain-Driven Design point of view, an important of EF is the ability to use POCO Domain Entities, also known as POCO Code-First entities in EF jargon. By using POCO Domain Entities, your Domain Model classes are persistence ignorant, as the [Persistence Ignorance](#) and the [Infrastructure Ignorance](#) principles state.

In addition, in EF Core 1.1 you can have plain fields in your entities instead of properties with public/private setters. If you don't want an entity field to be accessible from the outside, you just create the attribute/field. There is no need to use private setters if you prefer this cleaner approach.

In a similar way, you can now have properly encapsulated collections (like a `List<>` or `HashSet<>`) in your entities that rely on EF for persistence. Previous versions of Entity Framework required collection properties to support `ICollection<T>`, which meant any developer using the parent entity class could add or remove items from its property collections. 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.

You can use a private collection while exposing a read-only `IEnumerable`, 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 IEnumerable<OrderItem> OrderItems => _orderItems.AsReadOnly();

    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);
    }
}
```

Note that the property `OrderItems` can now only be accessed as read-only with `List<>.AsReadOnly()`. This will create a read only wrapper around the private list so it's protected

against external updates. It's much cheaper than using `.ToList()` because it won't have to copy all of the items in a new collection, just one heap alloc for the wrapper instance.

EF Core provides a way to map the domain model to the physical database without contaminating the domain model. It's 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 `OnModelCreating`, code shown below, the code in bold tells EF Core to access the `OrderItems` property through its field.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    // ...
    modelBuilder.Entity<Order>(ConfigureOrder);
    // Other entities ...
}

void ConfigureOrder(EntityTypeBuilder<Order> orderConfiguration)
{
    // Other configuration ...

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

    // Other configuration ...
}
```

When using fields instead of properties, the `OrderItem` entity is persisted just as if it had a `List<OrderItem>` property, but now it exposes a single interface (the `AddOrderItem()` method) for adding new items to the order, so behavior and data are tied together and will be consistent throughout any application code that uses the Domain Model.

Implementing 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...
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)
        {
            if (context == null)
            {
                throw new ArgumentNullException(
                    nameof(context));
            }
        }
    }
}
```

Repository contract implemented in the Domain Layer

The EF `DbContext` comes in the constructor through Dependency Injection and is shared between multiple Repositories within the same HTTP request/scope thanks to its by default lifetime (`ServiceLifetime.Scoped`) that can also be explicitly set at `services.AddDbContext<>`

```

        nameof(context));
    }

    _context = 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;
}
}

```

Annotations:

- Adds a Buyer entity to the UnitOfWork (DbContext)
- Optional query method

Methods to implement in a Repository (Updates/Transactions versus Queries)

Within each Repository class, you should place the persistence methods that update the state of entities contained by its related Aggregate. Remember there is 1:1 relationship between an Aggregate and its related Repository. Take into account that an Aggregate-Root entity object might have embedded child entities within its EF graph. For example, a Buyer might have multiple PaymentMethods related as child entities.

Since the selected approach for the Ordering microservice in the eShopOnContainers sample application 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 guidance might only have update/transactional methods but not query methods, unless you need a specific query for the transactional operations. For example, the BuyerRepository repository implements a `FindAsync()` method, because the application needs to know if a particular buyer exists before creating a new buyer related to the order. Therefore, having query methods in these repositories would be optional if using CQRS approaches and only used if needed by validations of data required for the transactions.

Using a custom repository versus using EF DbContext directly

The Entity Framework `DbContext` class is based on the UnitOfWork and Repository pattern and can be used directly from your code, for example from an ASP.NET Core MVC controller. That is the way you can create the simplest code, as in the CRUD Catalog microservice in the eShopOnContainers sample sample. So, in cases where you just want to have the simplest code possible, you might want to directly use the `DbContext` class.

However, implementing custom Repositories provides several benefits when implementing more complex microservices or applications. The repository and unit of work patterns are intended to create an abstraction layer between the infrastructure persistence layer and the application and domain

layers. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing.

Once you have implemented one repository class and repository interface per Aggregate-Root, when you get the injected instance (through DI) of the repository implementation in your controller, you are using the interface so that the controller will accept a reference to any object that implements that repository interface. When the controller runs under a web server, it receives a repository that works with the Entity Framework. When the controller runs under a unit test class, it could receive a mock repository implementation that works with fake data, probably hard-coded so it is predictable and stored in a way that you can easily manipulate for testing, such as an in-memory collection.

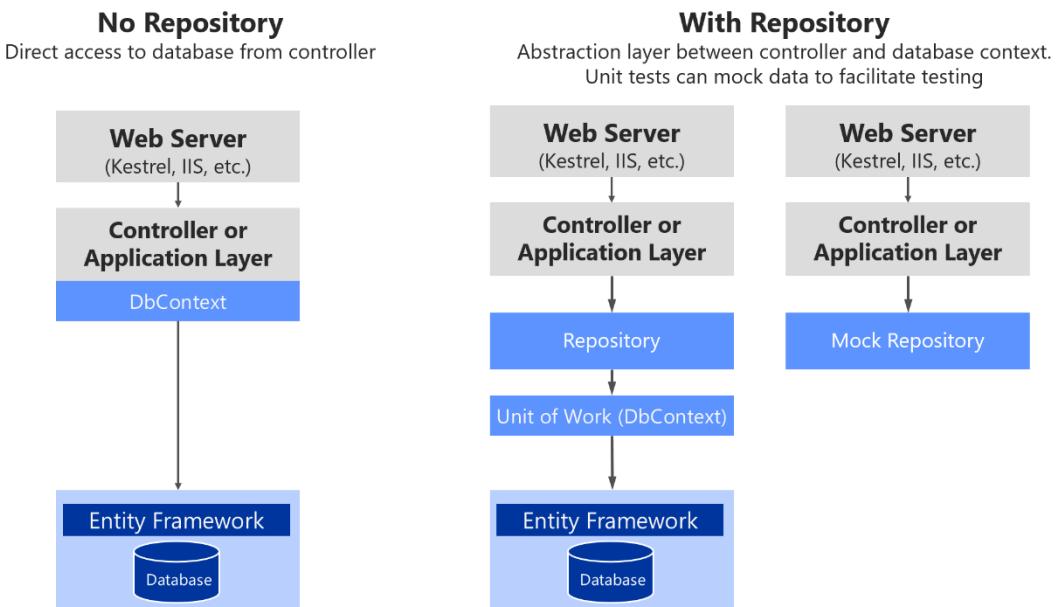


Figure X-XX. Using custom Repositories vs. plain DbContext

There are multiple alternatives when mocking. You could mock just repositories or you could also mock a whole unit of work.

Later, when focusing on the application layer, you'll 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 easier with unit tests that aren't 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 and more brittle.

If you were using **DbContext** directly, the only choice you have to run unit tests would be by using an In-memory SQL Server with predictable data for unit tests. You wouldn't be able to control mock objects and fake data in the same way.

EF DbContext and IUnitOfWork instance lifetime in your IoC container

It's important to highlight that the **DbContext** object (exposed as an **IUnitOfWork**) might need to be shared among multiple repositories within the same HTTP request scope. For example, when the operation being executed has to 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 the domain, not an EF type.

In order to do that, and as shown in the code below, the instance of the DbContext object has to be ServiceLifetime.Scoped, which is the default lifetime when registering your DbContext with services.AddDbContext<> in your IoC container, from the ConfigureServices() method of your Startup.cs file in your ASP.NET Core Web API project.

```
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc(options =>
    {
        options.Filters.Add(typeof(HttpGlobalExceptionFilter));
    }).AddControllersAsServices();

    services.AddEntityFrameworkSqlServer()
        .AddDbContext<OrderingContext>(options =>
    {
        options.UseSqlServer(Configuration["ConnectionString"],
            sqlp => sqlp.MigrationsAssembly(typeof(Startup).GetTypeInfo().
                Assembly.GetName().Name));
    },
    ServiceLifetime.Scoped // Note that Scope is the 'by default' choice
    // in AddDbContext<>. It's shown here only for
    // didactic purposes, but no need to be explicit
);
}
```

The DbContext instantiation mode shouldn't 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 memory when using the *scoped* lifetime.

```
// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
    .As<IOrderRepository>()
    .InstancePerLifetimeScope();
```

Important: Be aware that using the *singleton* lifetime could cause you serious problems when your DbContext is (by default) scoped.

References – Implementing Repositories with EF

Implementing Repositories with Entity Framework Core

<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>
<https://www.infoq.com/articles/repository-implementation-strategies>

Comparing ASP.NET Core IoC container service lifetimes with Autofac IoC container instance scopes

<https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/>

Table mapping

Table mapping identifies the table data to be queried from and saved to in the database.

Previously you saw how your domain entities (i.e. Product or Order) can be used to generate a related database schema. In EF, most of it is based on the concept of *conventions*. Conventions are topics like "*What will be the name of a table?*" or "*What property is going to be the primary key?*", and they are typically based on conventional names, for example "*a property ending with the suffix 'Id' will be the primary key*".

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 >` 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 method `OnModelCreating()` from the `DbContext` class, as shown in the code below from the Ordering microservice, part of the eShopOnContainers application.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    //Other entities
    modelBuilder.Entity<OrderStatus>(ConfigureOrderStatus);
    //Other entities
}
void ConfigureOrder(EntityTypeBuilder<Order> orderConfiguration)
{
    orderConfiguration.ToTable("orders", DEFAULT_SCHEMA);

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

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

    orderConfiguration.Property<DateTime>("OrderDate").IsRequired();
    orderConfiguration.Property<string>("Street").IsRequired();
    orderConfiguration.Property<string>("State").IsRequired();
    orderConfiguration.Property<string>("City").IsRequired();
    orderConfiguration.Property<string>("ZipCode").IsRequired();
    orderConfiguration.Property<string>("Country").IsRequired();
    orderConfiguration.Property<int>("BuyerId").IsRequired();
    orderConfiguration.Property<int>("OrderStatusId").IsRequired();
```

```

orderConfiguration.Property<int>("PaymentMethodId").IsRequired();

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 as a field
navigation SetPropertyAccessMode(PropertyAccessMode.Field);

orderConfiguration.HasOne(o => o.PaymentMethod)
    .WithMany()
    .HasForeignKey("PaymentMethodId")
    .OnDelete(DeleteBehavior.Restrict);

orderConfiguration.HasOne(o => o.Buyer)
    .WithMany()
    .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 is advisable to partition that code and have multiple sub-methods, one per entity, as shown in the code above. Going further, for particularly large models, it can even be advisable to have separate source files/static classes for configuring different entity types.

The above code is very explicit, however, EF Core conventions do most of this automatically, so the actual code you would need to write to achieve the same thing would be much smaller.

The Hi/Lo pattern in EF Core

An interesting configuration in that code is that it is using HiLo as the key generation strategy, based on the *Hi/Lo pattern*. EF Core supports HiLo with the `ForSqlServerUseSequenceHiLo` method.

The Hi/Lo pattern describes a mechanism for generating safe-ids on the client side rather than in the database. Safe in this context means without collisions. This pattern is interesting for three reasons:

- It doesn't break the Unit of Work pattern
- It doesn't need many round-trips as the Sequence generators in other DBMS.
- It generates a human readable identifier, unlike GUID techniques.

Mapping fields instead of properties

With the new feature in EF Core 1.1 to map columns to fields, it is possible to not use any 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 needs not be accessed from outside the entity.

For example, the `_someOrderInternalState` field can't have a property for either setter or getter. That field could be calculated within the order's business logic and also used by the order's methods, so it shouldn't be a property. However, it needs to be persisted in the database. In EF 1.1 there's a way to map a field (without a related property) to a column in the database.

You can do this with single fields or also with collections, like a `List<>` field.

This point was mentioned when 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.

Shadow properties and Value-Objects

Shadow properties are properties that do not exist in your entity class. The value and state of these properties are maintained purely in the Change Tracker.

Shadow property values can be obtained and changed through the ChangeTracker API.

From a DDD point of view, shadow properties are a convenient way to implement *Value-Objects* by hiding the Id as a shadow property primary key. This is important since a Value-Object shouldn't have identity or at least it is not important, as mentioned in the Domain Model Layer when shaping Value-Objects. The point here is that at the time, EF Core doesn't have any way to implement Value-Objects as Complex Types, as it is possible in EF 6.x. That's why it currently must be implemented as an entity with a hidden Id (primary key) as a shadow property.

References – Table Mapping

Table Mapping

<https://docs.microsoft.com/en-us/ef/core/modeling/relational/tables>

Use HiLo to generate keys with Entity Framework Core

<http://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/>

Backing Fields

<https://docs.microsoft.com/en-us/ef/core/modeling/backing-field>

Encapsulated Collections in Entity Framework Core

<http://ardalis.com/encapsulated-collections-in-entity-framework-core>

Shadow Properties

<https://docs.microsoft.com/en-us/ef/core/modeling/shadow-properties>

Using NoSQL databases as a persistence infrastructure

When using NoSQL databases for your infrastructure data tier, you wouldn't typically be using an ORM like Entity Framework Core. Instead you would use the API provided by the chosen NoSQL engines such as Azure Document DB, MongoDB, Cassandra, RavenDB, CouchDB, or Azure Storage Tables.

However, when using a No-SQL database, especially a Document-oriented database like Azure Document DB, CouchDB, or RavenDb, the way you design your model with DDD Aggregates is similar in regards to the identification of AggregateRoots, child entity classes, and value-object classes.

Basically, when using a document-oriented database, you would implement an Aggregate (group of Domain entities and value-objects that must keep consistency) as a single document, serialized in JSON or any other format.

The difference would be the way you persist that model. Therefore, when implementing a Domain Model, you want to have a model based on POCO entity classes, agnostic to the infrastructure persistence, so you could potentially move to a different persistence infrastructure. It wouldn't be trivial, as transactions and persistence operations will be very different, but at least you could have a clean and protected Domain Model, following the Persistence Ignorant principle.

In any case, when using NoSQL databases the entities will be more denormalized, so it's not a simple table mapping. Your domain model might have a few impacts, after all.

However, if you were modelling your Domain Model based on Aggregates, moving to NoSQL and document oriented databases might be easier, because you already defined the aggregate's boundaries which are similar to serialized documents in document-oriented databases.

For instance, the following JSON code is a sample implementation of an Order Aggregate when using a Document oriented database, similar to the order aggregate we implemented in the eShopOnContainers sample but not using EF Core underneath.

