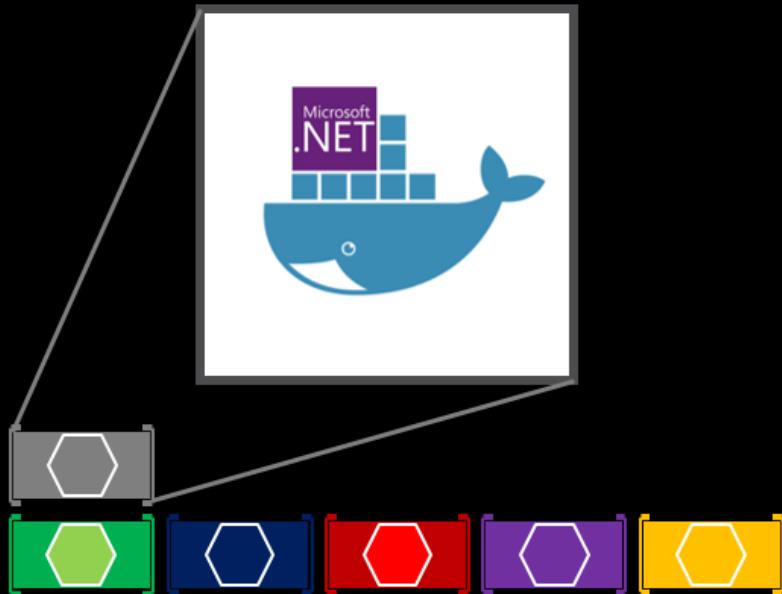




Architecting and Developing Containerized and Microservice based .NET Applications



Cesar de la Torre
Microsoft Corp.



PUBLISHED BY

DevDiv, .NET and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2016 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. All other marks are property of their respective owners.

Author:

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

Participants and reviewers (TBD):

John Gossman, Partner Software Eng, Azure product team, Microsoft

Jeffrey Richter, Partner Software Eng, Azure product team, Microsoft

Steve Lasker, Sr. PM, Visual Studio product team, Microsoft

Michael Friis, Product Manager, Docker Inc

Glenn Condron, Sr. PM, .NET product team, Microsoft

David Carmona, Principal PM Lead, .NET product team, Microsoft

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

Anand Chandramohan, Sr. Product Manager, Azure team, Microsoft

Scott Hunter, Partner Director PM, .NET product group, Microsoft

Contents

Summary	1
Purpose.....	1
Who should use this guide	1
How you can use this guide.....	1
Introduction to containers and Docker.....	2
What are containers?.....	2
What is Docker?.....	1
Comparing Docker containers with virtual machines.....	2
What is Container as a Service?.....	3
Basic Docker definitions	3
Basic Docker taxonomy: containers, images, and registries	5
Choosing between .NET Core and .NET Framework for Docker containers.....	6
Summary	6
When to choose .NET Core for Docker containers.....	6
When to choose .NET Framework for Docker containers.....	8
Decision table - .NET frameworks to use for Docker.....	10
What OS to target with .NET Containers.....	10
Official .NET Docker images	11
.NET Docker image optimizations per variant.....	12
Architecting containerized .NET applications with Docker and Azure	13
Vision.....	13
Architecting Docker applications.....	13
Common container design principles.....	13
Container equals a process	13
Monolithic applications.....	14
Monolithic application deployed as a container	15
Publishing a single Docker container app to Azure App Service.....	16
State and data in Docker applications	17
Service-oriented architecture applications.....	18
Microservices architecture	19
Data Sovereignty Per Microservice.....	21
Identifying domain-model boundaries per microservice	23

Challenges and solutions for Distributed Data Management.....	25
Stateless vs Stateful Microservices and advanced frameworks.....	28
API Gateway pattern vs. Direct Client-to-Microservice communication.....	29
Communication between microservices.....	32
Resiliency and high availability in Microservices	39
Health Reports and Diagnostics in Microservices.....	39
Orchestrating microservices and multi-container applications for high-scalability and availability... Docker clusters in Microsoft Azure	41
Azure Container Service	43
Azure Service Fabric	44
.....	46
Development process for Docker based applications	47
Vision	47
Development environment for Docker apps	47
Development tools choices: IDE or editor.....	47
.NET languages and frameworks for Docker containers	47
Development workflow for Docker apps	48
Workflow for developing Docker container based applications.....	48
Simplified workflow when developing containers with Visual Studio	59
Using PowerShell commands in Dockerfile to setup Windows Containers (Docker standard based)	59
Designing and developing multi-container and microservice based .NET applications	60
Vision	60
Designing a microservice oriented application	60
Application context	60
Development team context	61
Problem.....	61
Solution.....	61
Benefits.....	63
Drawbacks.....	64
External vs. Internal Architecture and Design Patterns	66
Creating a simple data-driven/CRUD microservice.....	67
Designing a simple data-driven/CRUD microservice.....	67
Implementing a simple CRUD microservice with ASP.NET Core.....	68
Creating microservices based on Domain-Driven Design (DDD) and Command-Query Separation (CQS) patterns.....	77
DDD vs. DDD patterns.....	77
Applying DDD and CQS patterns within a microservice	78
CQS and CQRS approaches in a DDD microservice	78
Implementing the Reads/Queries in a CQS/DDD microservice	81

Designing a Domain-Driven Design oriented microservice.....	83
Designing a microservice Domain-Model.....	87
Implementing a microservice's Domain Model with .NET Core and Entity Framework Core	92
Designing the Infrastructure-Persistence Layer.....	104
Implementing the Infrastructure-Persistence Layer with Entity Framework Core.....	107
Designing the microservice's Application Layer and Web API	108
Implementing the microservice's Application Layer and Web API	108
Implementing event based communication between microservices.....	110
Composing your multi-container application with docker-compose.yml.....	110
A database server running as a container	113
Testing ASP.NET Core services and web apps.....	116
Developing and deploying a .NET Framework application with monolithic deployment on Windows containers.....	120
TBD	120
Introduction to the Docker application lifecycle.....	121
Containers as the foundation for DevOps collaboration.....	121
Introduction to a generic E2E Docker application lifecycle workflow.....	122
Benefits from DevOps for containerized applications.....	123
Introduction to the Microsoft platform and tools for containers lifecycle	124
Conclusions.....	128
Key takeaways	128

Summary

Enterprises are increasingly adopting containers. The enterprise is realizing the benefits of cost savings, solution to deployment problems, and DevOps and production operations improvements that containers provide. Over the last years, Microsoft has been rapidly releasing container innovations to the Windows and Linux ecosystems – partnering with industry leaders like Docker and Mesosphere to deliver container solutions that help companies build and deploy applications at cloud speed and scale, whatever their choice of platform or tools.

Building containerized applications in an enterprise environment means more than just developing and running applications in containers. It means that you need to have an end-to-end lifecycle so you are capable of delivering applications through Continuous Integration, Testing, Continuous Deployment to containers, and release management supporting multiple environments, while having solid production management and monitoring systems.

Within the TBD ... This is all enabled through Microsoft tools and services for containerized Docker applications.

Commented [CDIT1]: Do not review this highlighted text. The whole Summary section has to change.

Purpose

This guide provides end-to-end guidance on the Docker application development lifecycle with Microsoft tools and services while providing an introduction to Docker development concepts for readers who might be new to the Docker ecosystem. This way, anyone can understand the global picture and start planning development projects based on Docker and Microsoft technologies/cloud.

This guide is complementary to the "Containerized Docker Application Lifecycle with Microsoft Platform and Tools" which focuses more on DevOps lifecycle, Tooling, IT Operations and Monitoring subjects.

Containerized Docker Application Lifecycle with Microsoft Platform and Tools

<https://aka.ms/dockerlifecyclebook>

Who should use this guide

The audience for this guide is mainly Development Leads, Architects, and IT Operations people who are new to Docker-based application development and would like to learn how to implement the whole Docker application lifecycle with Microsoft technologies and services in the cloud.

A secondary audience is technical decision makers who are already familiar to Docker but who would like to know the Microsoft portfolio of products, services, and technologies for the end-to-end Docker application lifecycle.

How you can use this guide

tbd

Introduction to containers and Docker

What are containers?

Containerization is an approach to software development in which an application and its versioned set of dependencies plus its environment configuration abstracted as deployment manifest files are packaged altogether (the container image), tested as a unit and finally deployed (the container or image instance) to the host Operating System (OS).

Similar to real-life shipping/freight containers (goods transported in bulk by ship, train, truck or aircraft), software containers are simply a standard unit of software that behaves the same on the outside regardless of what code, language and software/framework dependencies are included on the inside. This enables developers and IT Professionals to transport them across environments with none or little modification in the implementation regardless of the different configuration for each environment.

Containers isolate applications from each other on a shared operating system (OS). This approach standardizes application program delivery, allowing apps to run as Linux or Windows containers on top of the host OS (Linux or Windows). Because containers share the same OS kernel (Linux or Windows), they are significantly lighter than virtual machine (VM) images.

When running regular containers, the isolation is not as strong as when using plain VMs. If you need further isolation than the standard isolation provided in regular containers (like in regular Docker images), then, Microsoft offers an additional choice which is [Hyper-V containers](#). In this case, each container runs inside of a special virtual machine. This provides kernel level isolation between each Hyper-V container and the container host. Therefore, Hyper-V containers provide better isolation, with a little more overhead.

However, Hyper-V containers are less lightweight than regular Docker containers.

With a container oriented approach, you can eliminate most of the issues that arise when having inconsistent environment setups and the problems that come with them. The bottom line is that when running an app or service inside a container you avoid the issues that come with inconsistent environments.

Another important benefit when using containers is the ability to quickly instance any container. For instance, that allows to scale-up fast by instantly instancing a specific short term task in the form of a container. From an application point of view, instantiating an image (the container), should be treated in a similar way than instantiating a process (like a service or web app), although when running multiple instances of the same image across multiple host servers, you typically want each container (image instance) to run in a different host server/VM in different fault domains, for reliability.

In short and as the main takeaways, the main benefits provided by containers are Isolation, Portability, Agility, Scalability and Control across the whole application lifecycle workflow. But 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 any cloud or on-premises. Docker is also a [company](#) promoting and evolving this technology with a tight collaboration with cloud, Linux and Windows vendors, like Microsoft.

Docker is becoming the standard [unit of deployment](#) and is emerging as the de-facto standard implementation for containers as it is being adopted by most software platform and cloud vendors (Microsoft Azure, Amazon AWS, Google, etc.).

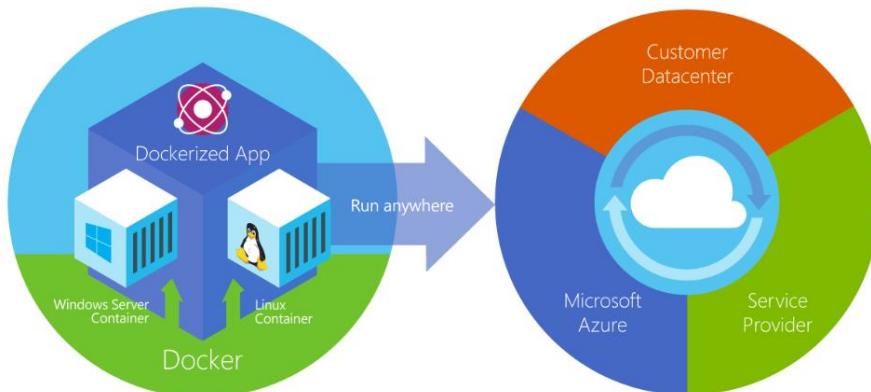


Figure 2-1. Docker deploys containers at all layers of the hybrid cloud

In regards supported Operating Systems, Docker containers can natively run on Linux and Windows.

You can use MacOS as another development environment alternative where you can edit code or run the Docker CLI, but containers do not run directly on MacOS. When targeting Linux containers, you will need a Linux host (typically a Linux VM) to run Linux containers. This applies to MacOS and Windows development machines.

To host containers, and provide additional developer tools, Docker ships [Docker for Mac](#) and [Docker for Windows](#). These products install the necessary VM to host Linux containers.

Related to [Windows Containers](#), there are two types or 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 all containers running on the host.

Hyper-V Containers – expands 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.

Comparing Docker containers with virtual machines

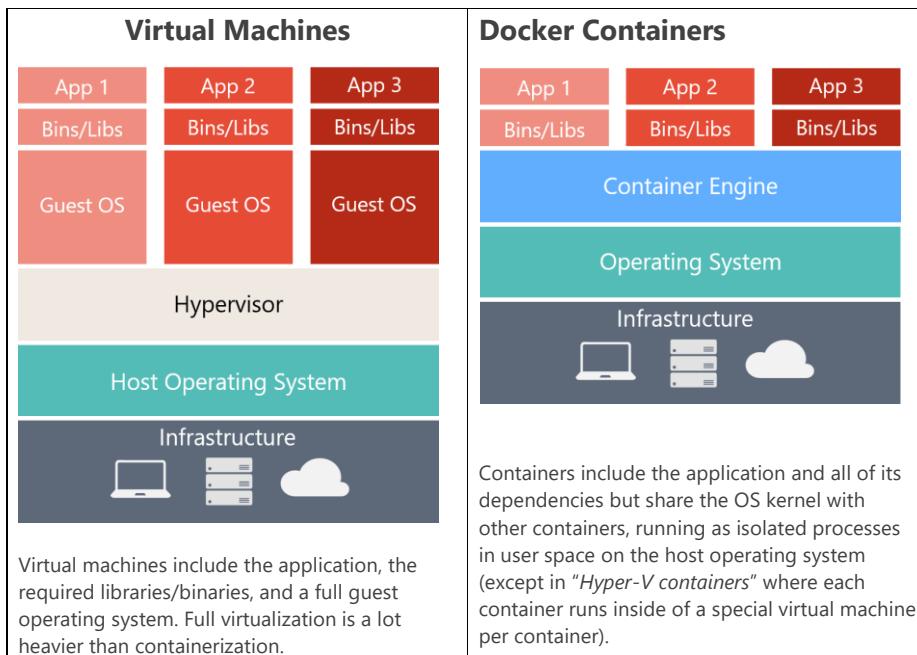


Figure 2-2. Comparison of traditional virtual machines to Docker containers

From an application architecture point of view, each Docker container is usually a single process which could be a whole app (monolithic app) or a single service or microservice. The benefits you get when your application or service process runs inside a Docker container is that it also includes all its dependencies, so its deployment on any environment that supports Docker is usually assured to be done right.

Since Docker containers are sandboxes running on the same shared OS kernel it provides very important benefits. They are easy to deploy and start fast. As a side effect of running on the same kernel, you get less isolation than VMs but also using far fewer resources. Because of that, containers start fast.

Docker also is a way to package up an app/service and push it out in a reliable and reproducible way. So, you can say that Docker is a technology, but also a philosophy and a process.

Coming back to the container's benefits, when using Docker, you won't get the typical developer's statement "*it works on my machine*". But you 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 do it on all the deployment targets (Dev/QA/Staging/Production, etc.).