```
JSON example of the Order Aggregate when using a Document oriented DB
{
  "id": "2017001",
  "orderDate": "2/25/2017",
  "buyerId": "1234567",
  "address": [
    {
      "street": "100 One Microsoft Way",
      "city": "Redmond",
      "state": "WA",
      "zip": "98052",
      "country": "U.S."
    }
  ],
  "orderItems": [
    {"id": 20170011, "productId": "123456", "productName": ".NET T-Shirt",
     "unitPrice": 25, "units": 2, "discount": 0},
    {"id": 20170012, "productId": "123457", "productName": ".NET Mug",
     "unitPrice": 15, "units": 1, "discount": 0}
  ]
}
```

When using a C# model to implement that aggregate to be used by, for instance, the Azure Document DB SDK, it would be similar to the C# POCO classes used with EF Core. The difference will be 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 DocumentDB 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).
// This can be saved as JSON as is without converting into rows/columns.

Order orderAggregate = new Order
{
```

```

        Id = "2017001",
        OrderDate = new DateTime(2005, 7, 1),
        BuyerId = "1234567",
        PurchaseOrderNumber = "PO18009186470"
    }

    Address address = new Address
    {
        Street = "100 One Microsoft Way",
        City = "Redmond",
        State = "WA",
        Zip = "98052",
        Country = "U.S."
    }

    orderAggregate.UpdateAddress(address);

    OrderItem orderItem1 = new OrderItem
    {
        Id = 20170011,
        ProductId = "123456",
        ProductName = ".NET T-Shirt",
        UnitPrice = 25,
        Units = 2,
        Discount = 0;
    };

    OrderItem orderItem2 = new OrderItem
    {
        Id = 20170012,
        ProductId = "123457",
        ProductName = ".NET Mug",
        UnitPrice = 15,
        Units = 1,
        Discount = 0;
    };
    //Using methods with domain logic within the entity. No anemic-domain model
    orderAggregate.AddOrderItem(orderItem1);
    orderAggregate.AddOrderItem(orderItem2);
    // *** End of Domain Model Code ***
    //...

    // *** Infrastructure Code using Document DB Client API ***
    Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName,
                                                                collectionName);
    await client.CreateDocumentAsync(collectionUri, order);

    // As your app evolves, let's say your object has a new schema. You can insert OrderV2
    // objects without any changes to the database tier.
    Order2 newOrder = GetOrderV2Sample("IdForSalesOrder2");
    await client.CreateDocumentAsync(collectionUri, newOrder);

```

You can see that the way you work with your Domain Model can be similar to the way you are using it in your Domain Model Layer when the infrastructure was EF underneath. You still use the same AggregateRoot's methods to ensure consistency, invariants and validations within the aggregate.

However, when persisting your model into the NoSQL db, implemented in the infrastructure and persistence layer, this is where the code and API will dramatically change internally.

References – NoSQL Databases

Azure Document DB

<https://docs.microsoft.com/en-us/azure/documentdb/documentdb-modeling-data>

DDD Aggregate storage

<https://vaughnvernon.co/?p=942>

Event storage

<https://github.com/NEventStore/NEventStore>

Designing the microservice's application layer and Web API

Using S.O.L.I.D. principles and Dependency Injection

The S.O.L.I.D. principles and Dependency Injection (DI) are critical techniques to be used in any modern and mission-critical application, such as developing a microservice with DDD patterns. However, you should also use DI and apply the S.O.L.I.D. principles even when you aren't using DDD approaches or patterns.

S.O.L.I.D. is an acronym that groups five fundamental principles:

- Single Responsibility Principle
- Open/close principle
- Liskov substitution principle
- Inversion Segregation principle
- Dependency Inversion principle

S.O.L.I.D. and DI tackle more about how you design your application/microservice internal layers and decoupled dependencies between them, so this is not related to the Domain but related to the application's technical design. But, DI allows you to decouple the infrastructure layer from the rest of the layers allowing a better decoupled implementation of the DDD layers.

Dependency injection (DI) is a technique for achieving loose coupling between objects and their dependencies. Rather than directly instantiating collaborators, or using static references, the objects a class needs 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. DI 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 S.O.L.I.D. Principles, your classes will naturally tend to be small, well-factored, and easily tested. What if you find that your classes tend to have way too many dependencies being injected? Using DI through the constructor it will be easy to detect by just taking a look at the number of parameters of your constructor. If there are too many dependencies, this is generally a sign ([code smell](#)) that your class is trying to do too much, and is probably violating SRP - the Single Responsibility Principle.

There is much to be said about S.O.L.I.D. and DI. It would really take another guide/book to cover it in detail, so this guide requires the reader to have a minimum knowledge or skills with these topics.

References – S.O.L.I.D. principles and Dependency Injection

S.O.L.I.D. principles

<http://devg.com/solid/>

Dependency Injection

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

New is Glue

<http://ardalis.com/new-is-glue>

Implementing the microservice's application layer and Web API

Using Dependency Injection to inject infrastructure objects into your application layer

The application layer, as mentioned previously, is whatever artifact you are building. In the case of a microservice built with ASP.NET Core, the application layer will usually be your Web API library. If you'd like to separate what is coming from ASP.NET Core (its infrastructure plus your controllers) from your custom application layer code, that could also be placed in a separate library.

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's container refers to the types it manages as services. You configure the built-in container's services in the `ConfigureServices` method in your application's `Startup` class.

Typically, you'd want to inject dependencies that implement infrastructure objects. The most typical dependencies to inject are Repositories, or for simpler implementations you could directly inject your Unit of Work pattern object (the EF `DbContext` object), as they are the implementation of your infrastructure persistence objects.

In the following example, you can see how .NET Core is injecting the needed Repository objects.

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;

    public CreateOrderCommandHandler(IOrderRepository orderRepository)
    {
        if (orderRepository == null)
        {
            throw new ArgumentNullException(nameof(orderRepository));
        }

        _orderRepository = orderRepository;
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
```

```

// ...
// ... Additional code
//

// Create the Order AggregateRoot
// Add child entities and value-objects through the Order Aggregate-Root
// methods and constructor so validations, invariants and business logic
// make sure that consistency is preserved across the whole aggregate

var address = new Address(message.Street, message.City, message.State,
    message.Country, message.ZipCode);
var order = new Order(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);
}

//Persist the Order through the Repository
_orderRepository.Add(order);

var result = await _orderRepository.UnitOfWork
    .SaveEntitiesAsync();

return result > 0;
}
}

```

Finally, it is using the injected repositories to execute the transaction and persist the state changes.

Registering the Dependency implementation types and interfaces/abstractions

You also need to know where to register the Interfaces and classes that will be injected to your objects through DI based on the constructors.

Using the built-in IoC container provided by ASP.NET Core

When using the built-in IoC container provided by ASP.NET Core (as in the simple Catalog microservice in the eShopOncontainers sample), you register the types in the ConfigureServices() method in the MVC Startup.cs file.

```

// Registration of types into ASP.NET Core built-in container
public void ConfigureServices(IServiceCollection services)
{
    // Register out-of-the-box framework services.
    services.AddDbContext<CatalogContext>(c =>
    {
        c.UseSqlServer(Configuration["ConnectionString"]);
    },
    ServiceLifetime.Scoped
);

services.AddMvc();

// Register custom application dependencies.

```

```
        services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();  
    }  
}
```

In this example, the last line of code 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.

Using Autofac as IoC container

You can also use additional IoC containers and plug them to the ASP.NET Core pipeline, as in the Ordering microservice in the eShopOnContainers sample 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 application module for one class library with the implemented custom types.

```
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<IOOrderQueries>()  
            .InstancePerLifetimeScope();  
  
        builder.RegisterType<BuyerRepository>()  
            .As<IBuyerRepository>()  
            .InstancePerLifetimeScope();  
  
        builder.RegisterType<OrderRepository>()  
            .As<IOOrderRepository>()  
            .InstancePerLifetimeScope();  
    }  
}
```

In the code above, the abstraction `IOOrderRepository` is registered along with the implementation class `OrderRepository`, which means that whenever a constructor is declaring a dependency through the abstraction or interface `IOOrderRepository`, 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 a single instance per LifetimeScope (referred to in ASP.NET Core as "scoped"), a new instance per dependency

(referred to in ASP.NET Core as “transient”), or a single instance shared across all objects using the IoC container (referred to in ASP.NET Core as “singleton”).

For additional information about DI, lifetime scopes and usage in ASP.NET Core, read the following references.

References – ASP.NET Core DI and Autofac

Using Dependency Injection in ASP.NET Core and .NET Core

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>

Autofac

<http://docs.autofac.org/en/latest/getting-started/index.html>

<http://docs.autofac.org/en/latest/lifetime/instance-scope.html>

Comparing lifetime scopes between ASP.NET Core built-in container and Autofac

<https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/>

Implementing the Command and Command-Handlers patterns

In the DI through constructor example shown in the previous section, the IoC container was injecting Repositories through a constructor, but exactly where were they injected? In a very simple Web API (for example, the Catalog microservice in the eShopOnContainers sample), you would injecting them at the MVC Controllers level, in a Controller constructor. However, in the previous example it is done at a CommandHandler level, so let's explain what a ComamndHandler is and why you would want to use it.

The Command pattern is intrinsically related to the CQRS pattern that was previously introduced in this guide. CQRS has two sides. The queries (previously explained using in this approach for simplified queries with [Dapper](#) Micro ORM) and the Commands as the starting point for the transactions/writes.

Remember, CQRS is not an architecture, it's a pattern which you can use in some microservices of your application architecture, or in all of them. You decide if you implement CQRS per Bounded Context or microservice. not as the top-level architecture for your whole application.

As shown in the high-level diagram below, the pattern is based on accepting commands from the clisnt side and process those commands based on the Domain Model rules and finally persisting the states with transactions.

High level “Writes-side” in CQRS

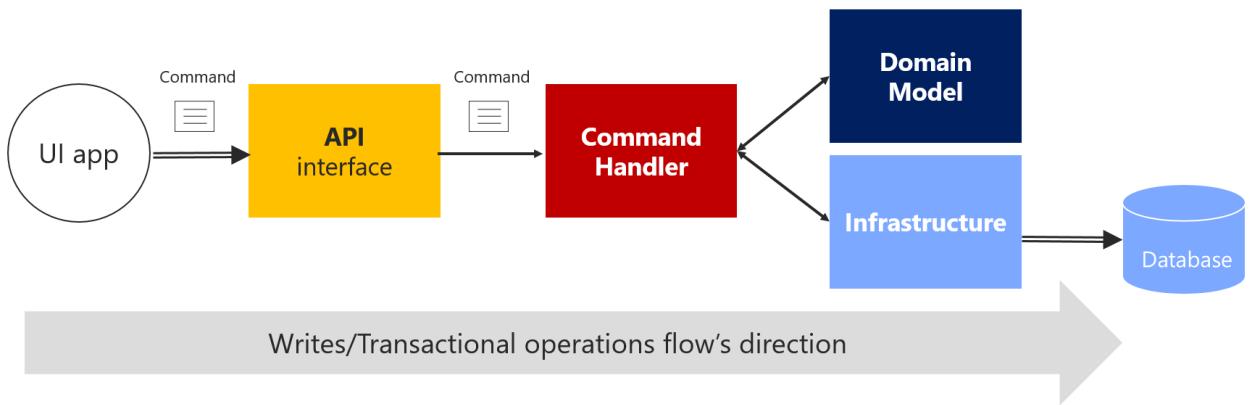


Figure X-XX. High level “Writes side” in a CQRS pattern

The Command

What is a command? – A command is a request for the system to perform an action that changes the state of the system. Since Commands are imperatives, they are typically named with a verb in the imperative tense and may include the aggregate type, for example *CreateOrderCommand*. Unlike an event, a command is not a fact from the past; it's only a request, and thus may be refused.

Commands can originate from either the user interface (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.

Another very important characteristic of a command is that *a command must be processed just once* by a single recipient. This is because commands might not be idempotent, therefore it's important that they be processed only once. For example, the same Order creation request shouldn't be processed more than once. This is a very important difference when comparing commands versus events. Usually you will want to process an event (something that happened in the past) multiple times, as many systems might be interested in that event.

Idempotency. Idempotency is a characteristic of an operation that means the operation can be applied multiple times without changing the result. For example, the operation "set the value x to ten" is idempotent, while the operation "add one to the value of x" is not. A Command is idempotent if it can be executed multiple times without changing the result, either because of the nature of the Command itself, or because of the way the system handles the Command.

Therefore, it is a good practice to make your commands and updates idempotent, when it makes sense. If for any reason (retry logic, hacking, etc.) the same *CreateOrder* command reaches your system multiple times, you should be able to identify it, insuring that you don't create multiple orders based on the same original *CreateOrder* command. To do so, you need to attach some kind of identity in the operations and identify whether that same command or update was already processed.

You send a command, you don't publish a command. Publishing is reserved for events which state a fact – that something has happened, and that the publisher has no concern about what receivers of that event do with it. But events are a different story related to Domain events and Integration events.

How then do you implement a Command? It's quite simple; a Command is implemented with a class that contains data fields or collections with all the information you need to execute that command. So, yes, a command is like a special kind of DTO (Data Transfer Object) used to request changes or transactions. The command itself is based on exactly what information is needed to process the command, and nothing more.

The following is an example of the simplified CreateOrderCommand, which is an immutable command, used in the Ordering microservice in the eShopOnContainers sample.

```
// 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://msdn.microsoft.com/en-us/library/bb383979.aspx

[DataContract]
public class CreateOrderCommand
    :IAsyncRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;
    [DataMember]
    public string City { get; private set; }
    [DataMember]
    public string Street { get; private set; }
    [DataMember]
    public string State { get; private set; }
    [DataMember]
    public string Country { get; private set; }
    [DataMember]
    public string ZipCode { get; private set; }
    [DataMember]
    public string CardNumber { get; private set; }
    [DataMember]
    public string CardHolderName { get; private set; }
    [DataMember]
    public DateTime CardExpiration { get; private set; }
    [DataMember]
    public string CardSecurityNumber { get; private set; }
    [DataMember]
    public int CardTypeId { get; private set; }
    [DataMember]
    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;

    public void AddOrderItem(OrderItemDTO item)
    {
        _orderItems.Add(item);
    }
    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }

    public CreateOrderCommand(string city, string street, string state, string country, string
    zipcode,
        string cardNumber, string cardHolderName, DateTime cardExpiration,
```

```

        string cardSecurityNumber, int cardTypeId) : this()
    {
        City = city;
        Street = street;
        State = state;
        Country = country;
        ZipCode = zipcode;
        CardNumber = cardNumber;
        CardHolderName = cardHolderName;
        CardSecurityNumber = cardSecurityNumber;
        CardTypeId = cardTypeId;
        CardExpiration = cardExpiration;
    }
    public class OrderItemDTO
    {
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public decimal Discount { get; set; }
        public int Units { get; set; }
        public string PictureUrl { get; set; }
    }
}

```

Basically, the Command class contains all the data you will need to perform 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 OO sense.

As an additional characteristic, *commands are immutable* because their expected usage is to be processed directly by the domain model. Usually, they do not need to change during their projected lifetime. The same happens with Events, but that is a different story.

In a C# class, immutability can be achieved by not having any setters, or other methods which change internal state. This immutability and lack of setters is an improvement in the Command's design, but it is not critical.

An example is a "Create an order" command. In this case, the Command class might be similar in terms of data to the Order you want to create, but you probably don't need the same attributes. For instance, the CreateOrderCommand doesn't have an Order Id because it hasn't been created yet.

Many other Command classes can be very simple, requiring only a few fields about some state that needs to be changed. For example, that would be the case if you are just changing the status of an Order from "InProcess" to "Paid" or "Shipped" status by using a command similar to the following:

```

[DataContract]
public class UpdateOrderStatusCommand
:IAsyncRequest<bool>
{
    [DataMember]
    public string Status { get; private set; }
    [DataMember]
    public string OrderId { get; private set; }
    [DataMember]
    public string BuyerIdentityGuid { get; private set; }
}

```

The Command-Handler class

The Command class example is pretty obvious. But where do you actually use that command object and provide the needed data to the Domain objects? In a Web API controller? In an Application Layer Service?

It turns out that it is pretty convenient to have a specific Command Handler class per Command. That is how the pattern works and it is precisely where you will 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 DDD.

A command handler receives a command and brokers a result from the appropriate aggregate. A result is either a successful application of the command, or an exception.

The command handler usually performs the following tasks:

- It receives the Command instance (from the mediator or any other infrastructure).
- It validates that the Command is a valid Command (if not validated by the mediator).
- It locates the aggregate instance that is the target of the Command.
- It invokes the appropriate method on the aggregate instance passing in any parameter from the command.
- It persists the new state of the aggregate to storage, which is the actual transaction.

The important point here is that all the domain logic in processing the command should be inside the domain model (the aggregates), fully encapsulated and unit-testable. The command-handler just acts as a way to get the domain model out of the persistent store and tell the infrastructure layer (Repositories) to persist the changes when the model is ready. The advantage of this approach is that you can now refactor the domain logic in a fully encapsulated, behavioral domain model without changing anything else in the application plumbing level (Web API, etc.).

When command handlers get complex with too much logic, review them and just push the behavior down to the domain objects (aggregate-root's and child entity's) methods as needed by refactoring them.

As an example of a Command-Handler class, the following code shows the same CreateOrderCommandHandler class that you saw earlier. In this case you can see highlighted the actual Handle() method and the operations with the Domain model objects/aggregates.

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IBuyerRepository _buyerRepository;
    private readonly IOrderRepository _orderRepository;

    public CreateOrderCommandHandler(IBuyerRepository buyerRepository,
                                    IOrderRepository orderRepository)
    {
        if (buyerRepository == null)
        {
            throw new ArgumentNullException(nameof(buyerRepository));
        }

        if (orderRepository == null)
```

```

    {
        throw new ArgumentNullException(nameof(orderRepository));
    }

    _buyerRepository = buyerRepository;
    _orderRepository = orderRepository;
}
public async Task<bool> Handle(CreateOrderCommand message)
{
    //
    // ... Additional code
    //

    // Create the Order AggregateRoot
    // Add child entities and value-objects through the Order Aggregate-Root
    // methods and constructor so validations, invariants and business logic
    // make sure that consistency is preserved across the whole aggregate

    var order = new Order(buyer.Id, payment.Id,
                          new Address(message.Street,
                                      message.City, message.State,
                                      message.Country, message.ZipCode));

    foreach (var item in message.OrderItems)
    {
        order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                           item.Discount, item.PictureUrl, item.Units);
    }

    //Persist the Order through the Aggregate's Repository
    _orderRepository.Add(order);

    var result = await _orderRepository.UnitOfWork.SaveChangesAsync();
    return result > 0;
}
}

```

This is the common sequence of steps a command handler might follow:

- Validate the command's incoming data.
- Use the command's data to operate with the aggregate root's methods and behavior.
- Internally within the Domain objects, Domain events could be raised while the transaction is executed, but that is transparent from a Command Handler point of view.
- If the aggregate's operation result is successful, integration events can be raised either from the infrastructure classes like Repositories or from the Command-Handler itself, after the transaction is finished.

References – Command and Command-Handler

At the Boundaries, Applications are Not Object-Oriented (by Mark Seemann)

<http://blog.ploeh.dk/2011/05/31/AttheBoundaries,ApplicationsareNotObject-Oriented/>

The Command pattern

<http://cqrs.nu/Faq/commands-and-events>

The Command-Handler pattern

<http://cqrs.nu/Faq/command-handlers>

The Command's process pipeline: how to trigger a command handler

The next question is, where do I call a Command-Handler? – You could manually call it from each related ASP.NET Core controller, however, that approach would be too coupled and not ideal.

The other two main options, which are the recommended options, are:

- Through an in-memory Mediator pattern artifact.
- With an asynchronous queue, in between controllers and handlers.

Using the mediator pattern (in-memory) in the Command's pipeline

As shown in figure X-XX, 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/DTO being received. The small single black arrows between components mean the dependencies between objects (in many cases, injected through DI) with their related interactions.

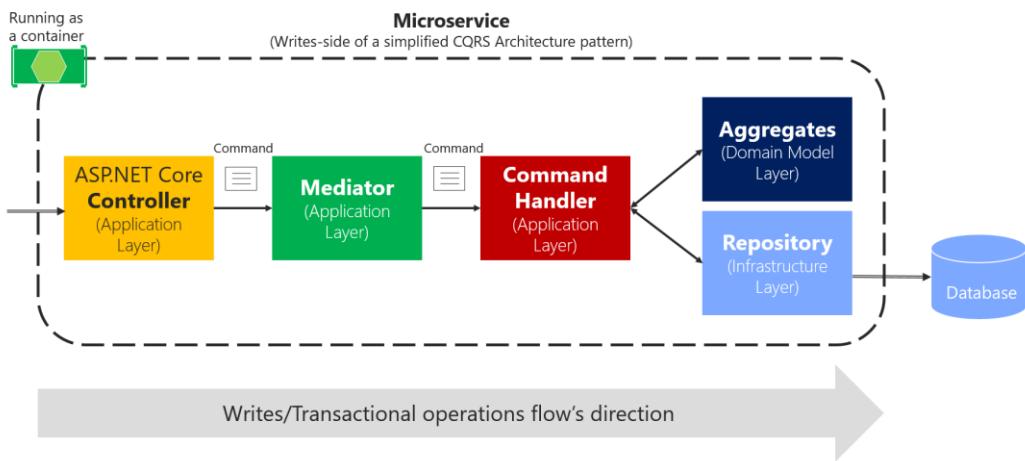


Figure X-XX. Using the Mediator pattern in CQRS microservice

The reason that using a mediator pattern makes sense is because in enterprise applications the processing requests can get increasingly complicated. You will want to be able to add an open number of cross-cutting concerns like logging, validations, transactions, 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” and coordinates execution based on state, the way it’s invoked, or the payload you provide to it.

Basically, with a Mediator component you can apply those mentioned cross-cutting concerns in a centralized and transparent way by just applying *decorators*. See the [decorator pattern](#).

Decorators are similar to [Aspect Oriented Programming – AOP](#), only applied to a specific process-pipeline managed by the mediator component. Aspects in AOP implementing cross-cutting concerns are magically applied based on *aspect weavers* injected in compilation time or based on object call interception. Both typical AOP approaches are like magic and when dealing with serious issues or bugs can be difficult to debug. On the other hand, these decorators are explicit and applied only in

the context of the mediator, so debugging is much more predictable and easy to do for any developer.

Using message queues (out-of-proc) in the Command's pipeline

Another choice is to use message queues, as shown in the image X-XX. That option could also be combined with the mediator component right before the command-handlers.

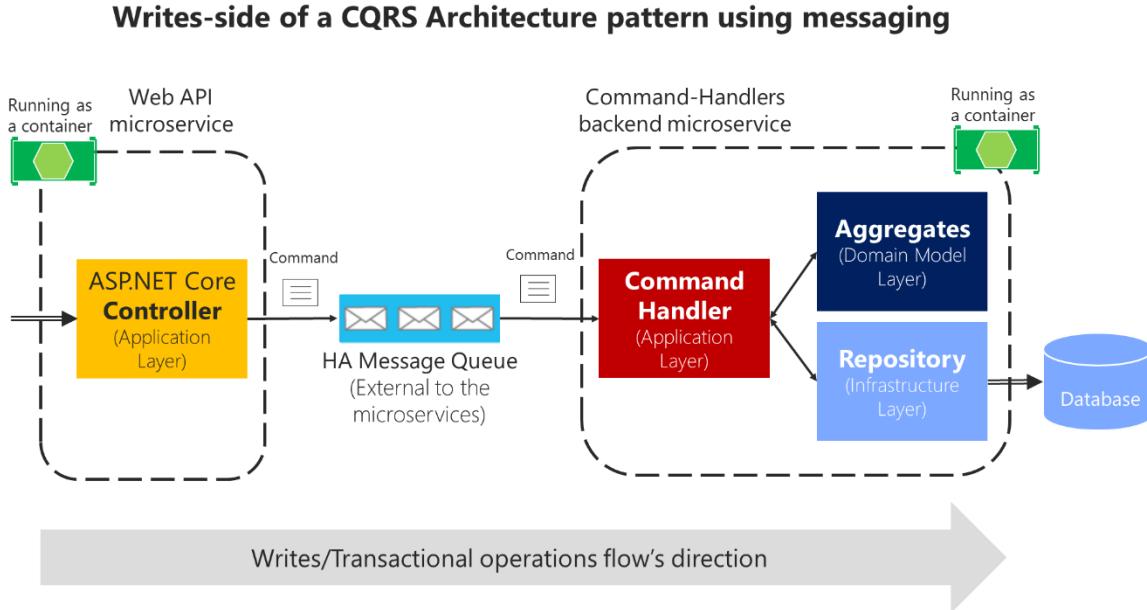


Figure X-XX. Using Message Queues with CQRS Commands

Using message queues to accept the commands can further complicate your command's pipeline, as you will probably need to split the pipeline in two processes connected through the external message queue. Still it should be used if you need to have better resiliency when submitting the command messages, plus providing better scalability and better performance because you can implement asynchronous messaging. Consider that in this case the controller just posts the command message into the queue and returns. Then, the command-handlers will be processing the messages at their own pace. That is a great benefit typical of queues, as the message queue can act as a buffer in cases when hyper scalability is needed for ingress data, for example for stocks or any other scenario with a high volume of ingress data.

However, because of the asynchronous nature of message queues, you will 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 explained by Greg Young in the following extracts:

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.

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, it's you telling me something has been done. This seems like a slight difference at first, but it has many implications.

- Greg Young -

In the eShopOnContainers implementation it was chosen to use synchronous command processing driven by the Mediator pattern, as that easily allows you to return the success or failure of the process.

In any case, this should be a decision based on your application's or microservice's business requirements. Sometimes a command might not need any confirmation and then it would be a lot simpler to implement it as a asynchronous command.

Implementing the Command process pipeline with a mediator pattern (MediatR)

As a sample implementation, this guidance is proposing the in-process pipeline based on the mediator pattern driving the commands ingestion and routing them, in memory, to the right command-handlers, plus applying decorators to separate cross-cutting concerns.

For implementation in .NET Core, there are multiple open source libraries available implementing the mediator pattern. The chosen library used in this guidance is the *MediatR* open source library (created by Jimmy Bogard), but you could use any other approach. MediatR is a small, simple in-process messaging library that allows you to process messages like a Command, while applying decorators.