What is Container as a Service?

Container as a Service (CaaS) is an IT managed and secured application environment of infrastructure and content provided as a service (elastic and pay as you go, similar to the basic cloud principles), with no upfront infrastructure design, implementation and investment per project, where developers can (in a self-service way) build, test and deploy applications and IT operations can run, manage and monitor those applications in production.

From its original principles, it is partially similar to Platform as a Service (PaaS) in the way that resources are provided "as a service" from a pool of resources. What's different in this case is that the unit of software is now measurable and based on containers. Images (per version) are immutable.

In regards host OS related updates, it usually can be responsibility of the person/organization owning the container image; however the service provider might also help to update the Linux/Windows kernel and Docker engine version at the host level.

Either PaaS or CaaS can be supported in public clouds (like Microsoft Azure, Amazon AWS, Google, etc.) or on-premises.

Basic Docker definitions

The following are the basic definitions anyone needs to understand before getting deeper into Docker. For further definitions, an extensive Docker Glossary is provided by Docker here:

<https://docs.docker.com/v1.11/engine/reference/glossary/>

Docker image: Docker images are the basis of containers. An Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems stacked on top of each other. An image does not have state and it never changes as it's deployed to various environments.

Container: A container is a runtime instance of a Docker image. A Docker container consists of: A Docker image, an execution environment and a standard set of instructions. When scaling a service, you would instance multiple containers from the same image. Or, in a batch job, instance multiple containers from the same image, passing different parameters to each instance. A container "contains" something singular, a single process, like a service or web app. It is a 1:1 relationship.

Tag: A tag is a label applied to a Docker image in a repository. Tags are how various images in a repository are distinguished from each other. They are commonly used to distinguish between multiple versions of the same image.

Dockerfile: A Dockerfile is a text document that contains instructions to build a Docker image.

Build: build is the process of building Docker images using a Dockerfile. The build uses a Dockerfile and a "context". The context is the set of files in the directory in which the image is built. Builds can be done with commands like "docker build" or "docker-compose" which incorporates additional information such as the image name and tag.

Repository: A collection of related images, differentiated by a tag that would differentiate the historical version of a specific image. Some repos contain multiple variations of a specific image, such as the SDK, runtime/fat, thin tags. As Windows containers become more prevalent, a single repo can contain platform variants, such as a Linux and Windows image.

Registry: A [Registry](#) is a hosted service containing repositories of images which responds to the Registry API. The default registry (from Docker as an organization) can be accessed using a browser at [Docker Hub](#) or using the Docker search command. Therefore, a Registry usually contains many Repositories from multiple teams. As companies will want to keep their images private, and network close to their deployment infrastructure, companies will instance private registries in their environment to maintain their apps and control over their base images.

Docker Hub: The Docker Hub is a centralized public resource for working with Docker and its components. It provides the following services: Docker image hosting, User authentication, Automated image builds plus work-flow tools such as build triggers and web hooks, Integration with GitHub and Bitbucket. Docker Hub is the public instance of a registry. Equivalent to the public GitHub compared to GitHub enterprise where customers store their code in their own environment.

Azure Container Registry: Centralized public resource for working with Docker Images and its components in Azure, a registry network-close to your deployments with control over access, making possible to use your Azure Active Directory groups and permissions.

Docker Trusted Registry: [Docker Trusted Registry \(DTR\)](#) is the enterprise-grade image storage solution from Docker. You install it behind your firewall so that you can securely store and manage the Docker images you use in your applications. Docker Trusted Registry is a sub-product included as part of the Docker Datacenter product.

Docker for Windows and Mac: The local development tools for building, running and testing containers locally. "Docker for x", indicates the target developer machine. Docker for Windows provides both Windows and Linux container development environments.

"Docker for Windows and Mac" deprecates "Docker Toolbox" which was based on Oracle VirtualBox. Docker for Windows is now based on [Hyper-V](#) VMs (Linux or Windows). Docker for Mac is based on Apple Hypervisor framework and [xhyve](#) hypervisor which provides a Docker-ready virtual machine on Mac OS X.

Compose: Compose is a tool for defining and running multi container applications. With compose, you define a multi-container application in a single file, then spin your application up in a single command which does everything that needs to be done to get it running. Docker-compose.yml files are used to build and run multi container applications, defining the build information as well the environment information for interconnecting the collection of containers.

Cluster: A Docker cluster pools together multiple Docker hosts and exposes them as a single virtual Docker host so it is able to scale-up to many hosts very easily. Examples of Docker clusters can be created with Docker Swarm, Mesosphere DC/OS, Google Kubernetes and Azure Service Fabric. If using Docker Swarm you typically call that "a swarm" instead of "a cluster".

Orchestrator: A Docker Orchestrator simplifies management of clusters and Docker hosts. These Orchestrators enable users to manage their images, containers and hosts through a user interface, either a CLI or UI. This interface allows users to administer container networking, configurations, load balancing, service discovery, High Availability, Docker host management and a much more.

An orchestrator is responsible for running, distributing, scaling and healing workloads across a collection of nodes. Typically, Orchestrator products are the same products providing the cluster infrastructure like Mesosphere DC/OS, Kubernetes, Docker Swarm and Azure Service Fabric.

Basic Docker taxonomy: containers, images, and registries

Figure 2-3 shows how each basic component in Docker relates to each other as well as the multiple Registry offerings from vendors.

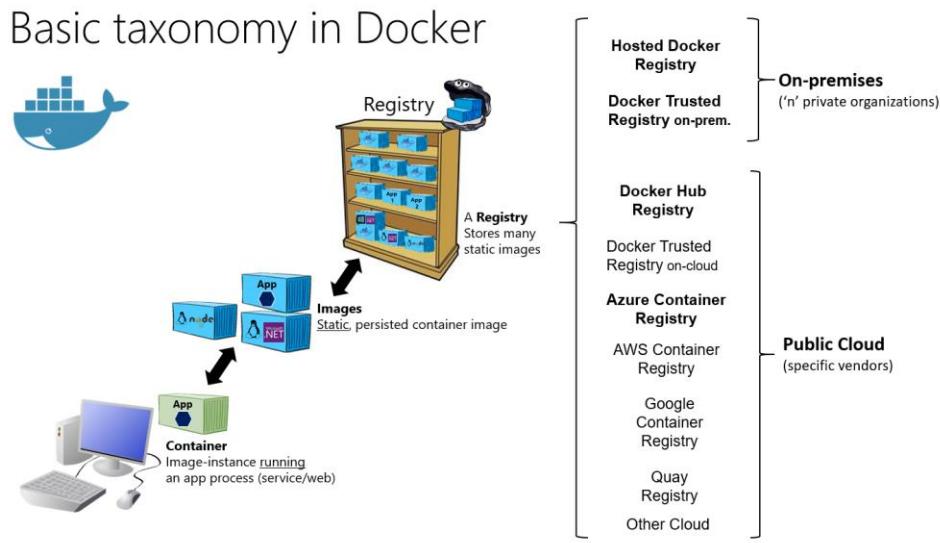


Figure 2-3. Taxonomy of Docker terms and concepts

As mentioned in the definitions section, a **container** is one or more runtime instances of a Docker image that usually will contain a single app/service. The container is considered the live artifact being executed in a development machine or the cloud or server.

An **image** is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. An image typically contains a union of layered filesystems (deltas) stacked on top of each other. An image does not have state and it never changes.

A **registry** is a service containing repositories of images from one or more development teams. Multiple development teams may also instance multiple registries. The default registry for Docker is the public "Docker Hub" but you will likely have your own private registry network close to your orchestrator to manage and secure your images, and reduce network latency when deploying images.

The beauty of the images and the registry resides on the possibility for you to store static and immutable application bits including all their dependencies at OS and frameworks level so they can be versioned and deployed in multiple environments providing a consistent deployment unit.

You should use a private registry (an example of use of *Azure Container Registry*) if you want to:

- Tightly control where your images are being stored
- Reduce network latency between the registry and the deployment nodes
- Fully own your images distribution pipeline
- Integrate image storage and distribution tightly into your in-house development workflow

Choosing between .NET Core and .NET Framework for Docker containers

Summary

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.

You should use .NET Core for your containerized Docker server application when:

- You have cross-platform needs. You want to use Linux containers and Windows containers.
- Your application architecture is based on microservices.
- You need best-in-class high performance and hyper-scale.
- You need side by side of .NET versions by application within the same host.

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 third-party .NET libraries or NuGet packages not available for .NET Core.
- You need to use .NET technologies that are not available for .NET Core.
- You need to use a platform that doesn't support .NET Core.

When to choose .NET Core for Docker containers

The following is a more detailed explanation of the previously-stated reasons for picking .NET Core.

Cross-platform needs

Clearly, if your goal is to have an application (web/service) that should be able to run across multiple platforms supported by Docker (Linux and Windows), the right choice is to use .NET Core as .NET Framework supports only Windows.

By itself, .NET Core also supports MacOS in addition to Windows and Linux, as your development workstation, but when deploying containers to a Docker host, that host currently has to be based on a Linux or Windows (for instance, in a development environment, it could be a Linux VM running on a Mac).

Visual Studio provides an Integrated Development Environment (IDE) for Windows and Mac (Visual Studio for Mac which is an evolution of Xamarin Studio). You can also use Visual Studio Code on macOS, Linux and Windows which fully support .NET Core, including IntelliSense and debugging. You can also target .NET Core with most third-party editors like Sublime, Emacs, VI and can get editor IntelliSense using the open source Omnisharp project. Ultimately, you could also avoid any code editor and directly use the .NET Core command-line tools, available for all supported platforms.

The “by-default” selection when targeting containers in new projects (“green-field”)

Containers are commonly used in conjunction with a microservices architecture, although they can also be used to containerize web apps or services which follow any architectural pattern. You will be able to use the .NET Framework for Windows containers, but the modularity and lightweight nature of .NET Core makes it perfect for containers. When creating and deploying a container the size of its image is far smaller with .NET Core than .NET Framework. Because .NET Core is cross-platform, you can deploy server apps to Linux Docker containers, for example.

Microservices architecture

.NET Core is the best candidate if you are embracing a microservices oriented system composed of multiple independent, dynamically scalable, stateful or stateless microservices. .NET Core is lightweight and its API surface can be minimized to the scope of the microservice. A microservices architecture also allows you to mix technologies across a service boundary, enabling a gradual embrace of .NET Core for new microservices that live in conjunction with other microservices or services developed with Node.js, Python, Java, Ruby, or other technologies.

The infrastructure platforms you could use when targeting microservices and containers are many.

For large and complex microservice systems being deployed as Linux containers, Azure Container Service with its multiple orchestrator offering (Mesos DC/OS, Kubernetes and Docker Swarm) is a great and mature choice. You can also use Azure Service Fabric for Linux (currently in Preview) which also supports Docker Linux containers.

For large and complex microservice systems being deployed as Windows containers, most orchestrators are currently in a less mature state, but you will be able to use Azure Service Fabric supporting Windows containers soon as well as Azure Container Service. However, Azure Service Fabric has a long experience running mission-critical Windows applications (without Docker) in comparison to other orchestrators.

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

A need for high performance and scalable systems

When your container-based system needs the best possible performance and scalability, .NET Core and ASP.NET Core are your best options. ASP.NET Core outperforms ASP.NET by a factor of 10, 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 you'd be able to run your system with a much lower number of servers/VMs, ultimately saving costs in infrastructure and hosting.

A need for side by side of .NET versions per application level within the same host

If you want to be able to install applications with dependencies on different versions of frameworks in .NET within the same machine, you need to use .NET Core, which provides 100% side-by-side. Easy side-by-side installation of different versions of .NET Core on the same machine allows you to have multiple services on the same server, each of them on its own version of .NET Core, eliminating risks and saving money in application upgrades and IT operations.

[When to choose .NET Framework for Docker containers](#)

While .NET Core offers significant benefits for new applications and application patterns, the .NET Framework will continue to be a good choice for many existing scenarios and as such, it won't be replaced by .NET Core for all containerized server applications.

Current .NET Framework application directly migrated to a Docker container

The reason why you could want to use Docker containers could be other than "targeting microservices". It could also be simply because you want to improve safety of your DevOps workflow and eliminate deployment issues caused by non-existing dependencies in production environments. In this case, even when the deployment-type of your application might be monolithic, it makes sense to use Docker and Windows containers for your current .NET Framework applications.

In addition to that and in most cases, you won't need to migrate your existing applications to .NET Core. Instead, a recommended approach is to use .NET Core as you extend an existing application, such as writing a new service in ASP.NET Core.

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

Libraries are quickly embracing .NET Standard, which enables sharing code across all .NET flavors including .NET Core. With .NET Standard 2.0 this will be even easier, as the .NET Core API surface will become significantly bigger and .NET Core applications can directly use existing .NET Framework libraries. This transition won't be immediate, though, so we recommend checking the specific libraries required by your application before making a decision one way or another.

However, take into account that whenever you run a library/process based on the traditional .NET Framework, because of its dependencies on Windows, the container image used for that application/service will need to be based on a Windows Container image.

A need to use .NET technologies not available for .NET Core

Some .NET Framework technologies are not available in .NET Core 1.1. Some of them will be available in later .NET Core releases, but others don't apply to the new application patterns targeted by .NET

Core and may never be available. The following list shows the most common technologies not found in .NET Core 1.1:

- ASP.NET Web Forms applications: ASP.NET Web Forms is only available on the .NET Framework, so you cannot use ASP.NET Core / .NET Core for this scenario. Currently there are no plans to bring ASP.NET Web Forms to .NET Core.
- ASP.NET Web Pages applications: ASP.NET Web Pages are not included in ASP.NET Core 1.1, although it is planned to be included in a future release as explained in the [.NET Core roadmap](#).
- ASP.NET SignalR server/client implementation. At .NET Core 1.1 release timeframe (November 2016), ASP.NET SignalR is not available for ASP.NET Core (neither client or server), although it is planned to be included in a future release as explained in the .NET Core roadmap. Preview state is available at the [Server-side](#) and [Client Library](#) GitHub repositories.
- WCF services implementation. Even when there's a [WCF-Client library](#) to consume WCF services from .NET Core, as of November 2016, WCF server implementation is only available on the .NET Framework. This scenario is not part of the current plan for .NET Core but it's being considered for the future.
- 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 the .NET Framework and there are no plans to bring them to .NET Core.
- Language support: Visual Basic and F# don't currently have tooling support .NET Core, but both will be supported in Visual Studio 2017 and later versions of Visual Studio.

In addition to the official [.NET Core roadmap](#), there are other frameworks to be ported to .NET Core - For a full list, take a look at CoreFX issues marked as [port-to-core](#). Please note that this list doesn't represent a commitment from Microsoft to bring those components to .NET Core — they are simply capturing the desire from the community to do so. That being said, 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](#).

A need to use a platform 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 not yet available for consumption on .NET Core. This is a transitional circumstance, as all of Azure services will use .NET Core (For example, the [Azure DocumentDB SDK for .NET Core](#) was released as preview on November 16th 2016). In the meantime, you can always use the equivalent REST API instead of the client SDK.