MediatR is also capable of using synchronous or asynchronous execution which is important depending on your desired application behavior.

Basically, using the mediator pattern helps you to reduce coupling and isolate the concerns of the requested work to be done while automatically connecting to the handler that performs that work (the Command-Handler, in this case).

First, let's take a look to the controller's code where you actually would use the mediator object.

The constructor of your controller can be a lot simpler with just a few dependencies instead of many dependencies that you would have if you had one per cross-cutting operation.

For instance, instead of a messy constructor with many cross-cutting dependencies, you can have a clean constructor like this:

```
public class OrdersController : Controller
```

```
{
    public OrdersController(IMediator mediator,
                           IOrderQueries orderQueries)
```

You can see that it provides a very clean and lean Web API controller. Within the controller's methods, the code is also pretty simple, basically just one line sending a Command to the mediator object:

```
[Route("new")]
[HttpPost]
public async Task<IActionResult> CreateOrder([FromBody]CreateOrderCommand
                                               createOrderCommand)
{
    var result = await _mediator.SendAsync(createOrderCommand);
    if (result)
    {
        return Ok();
    }
    return BadRequest();
}
```

In order for Mediator to be aware of your command-handler classes, you need first to wire it up by registering the mediator classes and the command-handler classes in your IoC container.

By default, Mediator 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 those types, 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();

        builder.RegisterAssemblyTypes(typeof(CreateOrderCommand).GetTypeInfo().Assembly)
            .As(o => o.GetInterfaces()
                .Where(i => i.IsClosedTypeOf(typeof(IAsyncRequestHandler<,>)))
                .Select(i => new KeyedService("IAsyncRequestHandler", i)));

        builder.RegisterType(typeof(LogDecorator<,>),
                           typeof(IAsyncRequestHandler<,>),
                           "IAsyncRequestHandler");
        //Other types registration
    }
}
```

Because each Command Handler is implementing the interface with generics `IAsyncRequestHandler<T>`, then by inspecting the `RegisteredAssemblyTypes` it is able to relate each Command with its Command-Handler, because that relationship is stated in the `CommandHandler` class, as in the following example:

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
```

```
{
```

This is the code that closes the loop and correlates Commands with CommandHandlers. The handler is just a simple class, but it inherits from RequestHandler<T> and MediatR makes sure it gets invoked with the correct payload.

Applying cross-cutting concerns when processing commands with the Mediator and Decorator patterns

There's one more thing: the capability of being able to apply cross-cutting concerns to the mediator pipeline. In the Autofac registration module code you can also see at the end of that code how it is registering a decorator type, specifically, a custom Log Decorator.

That LogDecorator class can be implemented as the following simple code which is simply logging info about the command handler being executed and whether it was successful or not.

```
public class LogDecorator<TRequest, TResponse>
    : IAsyncRequestHandler<TRequest, TResponse>
    where TRequest : IAsyncRequest<TResponse>
{
    private readonly IAsyncRequestHandler<TRequest, TResponse> _inner;
    private readonly ILogger<LogDecorator<TRequest, TResponse>> _logger;

    public LogDecorator(
        IAsyncRequestHandler<TRequest, TResponse> inner,
        ILogger<LogDecorator<TRequest, TResponse>> logger)
    {
        _inner = inner;
        _logger = logger;
    }
    public async Task<TResponse> Handle(TRequest message)
    {
        _logger.LogInformation($"Executing command {_inner.GetType().FullName}");

        var response = await _inner.Handle(message);

        _logger.LogInformation($"Succeeded executing command {_inner.GetType().FullName}");
        return response;
    }
}
```

Just by implementing this decorator class and by decorating my pipeline with it, all the commands processed through MediatR will be logging information about it.

In a similar way, you could implement other decorators like a validator decorator, transaction decorator, or any other aspect or cross-cutting concern you would like to apply to commands when handling them.

For additional information on the Mediator pattern and the MediatR library, see the following references.

References – Mediator

The mediator pattern

https://en.wikipedia.org/wiki/Mediator_pattern

The decorator pattern

https://en.wikipedia.org/wiki/Decorator_pattern

MediatR

<https://github.com/jbogard/MediatR>

<https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/>

<https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/>

<https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/>

<https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/>

<https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/>

<https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/>

<https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/>

FluentValidation

<https://github.com/JeremySkinner/FluentValidation>

Why sagas?

A saga is a technique that can be used to handle out of order messages. It is similar (but not equal) to a process manager or a workflow, and typically means a long-running business process that could be implemented either with custom code or based on a service bus.

When designing processes with more than one remote call it is usually recommended to use sagas. The length of time is not important in many cases. Sometimes “a single second means a lifetime” and you might need a saga, as well.

Sagas and long running processes**A Saga on Sagas**

<https://msdn.microsoft.com/en-us/library/jj591569.aspx>

Saga implementation patterns – variations

<https://lostechies.com/jimmybogard/2013/03/21/saga-implementation-patterns-variations/>

Saga definition and implementing a saga with NServiceBus

<https://docs.particular.net/nservicebus/sagas/>

Implementing Resilient Applications

Vision

Your microservice and cloud based applications need to embrace partial failure that will certainly happen, eventually. You need to design your application so it will be resilient to those partial failures.

Resiliency is the ability to recover from failures and continue to function. It's not 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 that will happen. Therefore, your application should be resilient. It should be designed to cope with partial failures, like network outages, nodes/VMs crashing in the cloud or even simply microservices/containers being moved to a different node within a cluster that might 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.

Handling partial failure

In distributed systems like microservices-based applications, there is the ever-present risk of partial failure, for instance, a single microservice/container failing or not being available to respond for a short time or a single VM or server crashing. 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 extremely 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 uses synchronous RPCs with no timeout), it would block threads indefinitely waiting for a response. In addition to creating a bad user experience, every unresponsive wait consumes or blocks a thread, which 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 10-1.

Partial failures

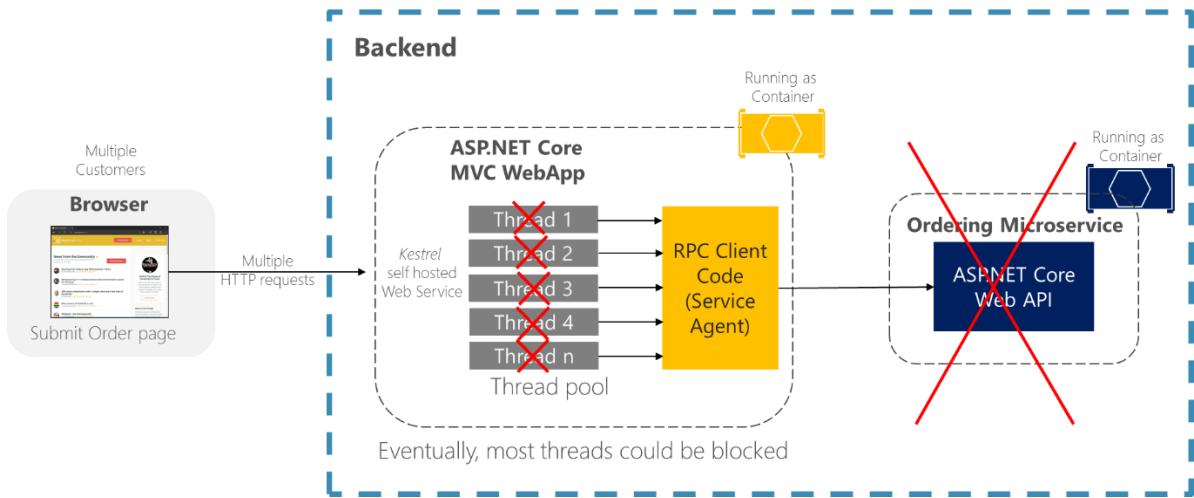


Figure 10-1. Partial failures because of dependencies impacting the service's thread availability

In a large microservice-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. These 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, if your system has a wrong design based on long chains of synchronous Http calls, as shown in figure 10-2, especially on the dependency #3.

Multiple distributed dependencies

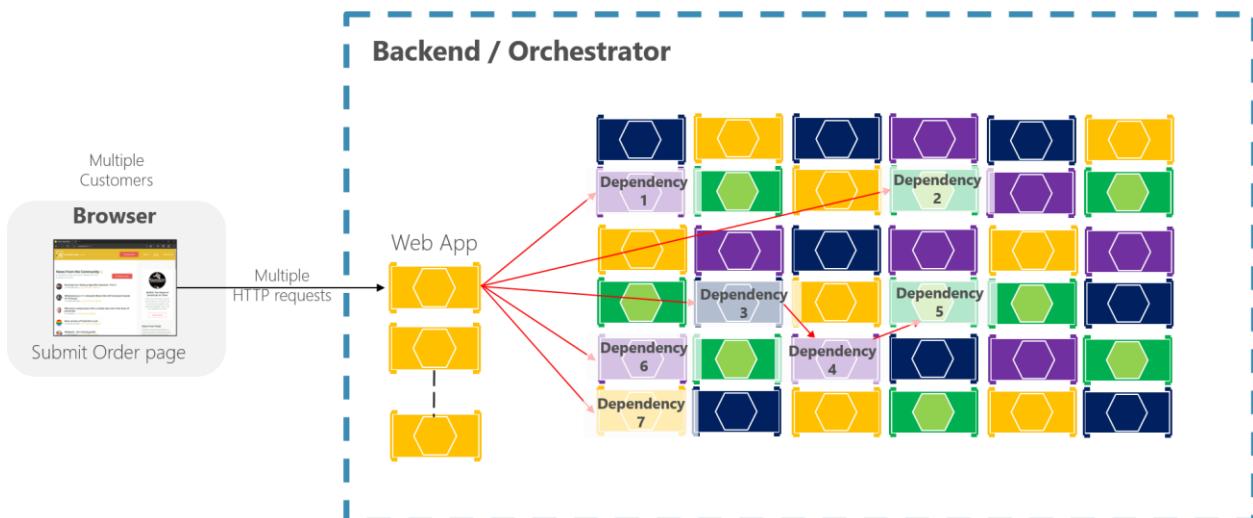


Figure 10-2. Huge impact when having a wrong design with long chains of Http requests

Intermittent failure is virtually guaranteed in a distributed and cloud based system, even if every dependency itself has excellent availability. This should be a fact you need to consider.

If you do not design and implement techniques to ensure fault tolerance, as an example, 50 dependencies each with 99.99% of availability would result in several hours of downtime each month because of that ripple effect.

When a microservice dependency fails while handling a high volume of requests , that can saturate pretty fast all available request threads in each service and crash the whole application.

Partial Failure Amplified in Microservices

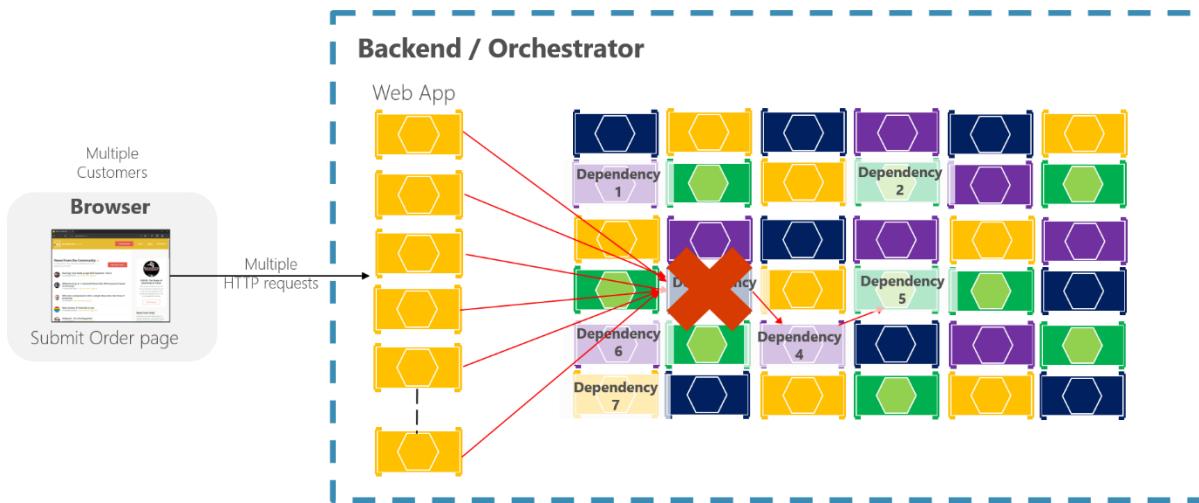


Figure 10-3. Partial failure amplified by microservices with long chains of synchronous Http calls

To minimize this problem, this is precisely why in the section "*Asynchronous microservices integration enforcing autonomy*" (introduced in the architecture chapter) we are encouraging to use asynchronous communication across the internal microservices, as also is briefly explained in the next section.

In addition to that, it is essential that you design your microservices and client applications to handle partial failures—that is, to build resilient microservices and client applications.

Strategies for handling partial failure

Strategies for dealing with partial failures include the following.

Use asynchronous communication (i.e. message based) across internal microservices. It is highly advisable not to create long chains of synchronous Http calls across the internal microservices because that wrong design will be the main cause for very bad outages, eventually. On the contrary, except for the frontend communications between the client applications and the first level of microservices or fine-grained API Gateways, it is recommended to just use asynchronous communication (i.e. message based), out of the initial request/response cycle, across the internal microservices. Eventual consistency and event-driven architectures will help to minimize the

mentioned ripple effects. These approaches will enforce a higher level of microservice autonomy and therefore prevent against the explained problem.

Use retries with exponential backoff. This technique helps to avoid short and intermittent failures by performing call retries a certain number of times, just in case the service was not available for a short time, due to intermittent network issues or when a microservice/container is moved to a different node in a cluster. However, these retries, if not handled correctly with circuit breakers, can aggravate the mentioned ripple effects, ultimately even causing a [DoS \(Denial of Service\)](#) if not designed properly.