Decision table - .NET frameworks to use for Docker

As a recap, below is a summary decision table depending on your architecture or application type and the server operating system you are targeting for your Docker containers.

Take into account that if you are targeting Linux containers you will need Linux based Docker hosts (VMs or Servers) and in a similar way, if you are targeting Windows containers you will need Windows Server based Docker hosts (VMs or Servers).

Architecture / App Type	Linux containers	Windows containers
Microservices	.NET Core	.NET Core
Monolithic deployment App	.NET Core	.NET Framework .NET Core
Best-in-class performance and scalability	.NET Core	.NET Core
Windows Server "brown-field" migration to containers	--	.NET Framework
Containers "green-field"	.NET Core	.NET Core
ASP.NET Core	.NET Core	.NET Core recommended .NET Framework is possible
ASP.NET 4 (MVC 5, Web API 2)	--	.NET Framework
SignalR services	.NET Core in upcoming releases	.NET Framework .NET Core in upcoming releases
WCF, WF and other traditional frameworks	--	.NET Framework
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 "by design" 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 just few of them are targeted in the official .NET Docker images (like Debian and Alpine). In Windows you can use Windows Server Core or Nano Server which provide different characteristics (like IIS vs. Kestrel, etc.) that might be needed by .NET Framework or .NET Core.

In the image X-X you can see the recommended OS version depending on the .NET frameworks.

What OS to target with .NET containers

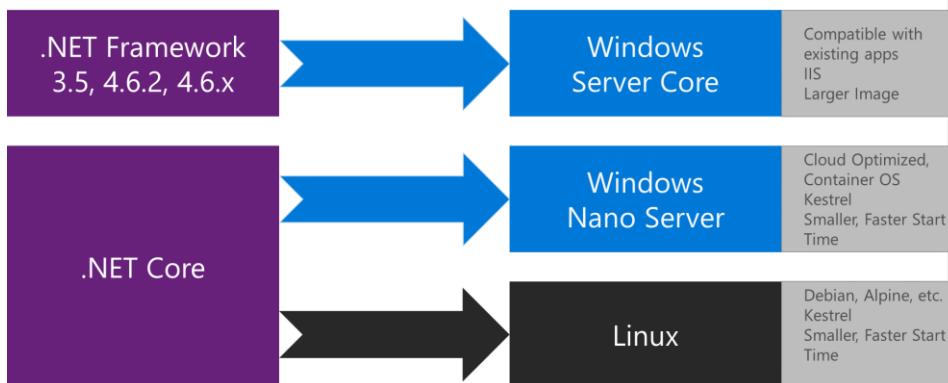


Figure X-X: OS to Target depending on .NET frameworks

However, you could also create your own Docker image from scratch in cases like when you want to use a different Linux distro or an image with versions not provided by Microsoft's images like ASP.NET Core running on traditional .NET Framework and Windows Server Core.

When setting the image name into your dockerfile file you can select the Operating System and version depending on the tag you use, as in the examples below.

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 and publicly available at [Docker Hub](#) within Microsoft's repositories.

Each repository can contain multiple images depending on specific .NET versions plus specific OS and versions (Linux Debian, Linux Alpine, Windows Nano Server, Windows Server Core, etc.).

Microsoft vision for .NET repositories is to have granular/focused repos, where a repo represents a specific scenario or workload. For instance, the [microsoft/aspnetcore](#) images should be used for ASP.NET Core containers as that image provides additional optimizations for ASP.NET Core.

On the other hand, the .NET Core images ([microsoft/dotnet](#)) is aimed to be used for "console" apps based on .NET Core like batch processes, Azure WebJobs and other scenarios based on .NET Core "console", because having the ASP.NET Core stack in this smaller image would be a high tax in regards increasing the "surface area" as it would be a bigger image

In any case, most image repos will provide extended tags so you can select not just a specific framework version but also the chose Operating System (Linux distro or Windows version), since those version don't change the application level scenario.

For further information about the official .NET Docker images provided by Microsoft, go to the below references.

Official .NET Docker Images reference
Summary on Official .NET Docker Images – When to use each image
https://aka.ms/dotnetdockermages

.NET Docker image optimizations per variant

When building Docker images for developers, we focused on three main scenarios:

- Images used to develop .NET Core apps
- Images used to build .NET Core apps
- Images used to run .NET Core apps

Why three images? When developing, building and running containerized applications, we have different priorities.

Development: How fast can you iterate changes, and the ability to debug the changes. The size of the image isn't as important, rather can you make changes to your code and see them quickly. Some of our tools, like yo docker for use in VS Code use this image during development time.

Build: What's needed to compile your app. This includes the compiler and any other dependencies to optimize the binaries. This image isn't the image you deploy, rather 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, or build environment. For instance, rather than installing all the dependencies directly on a build agent, the build agent would instance a build image to compile the application with all the dependencies required to build the app contained within the image. Your build agent only needs to know how to run this Docker image.

Production: How fast you can deploy and start your image. This image is small so it can quickly travel 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 immutable Docker model, there's no need for dynamic compilation of code. The content you place in this image would be limited to the binaries and content needed to run the application. For example, the published output using dotnet publish which contains the compiled binaries, images, js and .css files. Over time, you'll see images that contain pre-jitted packages.

Though there are multiple versions of the .NET Core image, they all share one or more layers. The amount of disk space needed to store or the delta to pull from your registry is much smaller than the whole because all of the images share the same base layer and potentially others.

This is why when exploring most of the .NET image repositories at Docker Hub you can find multiple image versions based on tags like:

microsoft/dotnet:1.1-runtime	.NET Core 1.1, with runtime-only, on Linux
microsoft/dotnet:1.1.0-sdk-msbuild	.NET Core 1.1 with SDK included, on Linux

Architecting containerized .NET applications with Docker and Azure

Vision

Architect and design scalable solutions with Docker in mind.

There are many great-fit use cases for containers, not just for microservices oriented architectures but also when you simply have regular services or web applications to run and you want to reduce frictions between development and production environment deployments.

Architecting Docker applications

In the first section of this document you already got the fundamental concepts regarding containers and Docker. That information is the basic level of information to get started. But enterprise applications can be complex and composed by multiple services instead of a single service/container. For those optional use cases, you need to know further architectural approaches like Service Orientation and the more advanced Microservices concepts and container orchestration concepts. The scope of this document is not limited to microservices but to any Docker application lifecycle, therefore, it does not drill down deeply into microservices architecture because you can also use containers and Docker with regular Service Orientation, background tasks/jobs or even with monolithic application deployment approaches.

However, before getting into the application lifecycle and DevOps, it is important to know what and how you are going to design and construct your application and what are the design choices.

Common container design principles

Container equals a process

In the container model, a container represents a single process. By defining a container as a process boundary, you start to create the primitives used to scale, or batch off processes. When running a Docker container, you'll see an `ENTRYPOINT` definition. This defines the process and the lifetime of the container. When the process completes, the container lifecycle ends. There are long running processes, like web servers and short lived processes like batch jobs, which may have been

implemented as Azure [WebJobs](#). If the process fails, the container ends, and the orchestrator takes over. If the orchestrator was told to keep 5 instances running, and one fails. The orchestrator will instance another container to replace the failed process. In a batch job, the process is started with parameters. When the process completes, the work is complete.

You may find a scenario where you may want multiple processes running in a single container. In any architecture document, there's never a "never", nor is there always an "always". For scenarios requiring multiple processes, a common pattern is to use <http://supervisord.org/>

Monolithic applications

In this scenario, you are building a single and monolithic-deployment based Web Application or Service and deploying it as a container. Within the application, it might not be monolithic but structured in several libraries, components or even layers (Application layer, Domain layer, Data access layer, etc.). Externally it is a single container like a single process, single web application or single service.

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

You can include multiple components/libraries or internal layers within each container, as illustrated in Figure 5-1. But, following the container principal of "*a container does one thing, and does it in one process*", the monolithic pattern might be a conflict.

Monolithic Containerized application

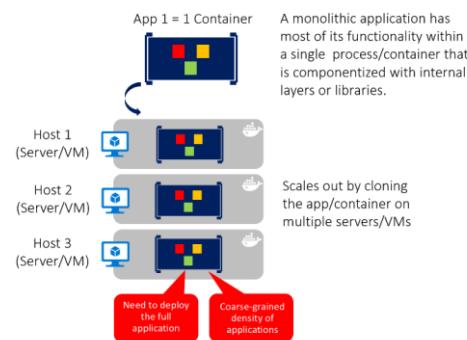


Figure 5-1. Monolithic application architecture example

The downside of this approach comes if/when the application grows, requiring it to scale. If the entire application scaled, it's not really a problem. However, in most cases, a few parts of the application are the choke points requiring scaling, while other components are used less.

Using the typical eCommerce example; what you likely need is to scale the product information component. Many more customers browse products than purchase. More customers use their basket than use the payment pipeline. Fewer customers add comments or view their purchase history. And you likely only have a handful of employees, in a single region, that need to manage the content and marketing campaigns. By scaling the monolithic design, all the code is deployed multiple times.

In addition to the scale everything problem, changes to a single component require complete retesting of the entire application, and a complete redeployment of all the instances.

The monolithic approach is common, and many organizations are developing with this architectural approach. Many are having good enough results, while others are hitting limits. Many designed their applications in this model, because the tools and infrastructure were too difficult to build service oriented architectures (SOA), and didn't see the need. Until the app grew.



Figure 5-2. Host running multiple apps/containers

From an infrastructure perspective, each server can run many applications within the same host and have an acceptable ratio of efficiency in your resources usage, as shown in Figure 5-2.

Deploying monolithic applications in Microsoft Azure can be achieved using dedicated VMs to each instance. Using [Azure VM Scale Sets](#), you can easily scale the VMs. [Azure App Services](#) can run monolithic applications and easily scale instances without having to manage the VMs. Since 2016, Azure App Services can run single instances of Docker containers as well, simplifying the deployment. And using Docker, you can deploy a single VM as a Docker host, and run multiple instances. Using the Azure balancer, as shown in the Figure 5-3, you can manage scaling.

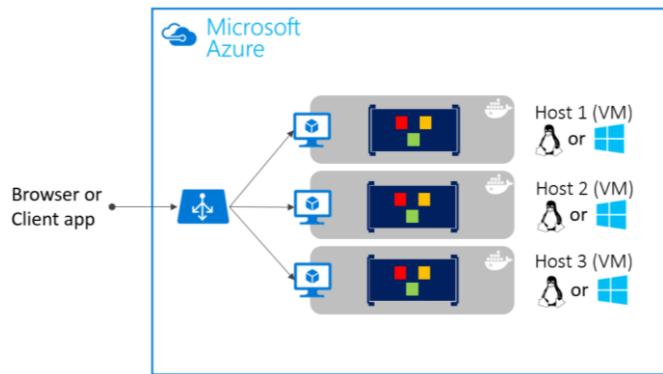


Figure 5-3. Multiple hosts scaling-out a single Docker application

The deployment to the various hosts can be managed with traditional deployment techniques. The Docker hosts can be managed with commands like "`docker run`" performed manually, through automation such as Continuous Delivery (CD) pipelines, to be explained later in this document.

Monolithic application deployed as a container

There are benefits of using containers to manage monolithic deployments. Scaling the instances of containers is far faster and easier than deploying additional VMs. While VM Scale Sets are a great feature to scale VMs, which are required to host your Docker containers, they take time to instance. When deployed as app instances, the configuration of the app is managed as part of the VM.

Deploying updates as Docker images are far faster and network efficient. The Vn instances can be instanced on the same hosts as your Vn-1 instances, eliminating additional costs of additional VMs. Docker Images typically start in seconds, speeding rollouts. Tearing down a Docker instance is as easy as “`docker stop`” command, typically completing in less than a second.

As containers are inherently immutable, by design, you never worry about corrupted VMs as update script forgot to account for some specific configuration or file left on disk.

While monolithic apps can benefit from Docker, we’re only touching on the tips of the benefits. The larger benefits of managing containers comes from deploying with container orchestrators which manage the various instances and lifecycle of each container instance. Breaking up the monolithic application into sub systems which can be scaled, developed and deployed individually are your entry point into the realm of microservices.

Publishing a single Docker container app to Azure App Service

Either if you want to get a quick validation of a container deployed to Azure or because the app is simply a single container app, Azure App Services provides a great way to provide scalable single container services.

Using Azure App Service is very simple and easy to get started as it provides great git integration to take your code, build it in Visual Studio and directly deploy it to Azure. But, traditionally (with no Docker), if you needed other capabilities/frameworks/dependencies that aren’t supported in App Services you needed to wait for it until the Azure team updates those dependencies in App Service or switched to other services like Service Fabric, Cloud Services or even plain VMs where you have further control and you can install a required component/framework for your application.

Now (announced at Microsoft Connect 2016, November 2016) and as shown in Figure 5-4 when using Visual Studio 2017, containers support in Azure App Service gives you the ability to include whatever you want in your app environment. If you added a dependency to your app, since you are running it in a container, you get the capability of including those dependencies in your dockerfile or Docker image.

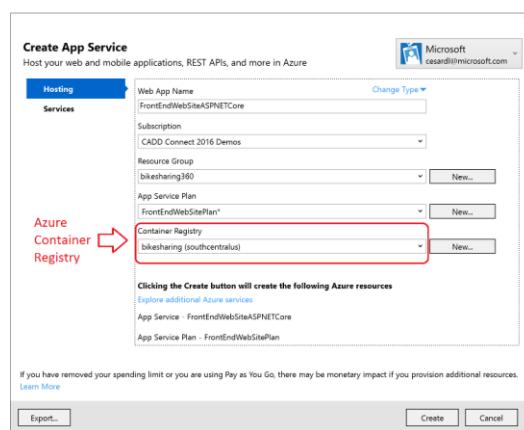


Figure 5-4. Publishing a Container to Azure App Service from Visual Studio

As also shown in figure 5-4, the publish flow pushes an image through a Container Registry which can be the Azure Container Registry (a registry near to your deployments in Azure and secured by Azure Active Directory groups and accounts) or any other Docker Registry like Docker Hub or on-premises registries.

State and data in Docker applications

A primitive of containers are immutability. When comparing to a VM, they don't disappear as a common occurrence. A VM may fail in various forms from dead processes, overloaded CPU, a full or failed disk. However, we expect the VM to be available and RAID drives are commonplace to assure drive failures maintain data.

However, containers are thought to be instances of processes. A process doesn't maintain durable state. While a container can write to its local storage, assuming that instance will be around indefinitely would be equivalent to assuming a single copy memory will be durable. Containers, like processes, should be assumed to be duplicated, killed or when managed with a container orchestrator, they may get moved.

Docker uses a feature known as an overlay file system to implement a copy-on-write process that stores any updated information to the root file system of a container, compared to the original image on which it is based. These changes are lost if the container is subsequently deleted from the system. A container therefore does not have persistent storage by default. While it's possible to save the state of a container, designing a system around this would be in conflict with the premise of container architecture.

To manage persistent data in Docker applications, there are common solutions:

- [Data volumes](#) which mount to the host as noted above
- [Data volume containers](#) which provide shared storage across containers, using an external container that may cycle
- [Volume Plugins](#) which mount volumes to remote locations, providing long term persistence
- Remote data sources like SQL, NO-SQL databases or cache services like Redis.
- [Azure Storage](#) which provides geo distributable PaaS storage, providing the best of containers as long term persistence.

Data volumes are specially-designed directories within one or more containers that bypasses the [Union File System](#). Data volumes are designed to persist data, independent of the container's life cycle. Docker therefore never automatically delete volumes when you remove a container, nor will it "garbage collect" volumes that are no longer referenced by a container. The data in any volume can be freely browsed and edited by the host operating system, and is just another reason to use data volumes sparingly.

Data volume container. A [data volume container](#) is an improvement over regular data volumes. It is essentially a dormant container that has one or more data volumes created within it (as described above). The data volume container provides access to containers from a central mount point. The benefit of this method of access is that it abstracts the location of the original data, making the data container a logical mount point. It also allows "application" containers accessing the data container volumes to be created and destroyed while keeping the data persistent in a dedicated container.

As shown in the Figure 5-5, regular Docker volumes can be placed on storage out of the containers themselves but within the host server/VM physical boundaries. **Docker volumes don't have the ability to use a volume from one host server/VM to another.**

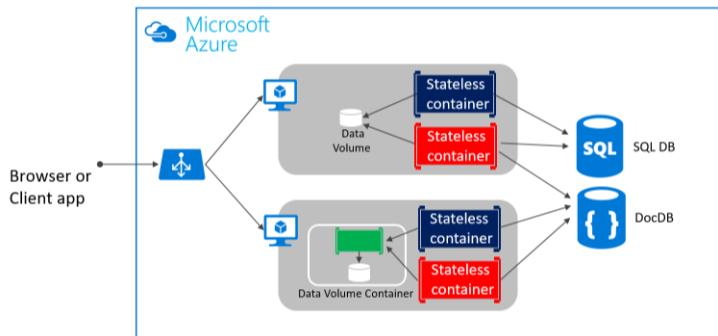


Figure 5-5. Data Volumes and external data sources for containers apps/containers

Due to the inability to manage data shared between containers that run on separate physical hosts, it is not recommended to use volumes for business data unless the Docker host is a fixed host/VM, because when using Docker containers in an orchestrator, containers are expected to be moved from one to another host depending on the optimizations to be performed by the cluster. Therefore, regular data volumes are a good mechanism to work with trace files, temporal files or any similar concept that won't impact the business data consistency if/when your containers are moved across multiple hosts.

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

Remote data sources and cache like SQL DB, DocDB or a remote cache like Redis would be used the same way as developing without containers. This is a proven ways to store business application data.

Service-oriented architecture applications

Service-oriented architecture (SOA) was an overused term and meant so many different things to different people. But as minimum and common denominator, SOA or Service Orientation mean that you structure the architecture of your application by decomposing it in multiple services (most commonly as Http services) that can be classified in different types like sub-systems or in other cases as tiers.

Those services can nowadays be deployed as Docker containers so it also solves deployment issues as all the dependencies are included within the container image. However, when you need to scale-out Service Oriented applications, you might have challenges if you are deploying based on single instances. This is where a Docker clustering software or orchestrator will help you out, as explained in later sections when describing deployment approaches for microservices.

Docker containers are useful for both, traditional SOA architectures and the more advanced microservices architectures. In regards architecture patterns and implementation, this paper is focusing on microservices because a SOA approach means you are using a sub-set of the requisites 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

Microservices is a hot buzzword at the moment. While there are many presentations and conference talks about the subject, a lot of developers remain confused. A common question is: "Isn't this just another service-oriented architecture (SOA) or Domain-Driven Design (DDD) approach?"

Certainly, many of the techniques used in the microservices approach derive from the experiences of developers in SOA and DDD. You can think of microservices as "SOA done right," with principles like autonomous services, Bounded-Context pattern and event-driven all having their roots in SOA and DDD.

As the name implies, microservices architecture is an approach to build a server application as a set of small services, each service running in its own process and communicating with each other via protocols such as HTTP and WebSockets. Each microservice implements specific, end-to-end domain/business capabilities within a certain Bounded-Context per service and must be developed autonomously and deployed independently by automated mechanisms. Finally, each service should own its related domain data model and domain logic (sovereignty and decentralized data management), and can employ different data storage technologies (SQL, No-SQL) and different programming languages per microservice.

What size should a microservice have? In service development, autonomy is much more important than size. It is much easier to reduce a monolithic service down to autonomous components than it is to unpick a web of complex service integrations. So, think about autonomous services within a context boundary rather than trying to create the smallest service possible, which would be bad in some cases.

Why microservices? In short, "agility in the long term". Microservices enable superior maintainability in large, complex and highly scalable systems by designing applications based on many independently deployable services that allow for granular release planning.

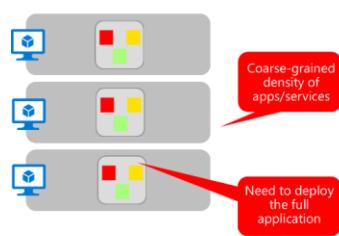
As an additional benefit, microservices can scale out independently. Instead of having giant monolithic application blocks that you must scale out at once, you can instead scale out specific microservices. That way, just the specific functional area that needs more processing power or network bandwidth to support demand can be scaled, rather than scaling out other areas of the application that really don't need it.

Architecting fine-grained microservice applications enables continuous integration and continuous development practices, and accelerates delivery of new functions into the application. Fine-grain decomposition of applications also lets you run and test microservices in isolation, and to evolve microservices independently while maintaining rigorous contracts among them. As long as you don't break the contracts or interfaces, you can change any microservice implementation under the hood and add new functionality without breaking the other microservices that depend on it.

As [Figure-X-X](#) shows, with the microservices approach it's all about efficiency for agile changes and rapid iteration because you're able to change specific, small portions of large, complex and scalable applications.

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

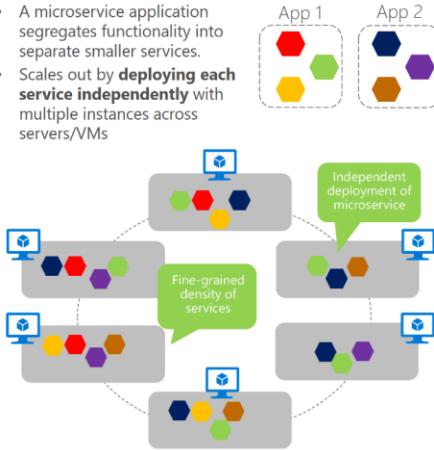


Figure X-X. Microservices approach compared to monolithic deployment approach

Before you go into production with a microservices system, you need to ensure that you have key prerequisites in place:

- Rapid Provisioning
- Basic Monitoring
- Rapid Application Deployment
- Devops Culture

References – Microservice architecture

Microservices: An application revolution powered by the cloud – By Mark Russinovich

<https://azure.microsoft.com/en-us/blog/microservices-an-application-revolution-powered-by-the-cloud/>

Understanding microservices

<https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-overview-microservices>

Microservices patterns – By Martin Fowler

<http://www.martinfowler.com/articles/microservices.html>

<http://martinfowler.com/bliki/MicroservicePrerequisites.html>

Data Sovereignty Per Microservice

An important rule to comply in this approach 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 sub-systems or microservices. Consider enterprise applications, where customer relationship management (CRM) applications, transactional purchase subsystems and customer support subsystem each call on unique customer entity attributes and data and employ a different bounded context.

This principle is similar in DDD where each Bounded-Context (BC), which is a pattern comparable to a subsystem/service, must own its domain-model (data+logic). Each DDD Bounded-Context would correlate to a different microservice.

On the other hand, the traditional (or monolithic) approach used in many applications is to have a single (or few) centralized database, often a normalized SQL database, for the whole application and all its internal subsystems, as shown in Figure X-X.

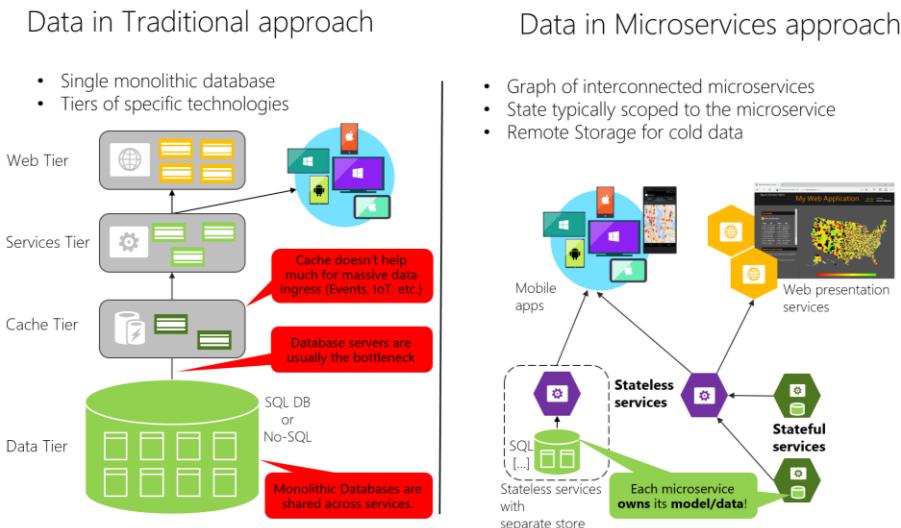


Figure X-X. Data Sovereignty Comparison: Microservices vs. Monolithic DB

The centralized database approach looks initially simpler and seems to enable re-use 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 include attributes and columns that simply 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. First, ACID transactions and second SQL language, but both across all the tables and data related to your app. That provides a very simple way to easily write a query that combines data from multiple tables.

However, data access becomes much more complex when we move to a microservices architecture. That is because 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 require coordinated updates to all of the services and that would kill the "microservice lifecycle autonomy".

Even going 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 particular NoSQL database (like 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 a better approach. Therefore, microservices-based applications often use a mixture of SQL and NoSQL databases, the so-called [polyglot persistence approach](#).

A partitioned, [polyglot-persistent](#) architecture for data storage has many benefits, including loosely coupled services and better performance and scalability. However, it does introduce some distributed data management challenges explained in the following section named "Identifying domain-model boundaries per microservice".

Relationship between the Microservice pattern and the Bounded-Context pattern

The concept of microservice derives from the [Bounded-Context pattern \(BC\)](#) in [Domain-Driven Design \(DDD\)](#). DDD deals with large models by dividing them into multiple Bounded-Contexts and being explicit about their boundaries where each Bounded-Context has to have its own model or database, in a similar way than a microservice owns its related data. In addition, each Bounded-Context usually have its own [Ubiquitous Language](#) to help communication between software developers and domain experts.

Those terms (Domain Entities mainly) in the Ubiquitous Language can be named differently between different Bounded-Contexts even when different Domain Entities might share the same Identity. For instance, in a "User-Profile" Bounded-Context or microservice you might have the "User" Domain Entity which can share the same identity with the "Buyer" Domain entity in the "Ordering" Bounded-Context or microservice.

Therefore, a microservice is pretty much like a Bounded-Context but it also specifies that it is a distributed service, so it is built as a separated process per Bounded-Context and needs to use distributed protocols like HTTP or [AMQP](#) in order to access to the microservice. The Bounded-Context pattern, however, doesn't specify if it is a distributed service or if it is simply a logical boundary within a monolithic-deployment application, but ultimately, both patterns are very much related.

DDD benefits from microservices by getting real boundaries (distributed microservices), and ideas like not sharing the model between microservices which is something that the microservice community has converged on are what you also want in a bounded context.

References – Data Sovereignty per Microservice and Bounded-Context pattern

"Database per microservice" pattern: <http://microservices.io/patterns/data/database-per-service.html>

Bounded-Context pattern: <http://martinfowler.com/bliki/BoundedContext.html>

The PolyglotPersistence approach: <http://martinfowler.com/bliki/PolyglotPersistence.html>

Identifying domain-model boundaries per microservice

The goal when identifying model boundaries and "the size" for each microservice is not to get to the most granular separation we could have with our microservices, although it is interesting to tend small microservices. But instead, your goal should be to get to the most meaningful separation guided with your domain knowledge. The emphasis is not on the size, but instead on the business capabilities.

The term microservices puts a lot of emphasis on the size of the services, a point that most practitioners find to be rather unfortunate. For instance, [Sam Newman](#) (a recognized promoter of microservices and author of the book "[Building Microservices](#)") emphasizes that you should derive your microservices based on the DDD notion of Bounded Context, as introduced in this paper previously.

A domain model with specific domain entities applies within a concrete bounded context or microservice. A Bounded-Context delimits the applicability of a particular model and gives developer team members a clear and shared understanding of what has to be consistent and what can develop independently, which are the same goals for microservices.

A DDD technique that can be used for this is "Context Mapping". Via this technique, you identify the various contexts in the application landscape and their boundaries. The Context Map is the primary tool used to make boundaries between domains explicit. A Bounded Context encapsulates the details of a single domain, such as the domain model with its domain entities and defines the integration points with other bounded contexts/domains. This matches perfectly with the definition of a Microservice: autonomous, well defined interfaces, implementing a business capability. This makes Context Mapping (and DDD in general) an excellent tool in the architect's toolbox for identifying and designing Microservices.

When dealing with a large application, its domain model will tend to fragment: a domain expert from the Catalog domain will think differently about 'inventory' than a logistics domain expert, for example. Or the user entity might be different in size and needed attributes when dealing with a CRM expert who wants to store every detail about the customer than a Ordering Domain expert who just needs partial data about the customer. It requires lots of coordinated efforts to disambiguate all terms across all domains. And worse, this 'unified vocabulary' if you try to have a single unified database for the whole application is awkward and unnatural to use, and will very likely be ignored in most cases. Here bounded contexts (implemented as microservices) will help again: they make clear where you can safely use the natural domain terms and where you will need to bridge to other domains. With the right boundaries and sizes of your bounded contexts you can make sure your domain models "fit in your head" and that you do not have to switch between models too often.

So maybe the best answer to the question of how big a Microservice should be is: it should have a well-defined bounded context that will enable you to work without having to consider, or swap, between contexts.

In the image X-XX you can see how multiple microservices (multiple bounded-contexts) with its own model per each microservice and how their entities can be defined depending on your specific requirements for each of the identified Domains in your system.