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, but 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 is probably pointless to make additional requests, and those attempts should fail immediately. In terms of implementation, [Polly's BulkheadPolicy](#) can be used to fulfil this requirement. It's essentially a parallelization throttle with [SemaphoreSlim](#) as implementation. It also permits a "queue" outside the Bulkhead. So, you can proactively shed excess load even before execution (i.e. because capacity is deemed full), which makes its response to certain failure scenarios faster than a circuit-breaker would be, since the circuit breaker waits for the failures. The BulkheadPolicy 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

- **Microsoft. Resiliency patterns**
<https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency>
- **Microsoft. Adding Resilience and Optimizing Performance**
<https://msdn.microsoft.com/en-us/library/jj591574.aspx>
- **Polly. Bulkhead. Implementation with Polly's policy**
<https://github.com/App-vNext/Polly/wiki/Bulkhead>
- **Designing resilient applications for Azure**
<https://docs.microsoft.com/en-us/azure/architecture/resiliency/>
- **Transient fault handling**
<https://docs.microsoft.com/en-us/azure/architecture/best-practices/transient-faults>

Implementing retries with exponential backoff

[Retries with exponential backoff](#) is a technique that attempts to retry an operation, with an exponentially increasing wait time, until 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 possible approaches to implement retries logic with exponential backoff.

Implementing resilient Entity Framework Core Sql connections

For Azure SQL DB, Entity Framework 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.

```
// Startup.cs from any ASP.NET Core Web API
public class Startup
{
    // Other code...
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        //...
        services.AddDbContext<OrderingContext>(options =>
        {
            options.UseSqlServer(Configuration["ConnectionString"],
                sqlServerOptionsAction: sqlOptions =>
            {
                sqlOptions.EnableRetryOnFailure(
                    maxRetryCount: 5,
                    maxRetryDelay: TimeSpan.FromSeconds(30),
                    errorNumbersToAdd: null);
            });
        });
    }
    //...
}
```

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 retriable 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 are defining your own group of operations that need to be treated as a unit—everything inside the transaction has been rolled back if a failure occurs. You will see an exception like the following if you attempt to execute that transaction when using an EF execution strategy (retry policy) and you include several `SaveChanges` from multiple `DbContexts` in it.

```
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 show how it is implemented in `eShopOnContainers` when using two multiple `DbContexts`, the `_catalogContext` and the `IntegrationEventLogContext` objects, when updating a product and then saving the `ProductPriceChangedIntegrationEvent` object that needs to use a different `DbContext`.

```
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem  
                                              productToUpdate)  
{  
    // Other code ...  
  
    // Update current product  
    catalogItem = productToUpdate;  
  
    // Use of an EF Core resiliency strategy when using multiple DbContexts  
    // within an explicit transaction  
    // See:  
    // https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-resiliency  
    var strategy = _catalogContext.Database.CreateExecutionStrategy();  
  
    await strategy.ExecuteAsync(async () =>  
    {  
        // Achieving atomicity between original Catalog database operation and the  
        // IntegrationEventLog thanks to a local transaction  
        using (var transaction = _catalogContext.Database.BeginTransaction())  
        {  
            _catalogContext.CatalogItems.Update(catalogItem);  
            await _catalogContext.SaveChangesAsync();  
  
            // Save to EventLog only if product price changed  
            if (raiseProductPriceChangedEvent)  
                await _integrationEventLogService.SaveEventAsync(priceChangedEventArgs);  
  
            transaction.Commit();  
        }  
    });
```

The first `DbContext` is the `_catalogContext` and the second `DdContext` is within the `_integrationEventLogService` object. Finally, the `Commit` action would be performed multiple `DbContexts` and using an EF Execution Strategy.

Additional resources

Microsoft. Connection Resiliency and Command Interception with the Entity Framework

<https://docs.microsoft.com/en-us/azure/architecture/patterns/category/resiliency>

- Cesar de la Torre. Using Resilient Entity Framework Core Sql Connections and Transactions
<https://blogs.msdn.microsoft.com/cesardelatorre/2017/03/26/using-resilient-entity-framework-core-sql-connections-and-transactions-retries-with-exponential-backoff/>

Implementing custom HTTP call retries with exponential backoff

To create resilient microservices that can handle possible HTTP failure scenarios, you need to create your own implementation of retries with exponential backoff. In addition to handling resource unavailability, the exponential backoff also needs to take into account that the cloud provider might throttle availability of resources to prevent usage overload. For example, creating too many connection requests very quickly might be viewed as a Denial of Service ([DoS](#)) attack by the cloud provider. As a result, you need to provide a mechanism to scale back connection requests when a capacity threshold has been encountered.

As an initial exploration, you could implement your own code with a utility class for exponential backoff as in [RetryWithExponentialBackoff.cs](#) plus code like the following (which is also available on a [GitHub repo](#)).

```
public sealed class RetryWithExponentialBackoff
{
    private readonly int maxRetries, delayMilliseconds, maxDelayMilliseconds;

    public RetryWithExponentialBackoff(int maxRetries = 50,
                                       int delayMilliseconds = 200,
                                       int maxDelayMilliseconds = 2000)
    {
        this.maxRetries = maxRetries;
        this.delayMilliseconds = delayMilliseconds;
        this.maxDelayMilliseconds = maxDelayMilliseconds;
    }

    public async Task RunAsync(Func<Task> func)
    {
        ExponentialBackoff backoff = new ExponentialBackoff(this.maxRetries,
                                                          this.delayMilliseconds,
                                                          this.maxDelayMilliseconds);

        retry:
        try
        {
            await func();
        }
        catch (Exception ex) when (ex is TimeoutException ||
                                  ex is System.Net.Http.HttpRequestException)
        {
            Debug.WriteLine("Exception raised is: " +
                           ex.GetType().ToString() +
                           " -Message: " + ex.Message +
                           " -- Inner Message: " +
                           ex.InnerException.Message);
            await backoff.Delay();
            goto retry;
        }
    }
}

public struct ExponentialBackoff
{
    private readonly int m_maxRetries, m_delayMilliseconds, m_maxDelayMilliseconds;
    private int m_retries, m_pow;

    public ExponentialBackoff(int maxRetries, int delayMilliseconds,
```

```

        int maxDelayMilliseconds)
{
    m_maxRetries = maxRetries;
    m_delayMilliseconds = delayMilliseconds;
    m_maxDelayMilliseconds = maxDelayMilliseconds;
    m_retries = 0;
    m_pow = 1;
}

public Task Delay()
{
    if (m_retries == m_maxRetries)
    {
        throw new TimeoutException("Max retry attempts exceeded.");
    }
    ++m_retries;
    if (m_retries < 31)
    {
        m_pow = m_pow << 1; // m_pow = Pow(2, m_retries - 1)
    }
    int delay = Math.Min(m_delayMilliseconds * (m_pow - 1) / 2,
                         m_maxDelayMilliseconds);
    return Task.Delay(delay);
}
}

```

Using this code in a client C# application (another Web API client microservice, an ASP.NET MVC app, or even a C# Xamarin app) is straightforward. The following example shows how, using the `HttpClient` class.

```

public async Task<Catalog> GetCatalogItems(int page,int take, int? brand, int? type)
{
    _apiClient = new HttpClient();
    var itemsQs = $"items?pageIndex={page}&pageSize={take}";
    var filterQs = "";

    var catalogUrl =
        $"{_remoteServiceBaseUrl}items{filterQs}?pageIndex={page}&pageSize={take}";
    var dataString = "";
    //
    // Using HttpClient with Retry and Exponential Backoff
    //
    var retry = new RetryWithExponentialBackoff();
    await retry.RunAsync(async () =>
    {
        // work with HttpClient call
        dataString = await _apiClient.GetStringAsync(catalogUrl);
    });
}

return JsonConvert.DeserializeObject<Catalog>(dataString);
}

```

However, this code is suitable only as a proof of concept. The next section explains how to use more sophisticated and proven libraries.

Implementing HTTP call retries with exponential backoff with Polly

The recommended approach for retries with exponential backoff is to take advantage of more advanced .NET libraries like the open source library [Polly](#).

Polly is a .NET library that provides resilience and transient-fault handling capabilities. You can implement those capabilities easily by applying Polly's policies such as Retry, Circuit Breaker, Bulkhead Isolation, Timeout and Fallback. Polly targets .NET 4.x and the .NET Standard Library 1.0 (which supports .NET Core).

The Retry policy in Polly is the approach used by eShopOnContainers when implementing HTTP retries. You can implement an interface so you can inject either standard `HttpClient` functionality or a resilient version of `HttpClient` using Polly, depending on what retry policy configuration you want to use.

The following example shows the interface implemented by eShopOnContainers.

```
public interface IHttppClient
{
    HttpClient Inst { get; }
    Task<string> GetStringAsync(string uri);
    Task<HttpResponseMessage> PostAsync<T>(string uri, T item);
    Task<HttpResponseMessage> DeleteAsync(string uri);
}
```

You can use the standard implementation if you do not want to use a resilient mechanism, as when you are developing or testing simpler approaches. The following example shows this.

```
public class StandardHttpClient : IHttppClient
{
    private HttpClient _client;
    private ILogger _logger;
    public HttpClient Inst => _client;
    public StandardHttpClient()
    {
        _client = new HttpClient();
        _logger = new LoggerFactory().CreateLogger(nameof(StandardHttpClient));
    }

    public Task<string> GetStringAsync(string uri) =>
        _client.GetStringAsync(uri);

    public Task<HttpResponseMessage> PostAsync<T>(string uri, T item)
    {
        var contentString = new StringContent(JsonConvert.SerializeObject(item),
                                                System.Text.Encoding.UTF8,
                                                "application/json");
        return _client.PostAsync(uri, contentString);
    }
    // Other methods ...
}
```

The interesting implementation is to code another, similar class, but using Polly to implement the resilient mechanisms you want to use—in the following example, retries with exponential backoff.

```

public class ResilientHttpClient : IHttpClient
{
    private HttpClient _client;
    private PolicyWrap _policyWrapper;
    private ILogger<ResilientHttpClient> _logger;
    public HttpClient Inst => _client;

    public ResilientHttpClient(Policy[] policies,
                               ILogger<ResilientHttpClient> logger)
    {
        _client = new HttpClient();
        _logger = logger;

        // Add Policies to be applied
        _policyWrapper = Policy.WrapAsync(policies);
    }

    public Task<string> GetStringAsync(string uri) =>
        HttpInvoker(() =>
            _client.GetStringAsync(uri));

    public Task<HttpResponseMessage> PostAsync<T>(string uri, T item) =>
        // a new StringContent must be created for each retry
        // as it is disposed after each call
        HttpInvoker(() =>
    {
        var response = _client.PostAsync(uri, new StringContent(JsonConvert.
                        SerializeObject(item),
                        System.Text.Encoding.UTF8, "application/json"));
        // raise exception if HttpResponseMessage 500
        // needed for circuit breaker to track fails
        if (response.Result.StatusCode == HttpStatusCode.InternalServerError)
        {
            throw new HttpRequestException();
        }

        return response;
    });

    public Task<HttpResponseMessage> DeleteAsync(string uri) =>
        HttpInvoker(() => _client.DeleteAsync(uri));

    private Task<T> HttpInvoker<T>(Func<Task<T>> action) =>
        // Executes the action applying all
        // the policies defined in the wrapper
        _policyWrapper.ExecuteAsync(() => action());

    // Other HTTP methods...
}

```

With Polly, you define a Retry policy with the number of retries, the exponential backoff configuration, and the actions to take when there is an HTTP exception, such as logging the error. In this case, the policy is configured so it will try the number of times specified once applied the policies when registering the types in the IoC container. Then, because of the exponential backoff configuration, whenever the code detects an `HttpRequest` exception, it retries the request after waiting an amount

of time that goes up exponentially 2 seconds the first time, 4 seconds the second time, 8 seconds the third time, and so on.

The important method is `HttpInvoker`, which is what makes HTTP requests throughout this utility class. The method internally executes the HTTP request with `_policyWrapper.ExecuteAsync`, which takes into account the retry policy.

In `eShopOnContainers` you specify Polly's policies when registering the types at the IoC container, as in the following code.

```
//Startup.cs class
if (Configuration.GetValue<string>("UseResilientHttp") == bool.TrueString)
{
    services.AddTransient<IResilientHttpClientFactory,
                           ResilientHttpClientFactory>();

    services.AddTransient<IHttpClient,
                           ResilientHttpClient>(sp =>
        sp.GetService<IResilientHttpClientFactory>().
            CreateResilientHttpClient());
}
else
{
    services.AddTransient<IHttpClient, StandardHttpClient>();
}
```

Where, basically, it is within the `ResilientHttpClientFactory` in the `CreateResilientHttpClient()` method where you apply Polly's `WaitAndRetryAsunc` policie, as shown in the following code.

```
public ResilientHttpClient CreateResilientHttpClient()
    => new ResilientHttpClient(CreatePolicies(), _logger);

private Policy[] CreatePolicies()
    => new Policy[]
{
    Policy.Handle<HttpRequestException>()
        .WaitAndRetryAsync(
            // number of retries
            6,
            // exponential backoff
            retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
            // on retry
            (exception, timeSpan, retryCount, context) =>
            {
                var msg = $"Retry {retryCount} implemented w/ Polly's
                    RetryPolicy " +
                    $"of {context.PolicyKey} " +
                    $"at {context.ExecutionKey}, " +
                    $"due to: {exception}.";
                _logger.LogWarning(msg);
                _logger.LogDebug(msg);
            },
        )
}
```

Implementing 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 because of resources being slow or 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 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 is unlikely to succeed. Instead, the application should be coded to accept that the operation has failed and handle the failure accordingly.

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 is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker, and it should abandon retry attempts if the circuit breaker indicates that a fault is not transient.

Implementing a Circuit Breaker pattern with Polly

As when implementing retries, the recommended approach for circuit breakers is to take advantage of proven .NET libraries like Polly.

The eShopOnContainers application uses the Polly Circuit Breaker policy when implementing HTTP retries. In fact, the application applies both policies to the `ResilientHttpClient` utility class.

Whenever you use an object of type `ResilientHttpClient` for HTTP requests (from `eShopOnContainers`), you will be applying both those policies, but you could add additional policies, too..

The only code added compared 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 at the end of the following code.