Identifying a Domain-Model per Microservice/Bounded-Context

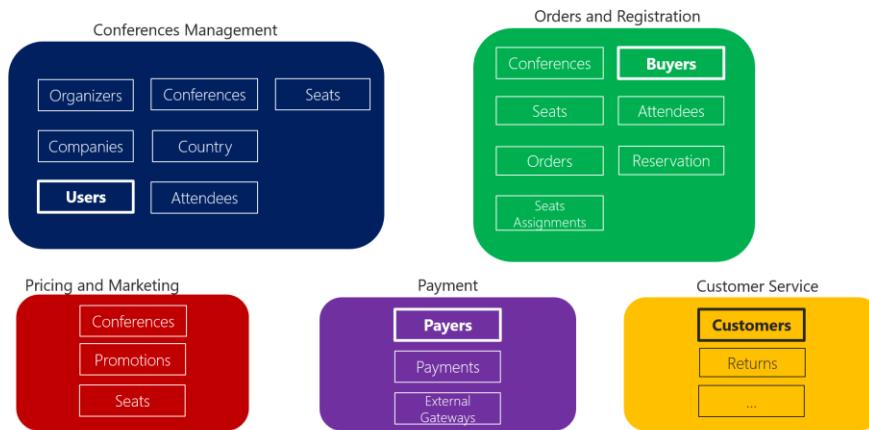


Figure X-X. Identifying Entities and Microservice's Model Boundaries

In figure X-X you can see a sample scenario related to a On-line Conferences Management system, you could have identified several Bounded-Contexts that could be implemented as microservices based on multiple identified Domains that each Domain expert defined for you. As you can observe, there are entities that are present just in a single microservice's model, like "Payments" in the Payment microservice or sub-system. Those will be easy to implement. However, you may also have entities which under different flavor or shape but sharing the same identity are shared across 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 "buyer" in the Ordering microservice, or named as "Payer" in the Payment microservice and even present in the "Customer Service" microservice but called this time as "Customer". The reason for that 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 "Conferences Management" might have most of its personal data attributes. However, that same user in the shape of a "Payer" in the microservice "Payment" or in the shape of a "Customer" in the microservice "Customer Service" might not need the same list of attributes. A similar approach is illustrated in the image X-XX.

Decomposing a Traditional Data-Model into multiple Domain-Models (One Domain-Model per microservice or Bounded-Context)

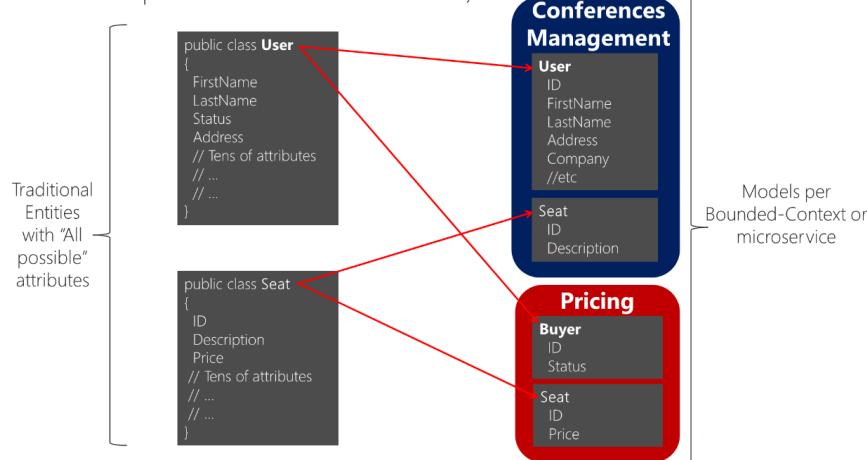


Figure X-X. Decomposing traditional data models into multiple domain-models

You can see how the User is present in the "Conferences Management" microservice's model, but it is also present in the form of a "Buyer", with alternate attributes, in the "Pricing" microservice's model because each microservice or Bounded-Context might not need all the data related to a User but just part of it, depending on the problem to solve or the context. For instance, for the "Pricing problem" you don't need the Address or the Passport number of the user but just his ID and the Status (like Gold/Silver/Bronze) which will impact on discounts when pricing the seats per buyer.

In the case of the "Seat", it is called with the same name but with different attributes per domain-model, however, it shares the same identity based on the same ID, as it happens with the User and Buyer.

Challenges and solutions for Distributed Data Management

Challenge #1: How to maintain consistency across multiple services

As stated, the data owned by each microservice is private to that microservice and can only be accessed via its microservice API, and it has multiple benefits as explained before. However, the first challenge coming from this approach is how to implement business transactions that maintain consistency across multiple microservices.

To analyze this problem, let's take a look at an example from the [eShopOnContainers reference application](#). The Catalog microservice maintains information about all the products, including their stock. The Ordering microservice manages orders and must verify that a new order doesn't exceed the available catalog product's stock. In a hypothetical monolithic version of this app, the Ordering subsystem could simply use an ACID transaction (like "Two Phase Commit" transactions that you can do with SQL Server and the DTC) to check the available stock, create the order and update the available stock in the Products table.

In contrast, in a microservices architecture the Order and Product tables are private to their respective services, as shown in image X-XX.

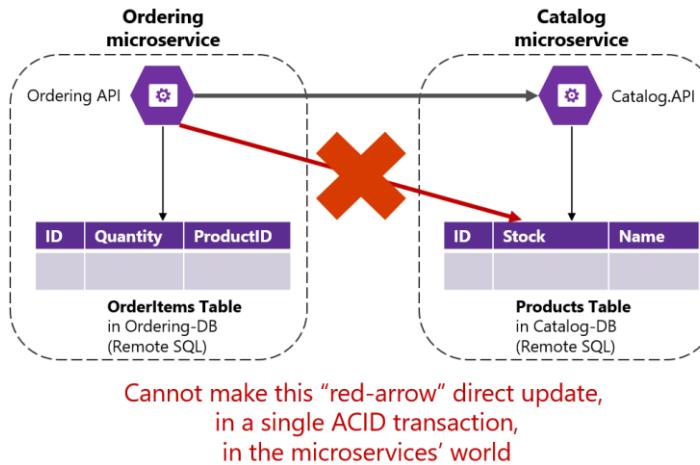


Figure X-X. Cannot access directly Tables from other microservices

The Ordering microservice should not access the Products table directly, as the product table is owned by the Catalog microservice. It can only use the API provided by the Catalog microservice.

As stated by the [CAP theorem](#), you need to choose between availability and ACID-style consistency, and availability is usually the better choice for large and scalable systems like the ones that microservice-based architectures target. Moreover, ACID-style or “Two-phase commit” transactions are not just against microservices’ principles, but most NO-SQL databases (like Azure DocumentDB, MongoDB, etc. do not support “Two-phase commit” transactions. However, maintaining data consistency across services and databases is essential and this challenge is also related to the question “How to propagate changes across multiple microservices when certain attributes are redundant?”.

A good solution for both questions is based on “Eventual Consistency between microservices” articulated through Event-Driven communication and a Publish/Subscription system, which is covered in the section about “Event-Driven Communication” later in this document.

Challenge #2: How to implement queries that retrieve data from multiple microservices

The second challenge is the question on how can you implement queries that retrieve data from multiple services while avoiding a super-chatty communication from remote client apps that might need data from multiple microservices? An example could be a mobile app screen that needs to show in the same screen info owned by multiple microservices. Another example would be a complex report involving many tables. The right solution really depends on the complexity of the queries. The most popular solutions are the following.

- A. **API Gateway:** For simple data aggregation coming from several microservices (several databases at the end of the day), the most recommended approach would be to make that aggregation in an “Aggregation microservice” also known with the API Gateway pattern explained in the following section when talking about inter-microservice communication.

- B. **CQRS “Query-Table”:** This solution is also known as the [Materialized view pattern](#) that pre-joins data owned by multiple microservices. For complex data aggregation from multiple tables and databases, comparable to a very complex join that you could do with a complex SQL sentence involving multiple tables, that could be addressed with a CQRS approach by creating a de-normalized “Query-Table” in a different database used just for queries. That table will be designed according to the data you need for that complex query, with a 1:1 relationship between fields needed by your application’s screen and the columns in that “query-table”. This approach not only solves this particular problem but also improves considerably the application performance when comparing it with a complex relational join targeting multiple tables, because you already have the query result persisted in an “Ad-Hoc” table especially made for that query. Of course, using a CQRS approach means more development work and you again need to embrace “eventual consistency”, but performance and high-scalability requires of these types of approaches and solutions.
- C. **“Cold-Data” in central databases:** For complex reports and queries, a common approach is to export your “hot data” (transactional data from the microservices) into large databases only used for reporting. That central database can be a relational database like in SQL Server, Data Warehouse based like Azure SQL Data Warehouse or even based on Big Data solutions like Hadoop. Keep in mind that this centralized database would be used only for queries, but not for the original updates and transactions, as *“your source of truth has 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 using Event-Driven communication, that integration would be very similar to the way you propagate data to the mentioned CQRS “Query Database”.

However, it is important to highlight that if you have this problem very often and you constantly need to aggregate information from multiple microservices for complex queries needed by your application (not considering reports/analytics that always should use cold-data central databases), that is a symptom of a possible bad design as a microservice should tend to be as isolated as possible from other microservices. Having this problem very often might be a reason why you would want to merge two microservices. You need to balance autonomy of evolution and deployment of each microservice with strong dependencies and data aggregation.

References – Distributed Data
The 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
CQRS (Command and Query Responsibility Segregation): http://martinfowler.com/bliki/CQRS.html
Materialized View pattern: https://msdn.microsoft.com/en-us/library/dn589782.aspx
ACID vs. BASE: http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/
Compensating Transaction pattern: https://msdn.microsoft.com/en-us/library/dn589804.aspx

Stateless vs Stateful Microservices and advanced frameworks

As mentioned earlier, each microservice must own its domain model (data+logic). In the case of stateless microservices, the databases will be external, employing relational options like SQL Server or No-SQL options like MongoDB or Azure Document DB. Going further, the services themselves can be stateful, which means the data resides within the same microservice. This data could exist not just within the same server, but within the same microservice's process, in-memory and persisted on hard drive and replicated to other nodes. Figure X-XX shows the different approaches.

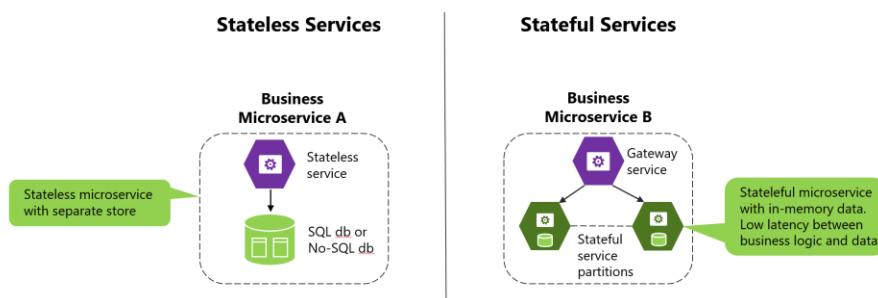


Figure X-XX. Stateless vs. Stateful services

Stateless is a perfectly valid approach and easier to implement than stateful microservices, as it is similar to traditional and well-known patterns. But stateless microservices impose latency between the process and data sources, while also presenting more moving pieces when improving performance via additional cache and queues. The result: you can end up with complex architectures with too many tiers.

Stateful microservices, on the other hand, can excel in advanced scenarios, as 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, however, complementary. For instance, you can see in the image X-XX that a stateful service could be split in multiple partitions and in order to get access to those partitions you might need a stateless service acting as a gateway service so it knows how to address each partition depending on partition keys.

The drawback in stateful services? - Stateful services impose a level of complexity in order to scale out. Functionality that would usually be implemented within the external database boundaries must be addressed for things such as data replication across stateful microservices replicas, data partitioning and so on. However, this is precisely one of the areas where an orchestrator like [Azure Service Fabric](#) can help you the most—by simplifying the development and lifecycle of [stateful microservices on Service Fabric](#) with Reliable Services API and Reliable Actor framework.

Other additional microservice oriented frameworks that allow stateful services and the actors pattern and improve fault tolerance and latency between business logic and data are, project [Orleans](#), from Microsoft Research, and [Akka.NET](#), nowadays both frameworks improving their Docker support.

Notice that Docker containers are by themselves, stateless. If you want to implement a stateful service you will need any of the mentioned additional, prescriptive and higher-level frameworks.

API Gateway pattern vs. Direct Client-to-Microservice communication

In a microservices architecture, each microservice exposes a set of what are typically fine-grained endpoints. That fact can impact the client-to-microservice communication.

Direct Client-to-Microservice communication

A first possible architecture approach with microservices can be using a "*Direct Client-To-Microservice communication architecture*" which means that a client app can make direct requests to each of the microservices, as shown in figure X-XX.

Direct Client-To-Microservice communication

Architecture

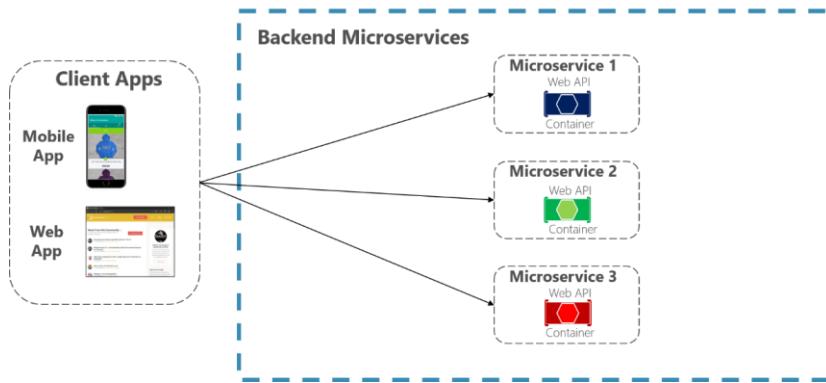


Figure X-XX: Using the Direct Client-To-Microservice communication architecture

Each microservice will have a public endpoint like <https://servicename.applicationname>, sometimes with a different TCP port per microservice. In production, that URL would map to the microservice's load balancer, which distributes requests across the available instances.

This "*Direct Client-To-Microservice communication architecture*" can be good enough for a small microservice-based application, however when building large and complex microservice based application (like 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 do clients 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 network round trips across Internet which increases latency and complexity in the UI side. Ideally, responses would need to be efficiently aggregated in the server side.

- *How to allow clients to communicate with services that use non-Internet-friendly protocols?* - Protocols used on the server side (like AMQP or binary protocols) are not always well supported in clients, so requests will need to be translated.
- *How can you handle cross-cutting concerns* such as authorization, load balancing, data transformations and dynamic request dispatching? – Implementing security and cross-cutting concerns on every microservice can be costly. A possible approach would be to have those services within the Docker host restricting access from the outside and implementing those cross-cutting concerns like security and authorization in a centralized place.
- *How to shape a façade especially made for mobile apps?* - API's are normally not designed around the needs of specific mobile platforms, so responses will need to be efficiently transformed, aggregated and compressed.

API Gateway

When designing and building large/complex microservice based applications, a better approach is to use what is known as an [API Gateway](#). An API Gateway is a service that is the single-entry point into the application's backend system. It is similar to the [Facade pattern](#) from object-oriented design, but in this case in a distributed system. The figure X-XX shows how an [API Gateway](#) can fit into a microservice-based architecture:

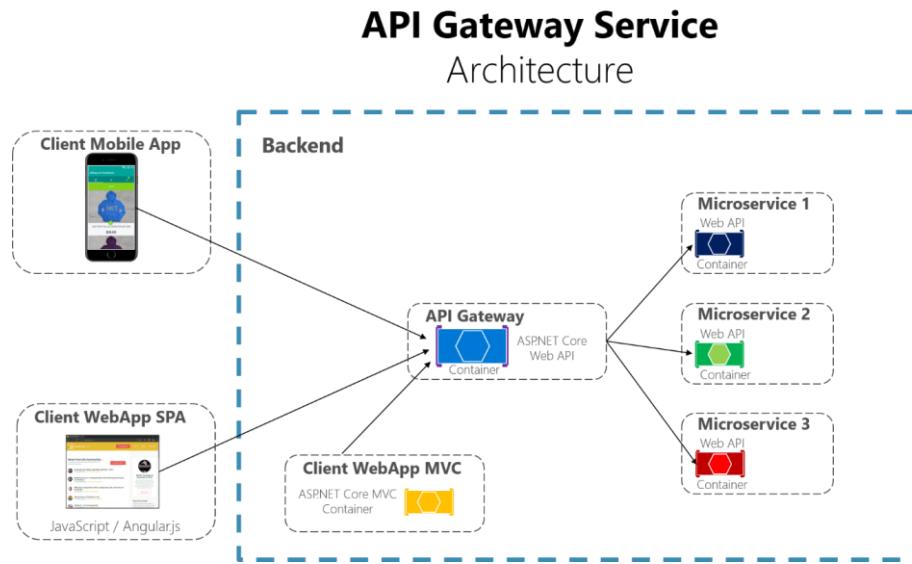


Figure X-XX. Using the API Gateway pattern in a microservice based architecture

In this case, the API Gateway would be implemented as a custom Web API service running as a container. That approach, based only on a custom-built API Gateway, might be good enough for medium size applications where your only requirement here is about that mentioned API Gateway.

Another alternative is to use a product like [Azure API Management](#) which can solve your API Gateway needs plus additional features like gathering insights from your APIs so you can get a better understanding of how your APIs are being used and performing by viewing near real-time analytics reports and identify trends that might impact your business. Plus, you can have log request and response data for further online and offline analysis.

API Gateway with Azure API Management

Architecture

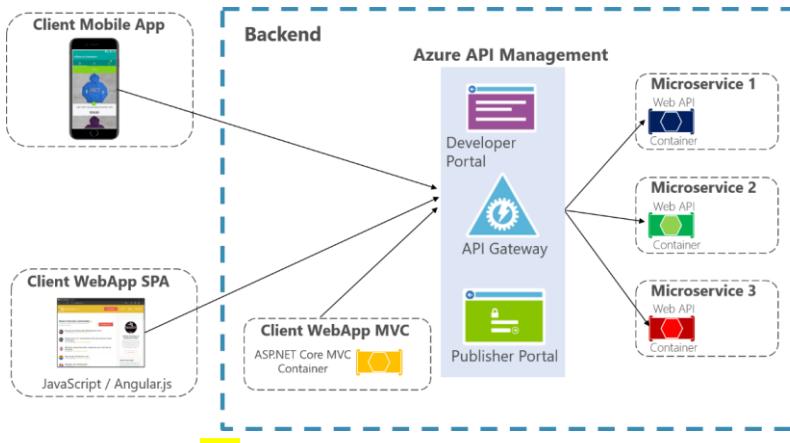


Figure X-XX Using Azure API Management for your API Gateway

With [Azure API Management](#) you can secure your APIs using a key, token, and IP filtering and enforce flexible and fine-grained quotas and rate limits, modify the shape and behavior of your APIs using policies, and improve latency and scale your APIs with response caching. However, this document is limiting the architecture to a simpler and custom-made containerized architecture to specifically focus on plain containers without using PaaS products like Azure API Management. But for large microservice-based applications deployed into Microsoft Azure, we encourage to review and adopt Azure API Management as the base for your API Gateways.

References – API Gateway and API Management

API Gateway pattern

<http://microservices.io/patterns/apigateway.html>

Azure API Management

<https://azure.microsoft.com/en-us/services/api-management/>

Communication between microservices

In a monolithic deployment application, components invoke one another via language-level method or function calls, strongly coupled if creating objects with sentences like “new ClassName” or in a decoupled way if using Dependency Injection. But either way, those objects are running within the same process. On the other hand, a microservices-based application is a distributed system running on multiple processes/services and even on multiple machines. Each service instance is typically a process. Therefore, services must interact using an inter-process communication protocols like *HTTP*, *TCP*, *AMQP* or binary protocols, depending on the nature of each service.

Communication Types

When selecting a communication mechanism between services, it is important to think first about how services should interact. Initially, these types can be classified along two dimensions.

The first dimension is whether the invocation is synchronous or asynchronous:

- *Synchronous* – The client waits for a response from the service. That waiting time usually blocks the execution of the client while it waits. It is easier to debug, but overall performance can be worse than when using asynchronous execution.
- *Asynchronous* – The client doesn't block while waiting for a response. Depending on the type of the logic, you can expect responses coming immediately or you could also have responses coming back at a much later time since it shouldn't impact the client execution as it is not blocked. When using asynchronous mechanisms, the overall performance can be much better balanced as you shouldn't have the same bottlenecks like you would have when using synchronous communication, however, development and debugging get much more complex.

The second dimension is whether the communication is one-to-one or one-to-many:

- *One-to-one* – Each client request is processed by exactly one service instance.
 - An example of this communication is the “[Command pattern](#)”.
- *One-to-many* – Each request is processed by multiple services or receivers. This type of communication needs to be asynchronous as a single client synchronous execution cannot usually get response from multiple services.
 - An example of this type of communication is a [Publish/Subscribe](#) mechanism used in patterns like “[Event-driven architecture](#)” based on an Event-Bus interface or Message Broker when propagating data-updates between multiple microservices through events, usually implemented through a Service Bus or similar artifact like [Azure Service Bus](#) by using [Topics](#) and subscription to topics.

The following table shows how those dimensions are usually applied in a complementary way.

	Synchronous	Asynchronous
One-to-One	Request/response	Request/async response Fire and forget (Notification)
One -to-Many	--	Publish/Subscription <ul style="list-style-type: none">- Registration action- Publish action- Message Handlers

A microservice-based application will usually use a combination of these communication styles. The most common type is a One-to-One communication (either sync or async) when invoking regular Web API HTTP services. However, when propagating data-updates between multiple microservices, a one-to-many asynchronous communication is very flexible and convenient as implemented in an [event-driven architecture](#).

Communication protocols and technologies

There are many different protocols and choices you can use 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 when publishing your services outside the Docker host or microservice cluster. If you are communicating microservices internally (within your Docker host or microservice cluster) you might also want to use binary format communication mechanisms depending on the development platform you are using. Alternatively, you can use asynchronous, message-based communication mechanisms such as AMQP.

Additionally, there are also a variety of different message formats. Services can use human readable, text-based formats such as JSON or XML. Alternatively, you can use a binary format (which can be more efficient) but if your chosen binary format is not a standard it is probably not a good idea to publicly publish your services using that format, but you might want to use it only for your internal communication between your microservices, like when communicating microservices between them within your Docker host or microservice cluster (Docker orchestrators or Azure Service Fabric).

Request/Response communication with HTTP and REST (Synchronous and Asynchronous)

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 (either sync or async) is especially well suited for querying data for "live UI" (live User Interface) from client apps, so in a microservice architecture you will probably use this communication mechanism for most of the needed queries for that purpose, as shown in figure X-XX.

Request/Response Communication for Live Queries and Updates

HTTP and REST based Services

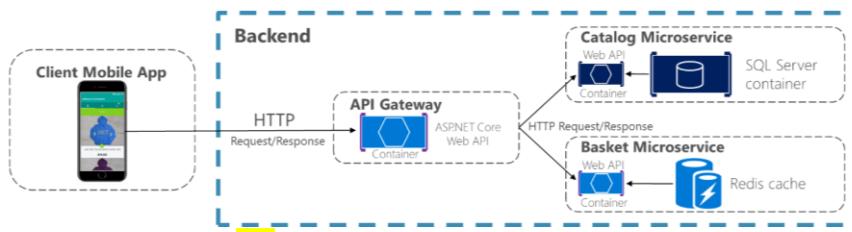


Figure X-XX. Using HTTP Request/Response communication (Sync or Async)

If it is a synchronous request/response communication, the thread that makes the request is blocked while waiting for a response, that is how it will behave in .NET when consuming an [ASP.NET Web API synchronously](#). However, you would usually want to consume a microservice asynchronously, so the client thread won't be blocked until you get a response from the server. When consuming a

service asynchronously, you usually will have a call-back method in the client that will be called by the service when returning the call response. However, in modern languages that is simplified and even when it will internally behave asynchronously, using modern [async/await keywords](#) in C# you can program async services and client calls in a very simplified way like if you were invoking synchronous methods. You can therefore [communicate asynchronously with ASP.NET Web API services](#).

Usually, when using a request/response communication (either sync or async), the client assumes that the response will arrive in a timely fashion, like less than a second or a few seconds as maximum, but not minutes, hours or days, of course. For those delayed responses you will need to implement asynchronous communication based on messaging technologies.

A popular architectural communication style for this the request/response communication style is [REST](#), which is based and tightly coupled to the [HTTP](#) protocol embracing HTTP verbs like PUT, POST and GET. REST is also the most used architectural communication approach when creating Data-Driven (resource/data oriented) services. You can implement REST services when developing ASP.NET Core Web API services, as it will be explained in the implementation sections of this document, later on.

There is additional value when using HTTP REST services as your interface definition language. For instance, using [Swagger metadata](#) to describe your service API you can use tools that generate client stubs or are able to directly discover and consume your services. In later sections of this document it is explained how to generate Swagger metadata in your ASP.NET Core Web API services.

The intention of the document is not to introduce or explain in detail REST or HTTP, so for further information about it, like "REST maturity levels", read the following reference.

References – REST and HTTP request/response services
REST Maturity Model: http://martinfowler.com/articles/richardsonMaturityModel.html

[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 about microservices, Bounded-Contexts and how can you identify each model for each microservice, after you accept that a User, Customer, Product, Account, etc. may mean different things to different bounded-contexts or microservices that means that at the end of the day, you may end up with these related concepts distributed around our architecture but you need some way to reconcile changes across these different models when changes happen. This is where event-driven communication based on asynchronous messaging have to be used.

When using messaging, processes communicate exchanging messages asynchronously. A client makes a request to a service by sending it a message. If the service is expected to reply, it does so by sending a separate message back to the client. Since the communication is asynchronous, the client does not block waiting for a reply, but in addition to that, since it is a message-based communication, the client is assuming that the reply will not be received immediately.

A message consists of headers (metadata such as identification or security information) and a message body. Messages are exchanged over channels. Any number of senders can send messages to a channel. Similarly, any number of consumers can receive messages from a channel.

There are two kinds of channels or communications, “one-to-one” communication and “publish-subscribe” communication.

One-to-One Asynchronous message communication

A point-to-point communication delivers a message to exactly one of the consumers that is reading from the channel, so it will be processed just once. A publish-subscribe channel delivers each message to all of the attached or subscribed consumers. Services use publish-subscribe channels for the one-to-many interaction styles described before.

Message-based asynchronous communication is especially well suited to propagate data updates across a microservice architecture. For instance, if a microservice’s data is updated (like a user name within the “User Profile” microservice) but that same data needs to be propagated to a different microservice (like the “Ordering” microservice that could also hold the buyer’s name) so they are eventually consistent, that kind of inter-microservice communication should be based on asynchronous communication by using integration events between microservices, as in image X-XX.

Asynchronous and Message based communication

One-to-One Async. communication

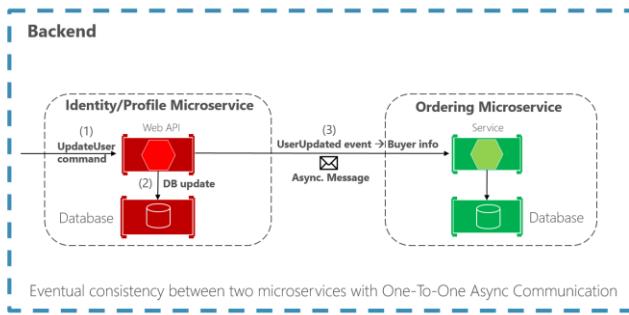


Figure X-XX. One-to-One async message communication

The protocols to be used for the asynchronous message communications can be multiple. You could use message queues for this communication or you could also use HTTP asynchronously.

One-to-Many Asynchronous message communication

Additionally, you might want to use a Publish/Subscribe mechanism so your communication starting from the sender will be open to additional subscriber microservices or even external application integration, in the future.

When using a Publish/Subscribe communication you might be using an Event-Bus interface in order to publish events to any subscriber, but another possibility (usually for different purposes) is a real-time and one-to-many communication (broadcast) that you can achieve with protocols like [WebSockets](#) and higher level frameworks like [ASP.NET SignalR](#).

Asynchronous Real-Time communication

As shown in image X-XX, real-time asynchronous communication means that you have the ability to have server code push content to connected clients instantly as it becomes available, rather than having the server wait for a client to request new data, plus since it is real-time, client apps will show the changes in the client apps almost in real-time. As mentioned it is usually handled by a protocol like WebSockets. A typical example is when a service communicates a change in the score of a sports game to many client web apps, simultaneously.

Asynchronous Real Time communication

One-to-many communication

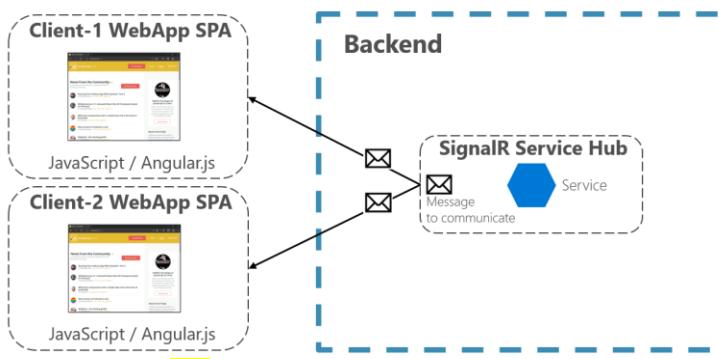


Figure X-XX. Asynchronous Real-Time communication

Asynchronous Event-Driven communication

When using this type of communication and architectural approach, 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.

As introduced when tackling on "data problems" in the previous section named "Challenges and solutions for Distributed Data Management", you can use events to implement business transactions that span multiple services and will have eventual-consistency between those services. An Eventual-Consistent transaction consists of a series of distributed steps. Each step consists of a microservice updating a business entity and publishing an event that triggers the next step.