```
public ResilientHttpClient CreateResilientHttpClient()
    => new ResilientHttpClient(CreatePolicies(), _logger);

private Policy[] CreatePolicies()
    => new Policy[]
    {
        Policy.Handle<HttpRequestException>()
            .WaitAndRetryAsync(
                // number of retries
                6,
                // exponential backoff
                retryAttempt => TimeSpan.FromSeconds(Math.Pow(2, retryAttempt)),
                // on retry
                (exception, timeSpan, retryCount, context) =>
                {

```

```

        var msg = $"Retry {retryCount} implemented w/ Polly's
        RetryPolicy" +
        $"of {context.PolicyKey} " +
        $"at {context.ExecutionKey}, " +
        $"due to: {exception}.";
        _logger.LogWarning(msg);
        _logger.LogDebug(msg);
    }),
Policy.Handle<HttpRequestException>()
.CircuitBreakerAsync(
    // number of exceptions before breaking circuit
    5,
    // time circuit opened before retry
    TimeSpan.FromMinutes(1),
    (exception, duration) =>
{
    // on circuit opened
    _logger.LogTrace("Circuit breaker opened");
},
() =>
{
    // on circuit closed
    _logger.LogTrace("Circuit breaker reset");
})});
}

```

The code adds a policy to the HTTP wrapper. That policy defines a circuit breaker that opens when the code detects the specified number of consecutive exceptions (exceptions in a row), as passed in the exceptionsAllowedBeforeBreaking parameter (5 in this case). When the circuit is open, HTTP requests do not work, but an exception is raised.

Circuit breakers should also be used to redirect requests to a fallback infrastructure if you might have issues in a particular resource that is deployed in a different environment than the client application or service that is performing the HTTP call. That way, if there is 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. For instance, Polly is planning a FailoverPolicy to automate this [Failover Policy](#) scenario.

Of course, all those features are for cases where you are managing the failover from within the .NET code, as opposed to having it managed automatically for you by Azure, with location transparency.

Using the ResilientHttpClient utility class from eShopOnContainers

You use the ResilientHttpClient utility class in a way similar to how you use the .NET HttpClient class. In the following example from the eShopOnContainer MVC web application (the OrderingServiceAgent used by OrderController), the ResilientHttpClient object is injected through the httpClient parameter of the constructor. Then the object is used to perform HTTP requests.

```

public class OrderingService : IOrderingService
{
    private IHttpClient _httpClient;
    private readonly string _remoteServiceBaseUrl;
    private readonly IOptionsSnapshot<AppSettings> _settings;

```

```

private readonly IHttpContextAccessor _httpContextAccesor;

public OrderingService(IOptionsSnapshot<AppSettings> settings,
                      IHttpContextAccessor httpContextAccesor,
                      IHttpClient httpClient)
{
    _remoteServiceBaseUrl = $"{settings.Value.OrderingUrl}/api/v1/orders";
    _settings = settings;
    _httpContextAccesor = httpContextAccesor;
    _httpClient = httpClient;
}

async public Task<List<Order>> GetMyOrders(ApplicationUser user)
{
    var context = _httpContextAccesor.HttpContext;
    var token = await context.Authentication.GetTokenAsync("access_token");

    _httpClient.Inst.DefaultRequestHeaders.Authorization = new
        System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
    var ordersUrl = _remoteServiceBaseUrl;
    var dataString = await _httpClient.GetStringAsync(ordersUrl);
    var response = JsonConvert.DeserializeObject<List<Order>>(dataString);

    return response;
}

// Other methods ...

async public Task CreateOrder(Order order)
{
    var context = _httpContextAccesor.HttpContext;
    var token = await context.Authentication.GetTokenAsync("access_token");

    _httpClient.Inst.DefaultRequestHeaders.Authorization = new
        System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
    _httpClient.Inst.DefaultRequestHeaders.Add("x-requestid",
        order.RequestId.ToString());

    var ordersUrl = $"{_remoteServiceBaseUrl}/new";
    order.CardTypeId = 1;
    order.CardExpirationApiFormat();
    SetFakeIdToProducts(order);

    var response = await _httpClient.PostAsync(ordersUrl, order);

    response.EnsureSuccessStatusCode();
}
}

```

Whenever the member object `_httpClient` is used, it internally uses the wrapper class with Polly policies—the Retry policy, the Circuit Breaker policy, any other policy that you might want to apply from the Polly policies collection.

Testing retries with 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 into Docker it needs to set up the images and the database. This delay in some containers starting slower than other 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 app can get an exception when trying to consume that particular container.

You might also see this type of error on startup when the app 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 this issue is by using the Retry pattern we illustrated earlier. It's also why, when starting the solution, you might get log traces or warnings like the following:

```
"Retry 1 implemented with Polly's RetryPolicy, due to:  
System.Net.Http.HttpRequestException: An error occurred while sending the  
request. ---> System.Net.Http.CurlException: Couldn't connect to server\n    at  
System.Net.Http.CurlHandler.ThrowIfCURLError(CURLcode error)\n    at [...].
```

Testing the circuit breaker with eShopOnContainers

There are a few ways you can 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 is 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 some custom middleware that is implemented in the ordering 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 /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 is no response.

- GET /failing?enable

This request enables the middleware.

GET /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 5102.

`http://localhost:5102/failing?enable`

You can then check the status using the URI `http://localhost:5102/failing`, as shown in Figure 10-4.

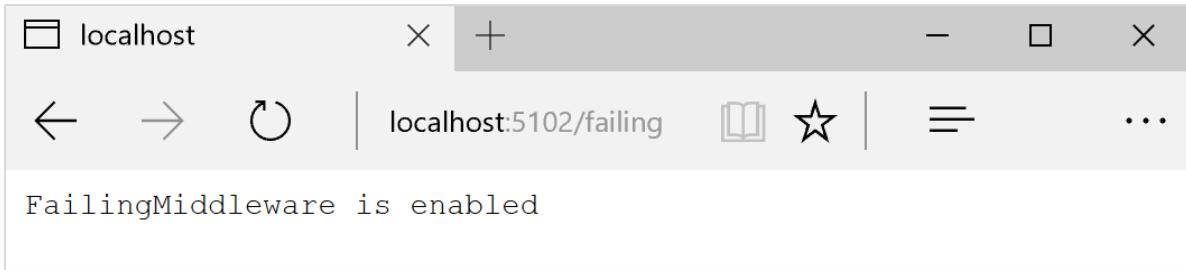


Figure 10-4 Simulating a failure with an ASP.NET middleware.

At this point, the ordering microservice responds 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 fails, 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.

```
[HttpPost]
public async Task<IActionResult> Create(Order model, string action)
{
    try
    {
        if (ModelState.IsValid)
        {
            var user = _appUserParser.Parse(HttpContext.User);
            await _orderSvc.CreateOrder(model);

            //Redirect to historic list.
            return RedirectToAction("Index");
        }
    }
    catch(BrokenCircuitException ex)
    {
        ModelState.AddModelError("Error",
                               "It was not possible to create a new order, please try later on");
    }
    return View(model);
}
```

Here's a summary. The Retry policy tries several times to make the HTTP request and gets HTTP errors. When the number of tries 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 10-5.

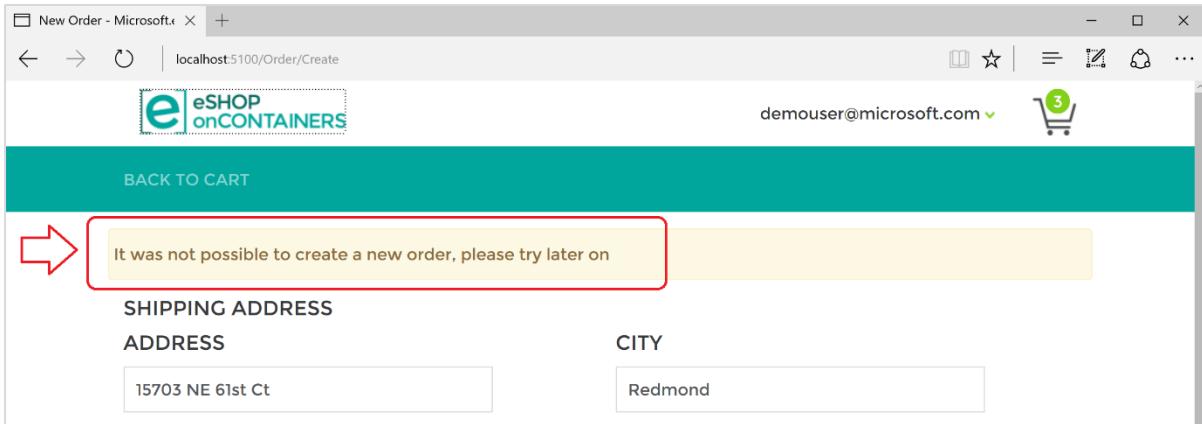


Figure 10-5. Circuit breaker returning an error to the UI

You can implement different logic for when to open the circuit. Or you can try an HTTP request against a different backend microservice if there is a fallback datacenter or redundant backend system.

Finally, and as another possibility for the `CircuitBreakerPolicy` is to use the methods `.Isolate()` (which forces open and holds the circuit open); and `.Reset()` (which closes it again). These could be used to build an utility HTTP endpoint which 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, for example when you want to upgrade it, or trip the circuit manually to protect a downstream system you suspect to be faulting.

Additional resources

Retry pattern

<https://docs.microsoft.com/en-us/azure/architecture/patterns/retry>

Connection Resiliency (Entity Framework Core)

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-resiliency>

Polly (.NET resilience and transient-fault-handling library)

<https://github.com/App-vNext/Polly>

Circuit Breaker pattern

<https://docs.microsoft.com/en-us/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 simply 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 are 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 is customized for your application, you can detect and fix issues for your running application much more easily.

Implementing health checks in ASP.NET Core services

When developing an ASP.NET Core microservice or web application, you can use a library named `HealthChecks` from the ASP.NET team. (As of April 2017, an early release is available on GitHub).

This library is easy to use and provides features that let you validate that any specific external resource (like a SQL Server database or remote API) is working properly. When you use this library, you can also decide what it means that the resource is healthy, as we explain later.

In order to use this library, you need to first use the library in your microservices. Second, you need a front-end application that queries for the health reports.

Using the `HealthChecks` library in your back-end ASP.NET microservices

You can see how the `HealthCheck` library is used in the `eShopOnContainers` sample application. To begin, you need to define what constitutes a healthy status for each microservice. In the sample application, the microservices are healthy if the microservice API is accessible via HTTP and if its related SQL Server database is also available.

In the future, you will be able to install the `HealthChecks` library as a NuGet package. But as of this writing, you need to download and compile the code as part of your solution. Clone the code available at <https://github.com/aspnet/HealthChecks> and copy the following folders to your solution.

```
src/common  
src/Microsoft.AspNetCore.HealthChecks  
src/Microsoft.Extensions.HealthChecks  
src/Microsoft.Extensions.HealthChecks.Data
```

Figure 10-6 shows the `HealthLibrary` library in Visual Studio, ready to be used as a building block by any microservices.

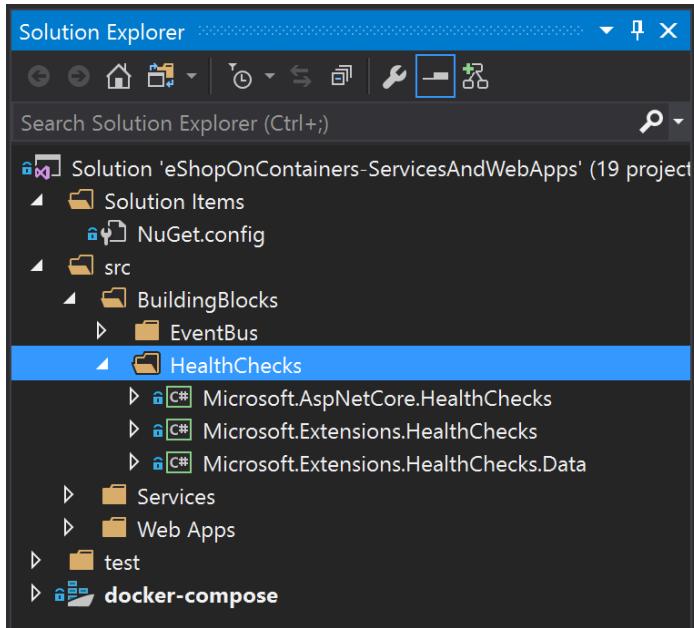


Figure 10-6. ASP.NET Core HealthChecks library source code in a Visual Studio solution

The first thing to do in each microservice project is to add a reference to the three `HealthCheck` libraries. After that, you add the `HealthCheck` actions that you want to perform in that microservice. These actions are basically dependencies on other microservices (`HttpCheck`) or databases (currently `SqlCheck`). You add the action within the `Startup` class of each ASP.NET microservice or ASP.NET web application.

Each service or web app should be configured by adding all its HTTP or database dependencies as one `AddHealthCheck` method. E.i. the MVC depends on many services, therefore has a several "AddCheck" added to the health checks to take into account.

For instance, in the following code you can see how the catalog microservice adds a dependency on its SQL Server database.