A very important point is that you might want to be able to communicate the same event to multiple destination microservices that are subscribed to the same event. For that, you can use the Publish/Subscribe messaging based on event-driven communication, as shown in image X-XX. This Pub/Subs mechanism is not exclusive from the microservice architecture, it is pretty similar to the way [Bounded-Contexts](#) in [Domain-Driven Design](#) should communicate or the way you propagate updates from the "writes-database" to the "reads-database" in [CQRS \(Command and Query Responsibility Segregation\)](#) architectural approach so you can have eventual consistency between multiple data sources across your distributed system.

Asynchronous Event-Driven communication

One-to-many communication

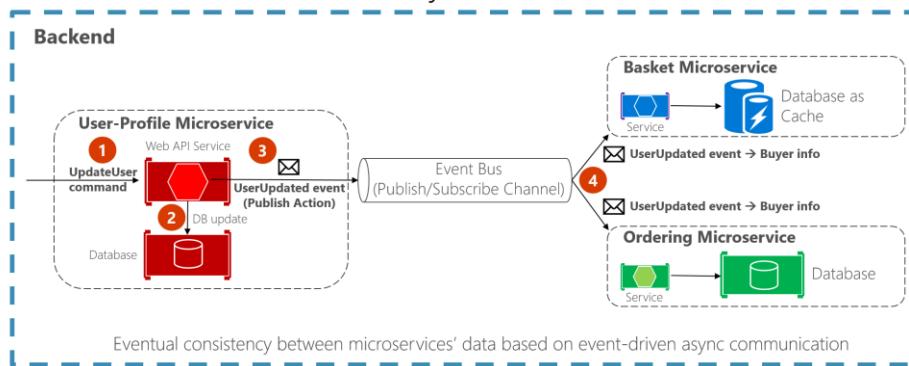


Figure X-XX Event-Driven and async message communication

In regards the protocol communication to use for event-driven message-based communications, it depends on your implementation. A reliable queued communication would be achieved by using [AMQP](#), but using HTTP asynchronously could also be a choice, although less reliable. What you will probably need is some kind of abstraction level (like an Event-Bus interface) with its related implementation in classes with code using the API from a Service Bus like [Azure Service Bus with Topics](#) or [RabbitMQ](#), so you can articulate the mentioned Publish/Subscribe system.

One challenge with implementing an event-driven architecture is how to atomically update state in the original microservice while publishing its related event, in a single transaction. There are a few ways to accomplish this:

1. Using 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
2. Using a [transaction log mining](#)
3. Using the [event sourcing](#) pattern.

References – Publish/subscribe, eventual consistency and other DDD patterns

Event Driven Messaging

http://soappatterns.org/design_patterns/event_driven.messaging

Publish/Subscribe channel

<http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html>

CQRS (Command and Query Responsibility Segregation)

<http://microservices.io/patterns/data/cqrs.html>

<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

Creating and Evolving 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 as long as you don't break your API contract, that is why that contract is so important. If you change that contract, it will impact your client applications (either your client apps 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 when you might be thoughtful about your initial contracts, a service API will need to change over time. When that happens, especially when your API is not used just by a single application but it is a public API consumed by multiple client applications, you usually cannot force all clients to upgrade to your new API contract. You usually will need to incrementally deploy new versions of a service such that both old and new versions of a service contract will be running simultaneously, therefore, it is important to have a strategy for your service versioning.

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

Sometimes, however, you need to make major and incompatible changes to a service API. Since 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 of time. If you are using an HTTP-based mechanism such as REST, one approach is to embed the API version number in the URL. Then, you can decide between implementing both versions simultaneously within the same service instance or alternatively, you could deploy different instances that each handle a particular version of the API.

References – Versioning ASP.NET Core Web API services

ASP.NET Core RESTful Web API versioning made easy

<http://www.hanselman.com/blog/ASPNETCoreRESTfulWebAPIVersioningMadeEasy.aspx>

Microservices addressability and the Service Registry

Each microservice has a unique name (URL) used to resolve its location. Your microservice needs to be addressable wherever it is running. If you are thinking about machines and which one is running a particular microservice, things will go bad quickly. In the same way that DNS resolves a particular URL to a particular machine, 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. Equally, when a machine fails, the registry service must tell you where the service is now running.

The [service registry](#) is a key part of service discovery. It 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 becomes 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 the next sections), 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.

References

The Service Registry pattern

<http://microservices.io/patterns/service-registry.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 we write as developers is 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 is able to 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, doesn't need to recover. It also needs to then 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.

Health Reports 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 emerges 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 as a whole.

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.

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 to solve as it is not just about building your microservice architecture but you will also need, for sure, high availability, addressability, resiliency, health and diagnostics if you intend to have a stable and cohesive system.



Figure X-XX. A Microservice Platform is fundamental for Microservice based applications

Those mentioned complex problems shown in figure XX-X are very hard to solve by yourself. However, development teams should focus on solving business problems and building custom applications with microservices approaches but not solving those complex infrastructure problems or the cost of any microservice-based application would be huge. This is why there are microservice-oriented platforms (usually called orchestrators or microservice clusters) that try to solve those hard problems of building and running a service and utilize infrastructure resources efficiently, reducing the complexities of building applications with a microservice approach.

Orchestrators might sound similar in concept, but the capabilities offered by each of them can be pretty different in features available from each and maturity state depending on the OS platform.

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

In this more enterprise and advanced scenario using microservices or even simpler multi-container applications, you are building an application composed by multiple services. If it is a microservice-approach, each microservice would own its model/data so it will be autonomous from a development and deployment point of view. But even if you have a more traditional application but also composed by multiple services (like SOA), you will also have multiple containers/services comprising a single business application that need to be deployed as a distributed system.

An architecture for composed and microservices approaches using containers would be similar to the diagram in Figure X-X.

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

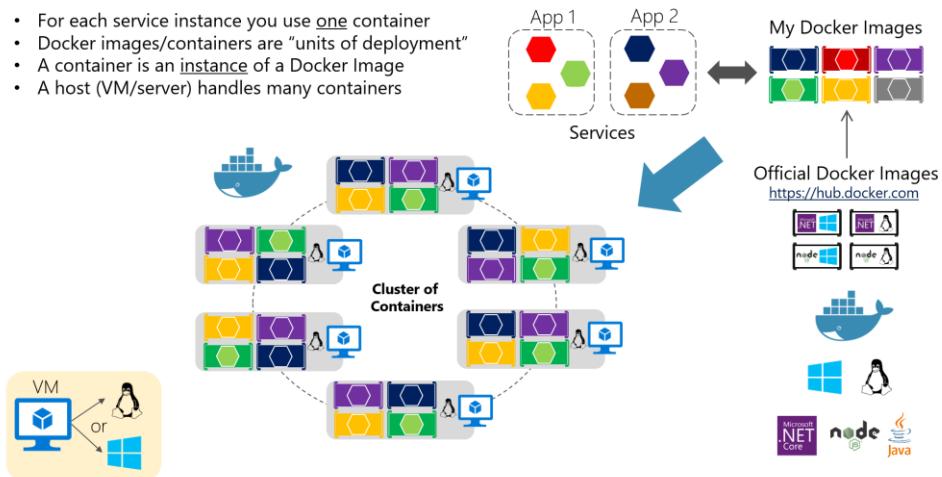


Figure X-X. Cluster of containers

It looks a logical approach, but now, how are you load-balancing, routing and orchestrating these composed applications?

While the Docker CLI meets the needs of managing one container on one host, it falls short when it comes to managing multiple containers deployed on multiple hosts targeting 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 the network and data storage.

To go beyond the management of individual containers or very simple composed apps and target larger enterprise applications and microservices approaches, you must turn to orchestration and clustering platforms for Docker containers like **Docker Swarm**, **Mesosphere DC/OS** and **Kubernetes**

available as part of **Microsoft Azure Container Service** or Microsoft's container orchestrator **Azure Service Fabric**.

From an architecture and development point of view it is important to drill down on those mentioned platforms and products supporting advanced scenarios (clusters and orchestrators) if you are building large enterprise composed or microservices based applications.

Clusters. When applications are scaled out across multiple host systems, the ability to manage each host system and abstract away the complexity of the underlying platform becomes attractive. That is precisely what Docker clusters and schedulers provide. Examples of Docker clusters are Docker Swarm, Mesosphere DC/OS. Both can run as part of the infrastructure provided by Microsoft Azure Container Service.

Schedulers. "Scheduling" refers to the ability for an administrator to load a service file onto a host system that establishes how to run a specific container. Launching containers in a Docker cluster tends to be known as scheduling. While scheduling refers to the specific act of loading the service definition, in a more general sense, schedulers are responsible for hooking into a host's init system to manage services in whatever capacity needed.

A cluster scheduler has multiple goals: using the cluster's resources efficiently, working with user-supplied placement constraints, scheduling applications rapidly to not let them in a pending state, having a degree of "fairness", being robust to errors and always available.

As you can see, the concept of cluster and scheduler are very tight, so usually the final product provided from different vendors provide both capabilities.

The list below shows the most important platform/software choices you have for Docker clusters and schedulers. Those clusters can be offered in public clouds like Azure with Azure Container Service.

Software Platforms for Container Clustering, Orchestration and Scheduling	
Docker Swarm 	Docker Swarm is a clustering and scheduling tool for Docker containers. It turns a pool of Docker hosts into a single, virtual Docker host. Because Docker Swarm serves the standard Docker API, any tool that already communicates with a Docker daemon can use Swarm to transparently scale to multiple hosts. Docker Swarm is a product created by Docker itself. Docker v1.12 or later can run native and built-in Swarm Mode, although v1.12 is also backwards compatible for people who desire K8S (Kubernetes)
Mesosphere DC/OS 	Mesosphere Enterprise DC/OS (based on Apache Mesos) is an enterprise grade datacenter-scale operating system, providing a single platform for running containers, big data, and distributed apps in production. Mesos abstracts and manages the resources of all hosts in a cluster. It presents a collection of the resources available throughout the entire cluster to the components built on top of it. Marathon is usually used as orchestrator integrated to DC/OS.

 Google Kubernetes	<p>Kubernetes spans cluster infrastructure plus containers scheduling and orchestrating capabilities. It is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.</p> <p>It groups containers that make up an application into logical units for easy management and discovery.</p>
 Azure Service Fabric	<p>Service Fabric is a Microsoft's 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 and more importantly you can mix both services in processes and services in containers together in the same application.</p> <p>This feature (Service Fabric deploying services as Docker containers) is in preview for Linux and will be in preview for Windows Server 2016 in the upcoming release</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>

Figure 5-7. Software platforms for container clustering, orchestrating, and scheduling

Docker clusters in Microsoft Azure

From a cloud offering perspective, several vendors are offering Docker containers support plus Docker clusters and orchestration support, like Microsoft Azure, Amazon EC2 Container Service, Google Container Engine, etc.

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's Azure Service Fabric** (a microservices platform) which will support Docker in upcoming release. Service Fabric runs on Azure or any other cloud and also [on-premises](#).

Azure Container Service

Coming back to the Docker cluster for composed applications, in Figure 5-8 represents how it maps to Azure Container Service (ACS). A Docker cluster pools together multiple Docker hosts and exposes them as a single virtual Docker host so you can deploy into the cluster multiple containers while the cluster will handle all the complex management plumbing, like scalability, health, etc.

Azure Container Service (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 upon 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.

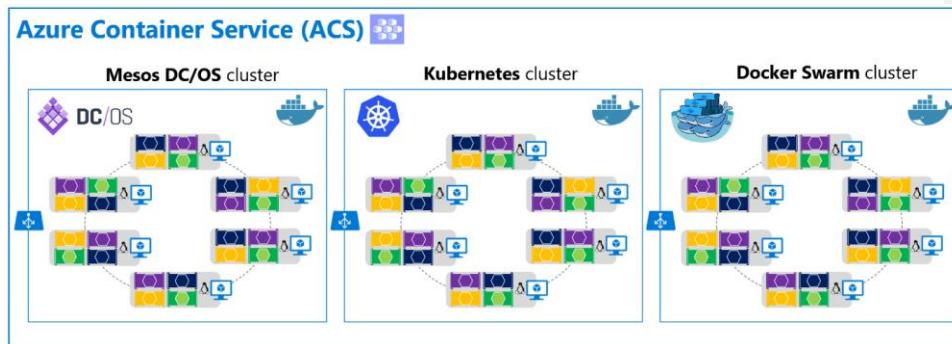


Figure 5-8. Clustering choices in ACS

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, 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.

From a usage perspective, the goal with Azure Container Service is to provide a container hosting environment by using open-source tools and technologies that are popular. To this end, it exposes the standard API endpoints for your chosen orchestrator. By using these endpoints, you can leverage any software that is capable of talking 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.

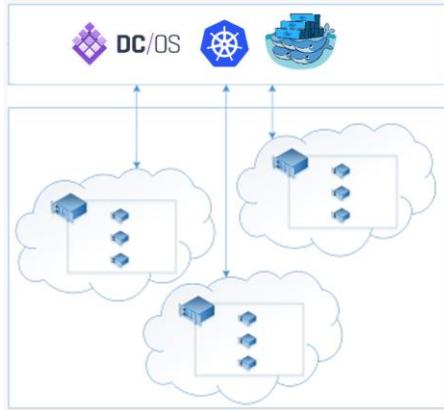


Figure 5-9. Orchestrators in ACS

Getting started with Azure Container Service

To begin using Azure Container Service, you deploy an Azure Container Service cluster via the portal (search for 'Azure Container Service'), by using an Azure Resource Manager template ([Docker Swarm](#), [Kubernetes](#) or [DC/OS](#)) or with the [CLI](#). 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 by 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.

References for Azure Container Service and related technologies
Azure Container Service introduction https://azure.microsoft.com/en-us/documentation/articles/container-service-intro/
Docker Swarm https://docs.docker.com/swarm/ https://docs.docker.com/engine/swarm/
Mesosphere DC/OS https://docs.mesosphere.com/1.7/overview/
Kubernetes http://kubernetes.io/

Azure Service Fabric

Azure Service Fabric emerged from a transition by Microsoft 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 DocumentDB, 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 utilize 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, 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 previously described.
- 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, for example, you can get health and diagnostics information, or you can take advantage of built-in high availability.

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

As shown in figure X-XX, you can create and run microservices in Service Fabric either as simple processes or as Docker containers.