```
// Startup.cs from Catalog.api microservice
//
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        // Add framework services
        services.AddHealthChecks(chcks =>
        {
            chcks.AddSqlCheck("Catalog_Db", Configuration["ConnectionString"]);
        });
        // Other services
    }
}
```

However, the MVC web application has multiple dependencies on the rest of the microservices. Therefore, it calls one `AddUrlCheck` method for each microservice, as shown in the following example.

```
// Startup.cs from the MVC web app
```

```

public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddMvc();
        services.Configure<AppSettings>(Configuration);

        services.AddHealthChecks(chcks =>
        {
            chcks.AddUrlCheck(Configuration["CatalogUrl"]);
            chcks.AddUrlCheck(Configuration["OrderingUrl"]);
            chcks.AddUrlCheck(Configuration["BasketUrl"]);
            chcks.AddUrlCheck(Configuration["IdentityUrl"]);
            chcks.AddUrlCheck(Configuration["CallBackUrl"]);
        });
    }
}

```

Thus, a microservice won't provide a "healthy" status until all of its checks are healthy as well.

If the microservice does not have an dependency on a service or on SQL Server, you should just add an AlwaysOK check. The following code is from the eShopOnContainers basket.api microservice. (The basket microservice uses the Redis cache, but the library does not yet include a Redis health check provider.)

```

services.AddHealthChecks(chcks =>
{
    chcks.AddValueTaskCheck("Always OK", () => new
        ValueTask<IHealthCheckResult>(HealthCheckResult.Healthy("Ok")));
});

```

For each service or web app to be able to expose the HealthCheck endpoint, it has to enable the `UserHealthChecks([url_for_health_checks])` extension method. This method goes at the `WebHostBuilder` level in the main method of the `Program` class of your ASP.NET Core service or web app, right after `UseKestrel` as shown in the code below.

```

namespace Microsoft.eShopOnContainers.WebMVC
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseHealthChecks("/hc")
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
            host.Run();
        }
    }
}

```

The process works like this: each microservice exposes the endpoint `/hc`. That endpoint is created by the `HealthCheck` library ASP.NET Core middleware. When that endpoint is invoked, it runs all the health checks that are configured in the `AddHealthChecks` method in the `Startup` class.

The `UseHealthChecks` method expects a port or a path. That port or path is the endpoint to be used to check the health state of the service. For instance, the catalog microservice uses the path `/hc`.

Querying your microservices to report about their health status

When you have configured health checks as described here, once the microservice is running in Docker, you can directly check from a browser if it is healthy. (This does require that you are publishing the container port out of the Docker host, so you can access the container through `localhost` or through the external docker host IP.) Figure 10-7 shows a request in a browser and the corresponding response.

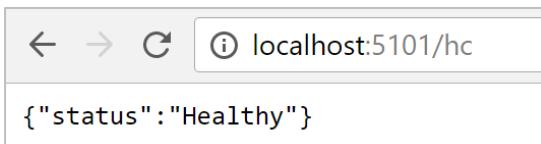


Figure 10-7. Checking health status of a single service from a browser

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. It also means that internally the service also checked the health of its SQL Server database dependency and that health check was reported itself as healthy.

Watchdogs

A watchdog is a separate service that can watch health and load across services, and report health about the microservices by querying the previously explained Health Checks. 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 10-8. This is the simplest watchdog you could have, since all it does is shows the state of the microservices and web applications in `eShopOnContainers`. Usually a watchdog also takes actions when it detects unhealthy states.

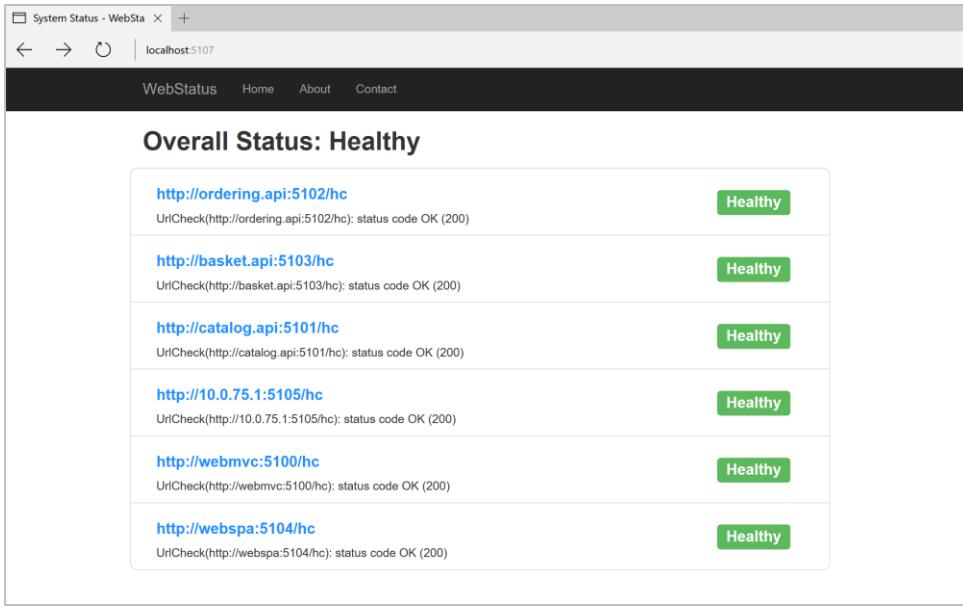


Figure 10-8. Sample HealthCheck report in eShopOnContainers

In summary, the ASP.NET middleware of the ASP.NET Core HealthChecks library provides a single health check endpoint for each microservice. This will execute all the health checks defined within it and return an overall health state depending on all of those checks.

The HealthChecks library is extensible through new health checks to future external resources. For example, we expect that in the future the library will have health checks for Redis cache and for other databases. The library allows health reporting by multiple service or application dependencies, and you can then take actions based on those health checks.

Health checks when using orchestrators

To monitor the availability of your microservices, orchestrators like Docker Swarm, Kubernetes, and Service Fabric periodically perform health checks by sending pings, attempting connections, and sending requests to test the microservices. When an orchestrator determines that a 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 the services in phases—for example, they might update one-fifth of the cluster surface per application upgrade. The set of nodes that is 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 if you are using Azure Service Fabric, it provides its own [Health Monitoring model](#), which is more advanced than simple Healthchecks.

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 we showed when we explained [ASP.NET Core Healthchecks](#). Or you could use more advanced tools like Azure Application Insights and Operations Management Suite to raise alerts based on the stream of events.

Finally, if you were storing all the event streams, you can use Microsoft Power BI or a third-party solution like Kibana or Splunk to visualize the data.

Additional resources

- **ASP.NET Core Healthchecks** (early release)
<https://github.com/aspnet/HealthChecks/>
- **Introduction to Service Fabric health monitoring**
<https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-health-introduction>
- **Azure Application Insights**
<https://azure.microsoft.com/en-us/services/application-insights/>
- **Microsoft Operations Management Suite**
<https://www.microsoft.com/en-us/cloud-platform/operations-management-suite>

Securing .NET Microservices and Web Applications

Implementing authentication in .NET microservices and web applications

It's often necessary for resources and APIs exposed 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 ascertaining a user's identity or granted permissions.

In microservice scenarios, authentication is typically handled centrally. If you are using an API gateway, the API gateway is a good place to authenticate, as shown in Figure 11-1. 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.

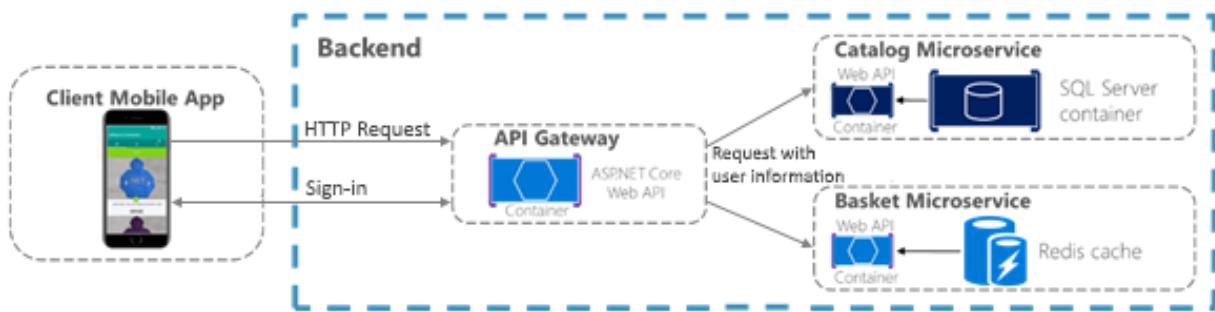


Figure 11-1. Centralized authentication with an API Gateway

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 can be shared between applications, if needed, in ASP.NET Core with [data protection services](#).) This pattern is illustrated in Figure 11-2.

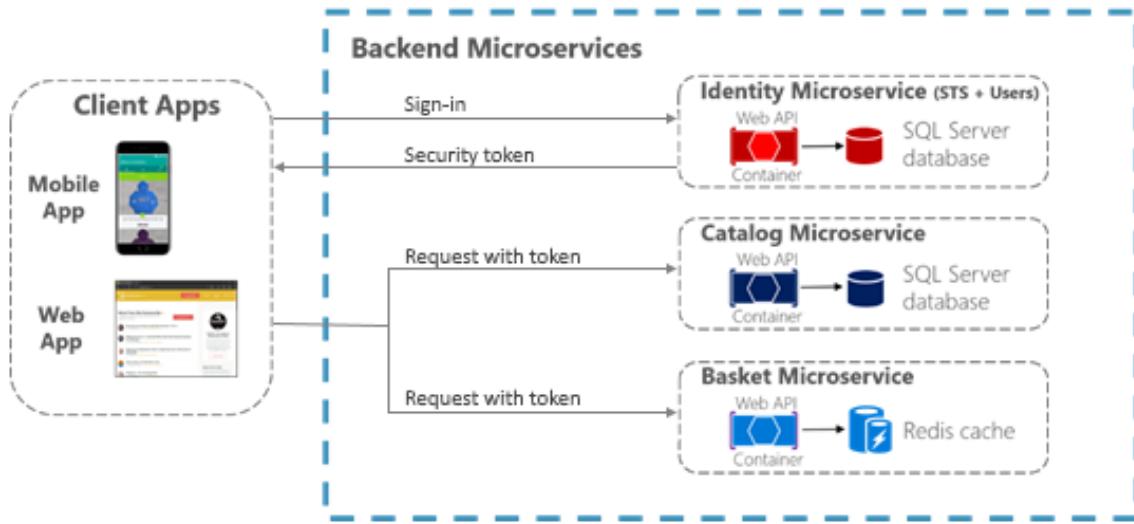


Figure 11-2. Authentication by identity microservice; trust shared using an authorization token

Authenticating using ASP.NET Core Identity

The primary mechanism in ASP.NET Core for identifying an app'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 EntityFramework 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 dtable storage, DocumentDB, or other locations.

The following code is taken from the ASP.NET Core Web Application project template with individual user account authentication selected. It shows how to configure ASP.NET Core Identity using `EntityFramework.Core` in the `Startup.ConfigureServices` method.

```

services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddIdentity< ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();

```

Once ASP.NET Core Identity is configured, you enable it by calling `app.UseIdentity` in the service's `Startup.Configure` method.

Using ASP.NET Core Identity enables several scenarios:

- Local user information can be created and stored using the `UserManager` type (`userManager.CreateAsync`).
- Users can be authenticated using the `SignInManager` type. You can use `signInManager.SignInAsync` to sign in directly, or `signInManager.PasswordSignInAsync` to verify the user's password first.
- User information and claims for signed-in users are stored in a cookie for use in subsequent requests.

- Middleware is registered in the ASP.NET Core application's pipeline to read user information from cookies 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.

Authenticating using external providers

ASP.NET Core also supports using [external authentication providers](#) to let users log in via [OAuth 2.0](#) flows. This means that users can log 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, you include the appropriate authentication middleware in your app's HTTP request processing pipeline. This middleware is responsible for handling requests to return URI routes from the authentication provider, capturing identity information, and making it available via the `SignInManager.GetExternalLoginInfo` method.

Popular external authentication providers and their associated NuGet packages are shown in the following table.

| Provider | Package |
|-----------|---|
| Microsoft | <code>Microsoft.AspNetCore.Authentication.MicrosoftAccount</code> |
| Google | <code>Microsoft.AspNetCore.Authentication.Google</code> |
| Facebook | <code>Microsoft.AspNetCore.Authentication.Facebook</code> |
| Twitter | <code>Microsoft.AspNetCore.Authentication.Twitter</code> |

In all cases, the middleware is registered with a call to a registration method similar to `app.Use{ExternalProvider}Authentication` in `startup.Configure`. These registration methods take an `options` object that contains an application ID and secret information, as needed by the provider. External authentication providers require the application to be registered (as explained in [ASP.NET Core documentation](#)) so that they can inform the user what application is requesting access to their identity.

Once the middleware is registered in `Startup.Configure`, you can prompt users to log in from any controller action. To do this, you create an `AuthenticationProperties` object that includes the authentication provider's name and a redirect URL. You then return a `Challenge` response that passes the `AuthenticationProperties` object.

```
var properties = _signInManager.ConfigureExternalAuthenticationProperties(provider,
    redirectUrl);
return Challenge(properties, provider);
```

The redirectUrl parameter includes the URL that the external provider should redirect to once the user has authenticated. The URL should represent an action that will sign in the user based on external identity information, as in the following simplified sample code sample.

```
// Sign in the user with this external login provider if the user
// already has a login.
var result = await _signInManager.ExternalLoginSignInAsync(info.LoginProvider,
info.ProviderKey, isPersistent: false);
if (result.Succeeded)
{
    return RedirectToLocal(returnUrl);
}
else
{
    ApplicationUser newUser = new ApplicationUser
    {
        // The user object can be constructed with whatever specific claims are
        // returned by the external authentication provider used, or can
        // be created by gathering input from the user.
        UserName = info.Principal.FindFirstValue(ClaimTypes.Name),
        Email = info.Principal.FindFirstValue(ClaimTypes.Email)
    };

    var identityResult = await _userManager.CreateAsync(newUser);
    if (identityResult.Succeeded)
    {
        identityResult = await _userManager.AddLoginAsync(newUser, info);
        if (identityResult.Succeeded)
        {
            await _signInManager.SignInAsync(newUser, isPersistent: false);
        }
    }

    return RedirectToLocal(returnUrl);
}
```

If you choose the **Individual User Account** authentication option when you create the ASP.NET Core web application project in Visual Studio, all the code necessary to sign in with an external provider is already in the project, as shown in Figure 11-3.

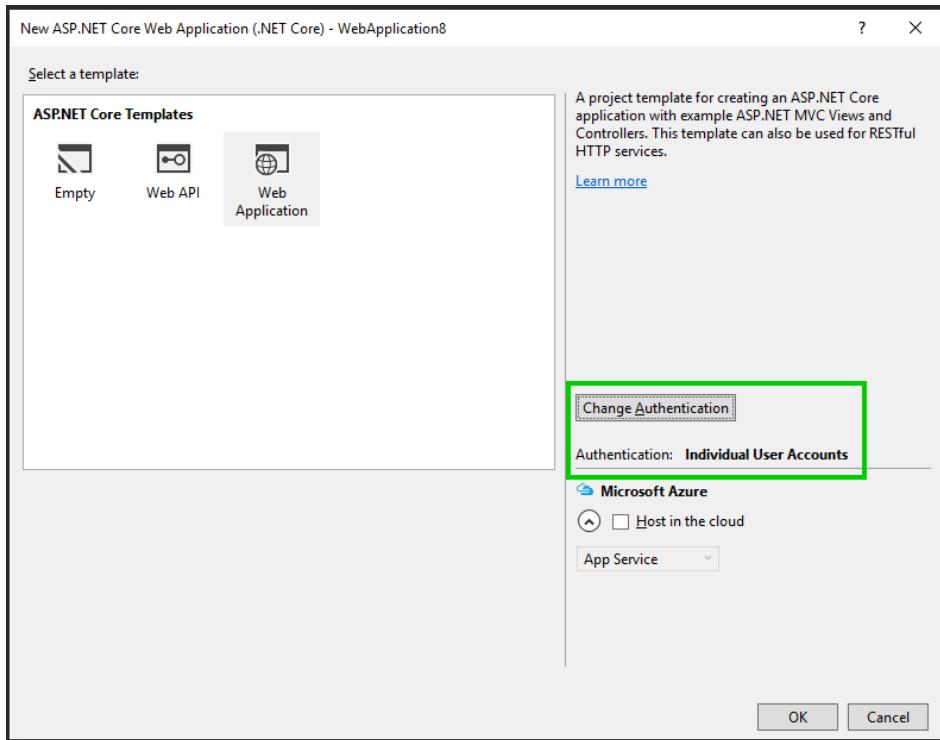


Figure 11-3. Selecting an option for using external authentication when creating a web application project

Other external authentication providers

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 repo](#) on GitHub.

It is also possible, of course, to create your own external authentication middleware.

Authenticating 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](#).

Authenticating 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 against Azure Active Directory](#), an ASP.NET Core web application can use middleware from that package as shown in the following example:

```
// Configure the OWIN pipeline to use OpenID Connect auth
app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions
{
    ClientId = Configuration["AzureAD:ClientId"],
    Authority = String.Format(Configuration["AzureAd:AadInstance"],
        Configuration["AzureAd:Tenant"]),
    ResponseType = OpenIdConnectResponseType.IdToken,
    PostLogoutRedirectUri = Configuration["AzureAd:PostLogoutRedirectUri"]
});
```

The configuration values are Azure Active Directory values that are created when your application is [registered as an Azure AD client](#). A single client ID can be shared among multiple microservices in an application if they all need to authenticate users authenticated via Azure Active Directory.

Note that when you use this workflow, the ASP.NET Core Identity middleware is not needed, because all user information storage and authentication is handled by Azure Active Directory.

Issuing 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](#) is an OpenID Connect provider that integrates 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 call `app.UseIdentityServer` in the `Startup.Configure` method 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`.
2. You configure IdentityServer4 in `Startup.ConfigureServices` by making a call to `services.AddIdentityServer`.

Further configuration is needed with subsequent calls for the following IdentityServer4 settings:

- The [credentials](#) to use for signing.
- [Identity and API resources](#) that users might request access to.
 - API resources represent some 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<T>` 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` might look like the following example:

```
// Add IdentityServer services
services.AddSingleton<IClientStore, CustomClientStore>();

services.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<ApplicationUser>();
```

Consuming 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. A simple example of how to use middleware might look like the following example. This code must precede calls to ASP.NET Core's MVC middleware (`app.UseMvc`).

```
app.UseJwtBearerAuthentication(new JwtBearerOptions()
{
    Audience = "http://localhost:5001/",
    Authority = "http://localhost:5000/",
    AutomaticAuthenticate = true
});
```

The parameters in this usage are:

- `Audience` represents the recipient of the incoming token or the resource that the token grants access to. If the value specified in this parameter doesn't match the `aud` 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.
- `AutomaticAuthenticate` is a Boolean value that indicates whether the user defined by the token should be automatically logged in.

Another parameter, `RequireHttpsMetadata`, is not used in this example. It is useful for testing purposes; you set this parameter to `false` so that 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://docs.microsoft.com/en-us/aspnet/core/security/data-protection/compatibility/cookie-sharing#sharing-authentication-cookies-between-applications>
- **Introduction to Identity**
<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity>
- **Rick Anderson. Two-factor authentication with SMS**
<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/2fa>
- **Enabling authentication using Facebook, Google and other external providers**
<https://docs.microsoft.com/en-us/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>
- **Danny Strockis. Integrating Azure AD into an ASP.NET Core web app**
<https://azure.microsoft.com/en-us/resources/samples/active-directory-dotnet-webapp-openidconnect-aspnetcore/>
- **IdentityServer4. Official documentation**
<https://identityserver4.readthedocs.io/en/release/>

About authorization in .NET microservices and web applications

After authentication, ASP.NET Core Web APIs often 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.

Implementing 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 (which adjusts roles in persisted storage) and `UserManager` type (which can assign or unassign users from roles).

If you are 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` header, as shown in the following example:

```
[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 `Adminstrator` or `PowerUser` role, *and*
- Be in the `RemoteEmployee` role, *and*
- Satisfy a custom handler for `CustomPolicy` authorization.

Implementing policy-based authorization

Custom authorization rules can also be written using [authorization policies](#). In this section we provide an overview. More detail is available in the online [ASP.NET Authorization Workshop](#).

Custom authorization policies are registered in the `Startup.ConfigureServices` method using the `service.AddAuthorization` method. This method takes an action method 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 are 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`, 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<T>`, where `T` is the type of `IAuthorizationRequirement` that the handler can satisfy. The handler must implement the `HandleRequirementAsync(AuthorizationHandlerContext context, T requirement)` 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(requirement)` 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](#).

Additional resources

- **ASP.NET Core Authentication**
<https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity>
- **ASP.NET Core Authorization**
<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction>
- **Role based Authorization**
<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/roles>
- **Custom Policy-Based Authorization**
<https://docs.microsoft.com/en-us/aspnet/core/security/authorization/policies>

Storing app 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 is a best practice to not include secrets in source code and certainly 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 should separate the secrets for accessing development and staging resources from those 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 as 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.

Storing secrets as 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 (those nested in configuration sections), create a name for the environment variables that includes the full hierarchy of the secret's name, delimited with colons (:).

For example, setting an environment variable `Logging:LogLevel:Default` to `Debug` 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 that environment variables are generally stored as plain text, so if the machine or process with the environment variables is compromised, the environment variable values will be visible.

Storing secrets using the ASP.NET Core Secret Manager

The ASP.NET Core [Secret Manager](#) tool provides another method of keeping secrets out of source code. To use the Secret Manager tool, include a tools reference (`DotNetCliToolReference`) to the `Microsoft.Extensions.SecretManager.Tools` package in your project file. Once that reference 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 is using the secrets. Therefore, you must be sure to set the `UserSecretsId` property in your project file. The actual string used as the ID is not important as long as it is unique in the project.

```
<PropertyGroup>
  <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

Using secrets stored with Secret Manager in an application is similar to using secrets stored as environment variables. You just need to call `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.

Using 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 `Microsoft.Extensions.Configuration.AzureKeyVault` 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-AzureRmADApplication](#) PowerShell cmdlet. If you do this, you pass the raw certificate data as a base-64 string as the `CertValue` parameter. 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-AzureRmADServicePrincipal -ApplicationId "<Application ID guid>"
Set-AzureRmKeyVaultAccessPolicy -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 `IConfigurationBuilder.AddAzureKeyVault` extension method when you create an `IConfigurationRoot` instance. Note that calling `AddAzureKeyVault` will require the application ID that was registered and given access to the key vault in the previous steps.

Currently, the .NET Standard Library and .NET Core support getting configuration information from an Azure Key Vault using a client ID and client secret. .NET Framework applications can use an overload of `IConfigurationBuilder.AddAzureKeyVault` that takes a certificate in place of the client secret. As of this writing, work is [in progress](#) to make that overload available in .NET Standard and .NET Core. Until the `AddAzureKeyVault` overload that accepts a certificate is available, ASP.NET Core applications can access an Azure Key Vault with certificate-based authentication by explicitly creating a `KeyVaultClient` object, as shown in the following example:

```

// Configure Key Vault client
var kvClient = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(async
    (authority, resource, scope) =>
{
    var cert = // Get certificate from local store/file/key vault etc. as needed
    // From the Microsoft.IdentityModel.Clients.ActiveDirectory pacakge
    var authContext = new AuthenticationContext(authority,
                                                TokenCache.DefaultShared);
    var result = await authContext.AcquireTokenAsync(resource,
                                                    // From the Microsoft.Rest.ClientRuntime.Azure.Authentication pacakge
                                                    new ClientAssertionCertificate("<Application ID>", cert));
    return result.AccessToken;
}));

// Get configuration values from Key Vault
var builder = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    // Other configuration providers go here.
    .AddAzureKeyVault("<KeyValueUri>", kvClient,
                      new DefaultKeyVaultSecretManager());
```

In this example, the call to `AddAzureKeyVault` comes at the end of configuration provider registration. It is a best practice to register Azure Key Vault as the last configuration provider so that it has an opportunity to override configuration values from previous providers, and so that no configuration values from other sources override those from the key vault.

Additional resources

- **Using Azure Key Vault to protect application secrets**
<https://docs.microsoft.com/en-us/azure/guidance/guidance-multitenant-identity-keyvault>
- **Safe storage of app secrets during development**
<https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets>
- **Configuring data protection**
<https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/overview>

- **Key management and lifetime**
<https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/default-settings#data-protection-default-settings>
- **Microsoft.Extensions.Configuration.DockerSecrets** GitHub repo
<https://github.com/aspnet/Configuration/tree/dev/src/Microsoft.Extensions.Configuration.DockerSecrets>

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 the important benefit of cost savings because containers are a solution to deployment problems caused by the lack of dependencies in production environments. Containers significantly improve DevOps and production operations.

Containers will be ubiquitous. Docker is becoming the de facto standard in the container industry, supported by the most significant vendors in the Windows and Linux ecosystems. This includes Microsoft, Amazon AWS, Google, and IBM. In the near future, Docker will probably be ubiquitous in both cloud and on-premises datacenters.

Containers as 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 multiple independent subsystems in the form of autonomous services. In a microservice-based architecture, the application is built as a collection of services that can be developed, tested, versioned, deployed, and scaled independently; this can include any related autonomous database.

Domain-Driven Design and SOA. The microservices architecture patterns derive from SOA (Service Oriented Architecture) and Domain-Driven Design. When designing and developing microservices for environments with evolving business rules shaping a particular domain, it is important to take into account Domain-Driven Design 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 higher level of complexity 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 is worth investing in a more complex software architecture, because it will provide better long-term agility and application maintenance.

Containers for any app. Containers are convenient for microservices, but are not exclusive for them. Containers can also be used with monolithic applications, including legacy applications based on the traditional .NET Framework and modernized through 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 being able to debug multi-container applications.

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 extremely slowly to requests; or it might simply 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 custom ones, or ones based on cloud infrastructure, such as high-level frameworks from orchestrators or service buses.

Security. Our modern world of containers and microservices can expose new vulnerabilities. Basic application security is based on authentication and authorization; multiple ways exist to implement these. However, container security includes 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 other types of confidential information—usually referred to as application secrets. Any secure solution will must follow security best practices, such as encrypting secrets while in transit; encrypting secrets at rest; and preventing secrets from unintentionally leaking when consumed by the final application. Those secrets need to be stored and kept safe somewhere. To help with security, you can take advantage of your chosen orchestrator's infrastructure, or of cloud infrastructure like Azure Key Vault and the ways it provides for application code to use it.

Orchestrators. Docker orchestrators like the ones provided in Azure Container Service (Kubernetes, Mesos DC/OS, and Docker Swarm) and Azure Service Fabric are indispensable for any production-ready microservice-based and for any multi-container application with significant complexity, scalability needs, and 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