Azure Service Fabric – Types of clusters

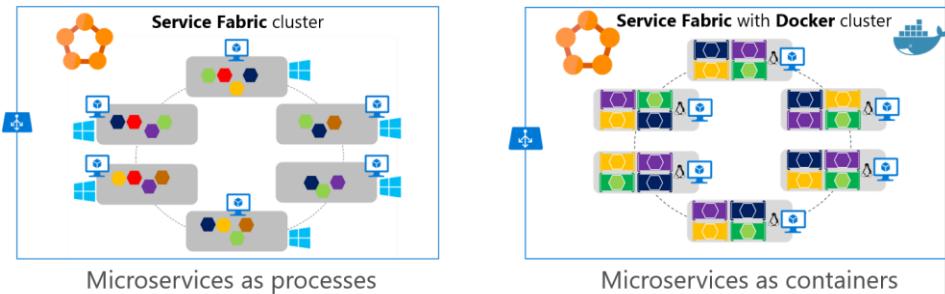


Figure X-XX. Deploying microservices as processes or as containers in Azure Service Fabric

Note that until late 2016, Service Fabric clusters on Windows didn't have Docker container support and only Service Fabric clusters based on Linux hosts were able to run Docker containers. However, in upcoming versions of Azure Service Fabric, you will be able to run either Linux containers or Windows Containers using Docker engine and Azure Service Fabric infrastructure.

Development process for Docker based applications

Vision

Develop containerized .NET applications the way you like it, either IDE focused with Visual Studio and Visual Studio tools for Docker or CLI/Editor focus with Docker CLI and Visual Studio Code.

Development environment for Docker apps

Development tools choices: IDE or editor

No matter if you prefer a full and powerful IDE or a lightweight and agile editor, either way Microsoft have you covered when developing Docker applications.

Visual Studio with Docker Tools. When using *Visual Studio 2015* you can install the add-on tools "Docker Tools for Visual Studio". When using *Visual Studio 2017*, Docker Tools come built-in already. In both cases you can develop, run and validate your applications directly in the target Docker environment. F5 your application (single container or multiple containers) directly into a Docker host with debugging, or CTRL + F5 to edit & refresh your app without having to rebuild the container. This is the simplest and more powerful choice for Windows developers targeting Docker containers for Linux or Windows.

[Download Docker Tools for Visual Studio](#)

[Download Docker for Mac and Windows](#)

Visual Studio Code and Docker CLI (Cross-Platform Tools for Mac, Linux and Windows). If you prefer a lightweight and cross-platform editor supporting any development language, you can use Microsoft Visual Studio Code and Docker CLI. These products provide a simple yet robust experience which is critical for streamlining the developer workflow. By installing "Docker for Mac" or "Docker for Windows" (development environment), Docker developers can use a single Docker CLI to build apps for both Windows or Linux (execution environment). Plus, Visual Studio code supports extensions for Docker with intellisense for Dockerfiles and shortcut-tasks to run Docker commands from the editor.

[Download Visual Studio Code](#)

[Download Docker for Mac and Windows](#)

.NET languages and frameworks for Docker containers

As introduced in initial sections, you can use **.NET Framework** or **.NET Core** and also the OSS project **Mono** when developing Docker containerized .NET applications which provide the ability to develop in **C#**, **F#** or **VB.NET** targeting **Linux** or **Windows** containers, depending on the chosen framework.

Development workflow for Docker apps

The application development Lifecycle starts from each developer's machine working coding the app itself, using his preferred language and testing it locally. But in every case, you will have a with a very important point in common no matter what language/framework/platforms you choose. In this specific workflow you are always developing and testing Docker containers, but locally.

The container or instance of a Docker image will contain these components:

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

The inner-loop development workflow that utilizes Docker can be set up as the following process explained in several steps. Note that the initial steps to set up the environment is not included, as that has to be done just once.

Workflow for developing Docker container based applications

An app will be made up from your own services plus additional libraries (Dependencies).

The following steps are the basic steps usually needed when building a Docker app, as illustrated in Figure X-XX.

Inner-Loop development workflow for Docker apps

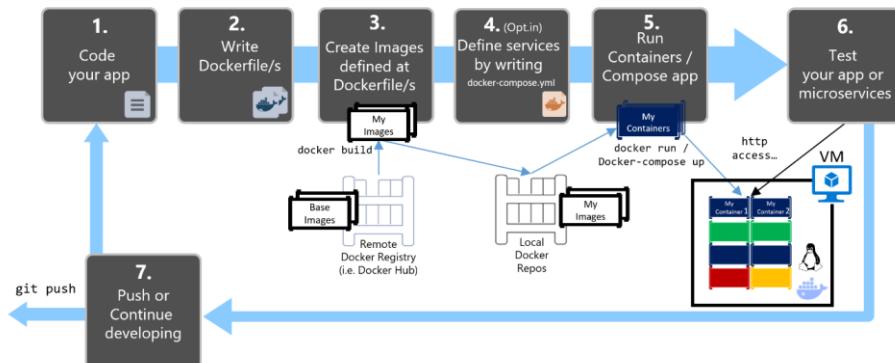


Figure X-XX. Step-by-step workflow developing Docker containerized apps

In this guide, this whole process is detailed and every critical step is explained.

When using a CLI+Editor development approach like using just **Visual Studio Code** plus **Docker CLI**, you need to know every step. If using VS Code and Docker CLI, check the eBook [Containerized Docker Application lifecycle with Microsoft Platforms and Tools](#) for an explicit non-Visual Studio details.

When using **Visual Studio 2015** or **2017**, many of those steps are transparent so it dramatically improves your productivity. This is especially true when 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* to your projects with the needed configuration, or VS builds the Docker image and runs the multi-container application directly in Docker after hitting F5 or it even allows you to debug several containers at once. Those features will boost your development speed.

However, making transparent those steps doesn't mean that you don't need to know what's going on underneath with Docker, this is why every step is detailed in the following step-by-step guidance.

But, yes, Visual Studio simplifies that workflow to "the minimum" as explained in the next sections.



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

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

Setup of your local environment

With the latest version of **Docker for Windows**, it is easier than ever to develop Docker applications, as the setup is straight forward, as explained in the following reference.

Installing Docker for Windows: <https://docs.docker.com/docker-for-windows/>

In addition, you'll need to have installed Visual Studio 2015 with the tools for Docker or Visual Studio 2017 which comes with tooling for Docker built-in when selecting the ".NET Core and Docker workload" when you install it, as shown in the image X-X.

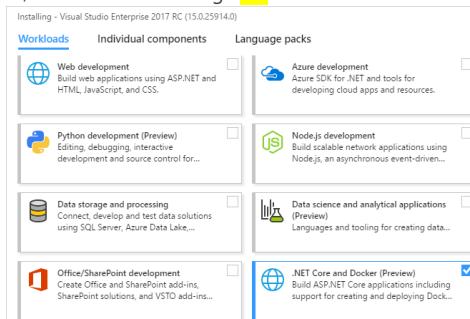


Figure X-X. Selecting the Docker and .NET Core workload

Visual Studio 2017 RC: <https://www.visualstudio.com/vs/visual-studio-2017-rc/>

Visual Studio 2015: <https://www.visualstudio.com/vs/>

Visual Studio Tools for Docker:

<http://aka.ms/vstoolsfordocker>

<https://docs.microsoft.com/en-us/dotnet/articles/core/docker/visual-studio-tools-for-docker>

Working with .NET and Visual Studio

You can start coding your app in .NET (usually .NET Core if you are going to go to containers) even before enabling Docker in your app and deploying/testing in Docker, however, the recommendation is to start working on Docker as soon as possible as that will be the real environment and any issue is worth to be catch as soon as possible. This is very much encouraged because Visual Studio makes it so easy to work with Docker that it almost feels transparent even with debugging support with multi-container applications.



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

You will need a *dockerfile* per custom image to be built and per container to be deployed, therefore, if your app is made up by a single custom service, you will need a single *dockerfile*. But if your app is composed by multiple services (like in a microservices architecture), you'll need one *dockerfile* per service.

The *dockerfile* is usually placed within the root folder of your app/service and contains the required commands so Docker knows how to setup up and run your app/service. You can create manually (typing code) your *dockerfile* and add it to your project along with your .NET, however, with Visual Studio and its tools for Docker, it is as simple as a few mouse clicks.

If you create a new project in VS 2017, there's a new check-box option named "Enable Container (Docker) Support" as highlighted in figure X-X.

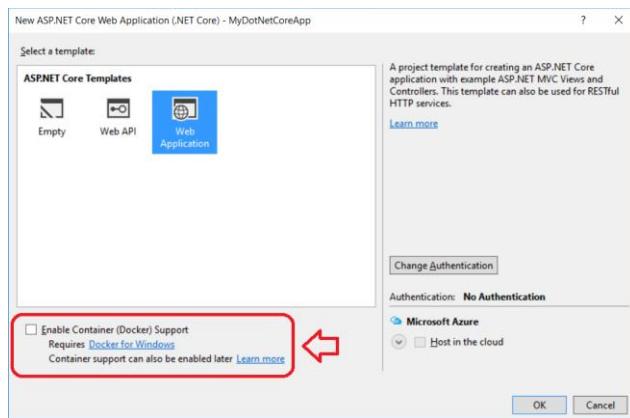


Figure X-X. Enabling Docker Support when creating a new project

The other choice you can take on new or even an already created project is to enable Docker support on your existing project by simply right clicking on your project file in VS and selecting the menu option "Add-Docker Project Support" if your app is made by a single project/service or "Add-Docker Solution support" if your app is a multi-container application, as shown in figure X-X.

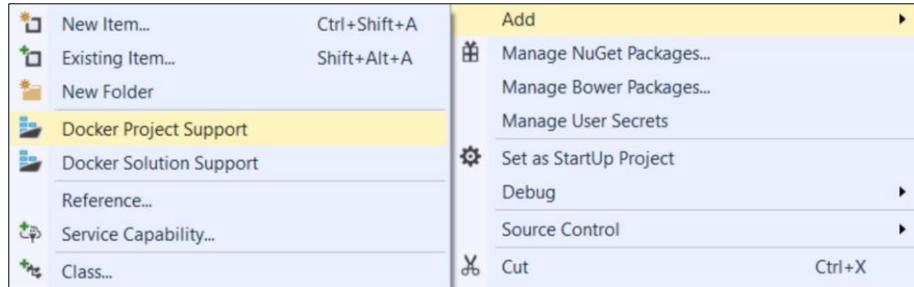


Figure X-XX. Enabling Docker support in a Visual Studio 2017 project

That simple action on a single project (single container application) will add a *dockerfile* to your project with the required configuration so you might not need to do anything else. However, the following is what happened under the covers when Visual Studio created the dockerfile for you.

Option A - Using an existing official .NET Docker image

You usually build your custom image for your container on top of a base-image you can get from any official repository at the [Docker Hub registry](#). In previous sections in this paper it was explained what Docker images and repos you can have depending on the chose framework and OS. For instance, if you chose to use ASP.NET Core and Linux, the image to be use would be "microsoft/aspnetcore:1.1.0". Therefore, you just need to specify what base Docker image you'll be using for your container by writing that in your *dockerfile*, like setting "FROM microsoft/aspnetcore:1.1.0" in your dockerfile.

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).

For instance, a sample **dockerfile** for an ASP.NET Core container would be the following:

```
Dockerfile
1 FROM microsoft/aspnetcore:1.1.0
2 ARG source
3 WORKDIR /app
4 EXPOSE 80
5 COPY ${source:-bin/Release/PublishOutput} .
6 ENTRYPOINT ["dotnet", "MySingleContainerApp.dll"]
```

Figure X-XX. Sample Dockerfile for a .NET Core container

In this case, it is using the version 1.1.0 of the official ASP.NET Core Docker image for Linux named "[microsoft/aspnetcore:1.1.0](#)". For further details, consult 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 to the TCP port you will use at runtime (like port 80, in this case).

There are other lines of configuration you can add in the Dockerfile depending on the language/framework you are using, so Docker knows how to run the app. For instance, the ENTRYPPOINT line with ["dotnet", "MySingleContainerApp.dll"] is needed to run a .NET Core app, although you can have multiple variants depending on the approach to build and run your service. If using the SDK and dotnet CLI to build and run the .NET app it would be slightly different. The bottom line is that the ENTRYPPOINT line plus additional lines will be different depending on the language/platform you choose for your application.

References - Base Docker images

Building Docker Images for .NET Core Applications

<https://docs.microsoft.com/en-us/dotnet/articles/core/docker/building-net-docker-images>

Build your own images

<https://docs.docker.com/engine/tutorials/dockerimages/>

Multi-Platform Image repositories

As Windows containers become more prevalent, a single repo can contain platform variants, such as a Linux and Windows image. This feature allows vendors to use a single repo to cover multiple platforms. Such as [microsoft/dotnet](#) repository available at DockerHub registry which provides support for Linux and Windows Nano Server by using the same repo name with different tags, as shown in the following examples.

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

Going further in the future, it probably will be possible to use exactly the same repo name and tag so when pulling an image from a Windows host it will pull the Windows variant. While pulling the same image name from a Linux host will pull the Linux variant.

Option B - Create your base-image from scratch

You can create your own Docker base image from scratch as explained in this [article](#) from Docker. This is a scenario that is probably not for people starting with Docker, but if you want to set the specific bits of your own base image, you can do so.



Step 3. Create your custom Docker images embedding your service in it

Per each custom service you may have composing your app, you'll need to create its related image. If your app is made up of a single service or web-app, then you just need a single image.

Each developer needs to develop and test locally until you push a complete feature or change to the source control system (Git, etc.). This means that you need to create the Docker images and deploy your containers to a local Docker host (like a Linux VM) and run/test/debug against those containers.

In order to create a custom image in your local environment by using Docker CLI and your *dockerfile*, you can do it by using the “**docker build**” command, as in the following example (You can also run “**docker-compose up --build**” for applications composed by several containers/services).

Optionally, instead of directly running “**docker build**” from the project’s folder, you can first generate a deployable folder with the .NET libraries/binaries needed with **run dotnet publish**, and then run “**docker build**”:

```
PS C:\dev\dotnetcore-webapi-microservice-docker> docker build -t cesardl/netcore-webapi-microservice-docker:first .
Sending build context to Docker daemon 1.148 MB
Step 1/10 : FROM microsoft/dotnet:latest
latest: Pulling from microsoft/dotnet
5c90d4a2d1a8: Downloading [=====] 18.34 MB/51.35 MB
ab30c63719b1: Downloading [=====] 18.48 MB/18.55 MB
c6072700a242: Downloading [=====] 18.34 MB/42.53 MB
121d7eef6c20: waiting
eb57cf4f29ee: waiting
b2c5ae2d325b: waiting
```

Figure X-XX. Creating a custom Docker Image

It will create a Docker image with the name “**cesardl/netcore-webapi-microservice-docker:first**” (“:first” is a tag, like a specific version). You can take this step for each custom image you need to create for your composed Docker application with several containers.

You can find the existing images in your local repository (your dev machine) using “**docker images**” command.

```
PS C:\dev\dotnetcore-webapi-microservice-docker> docker images
REPOSITORY          TAG      IMAGE ID      CREATED       SIZE
cesardl/netcore-webapi-microservice-docker    first   384c4ac1809b  4 minutes ago  579.8 MB
microsoft/dotnet        latest  49aaaf5daa850  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 X-XX. Viewing existing images using

Creating Docker Images with Visual Studio

When you are using Visual Studio and a project with Docker support, you don't explicitly create an image, it will be created for you under the covers when you just hit F5 and run the “*dockerized*” application or service. Therefore, this step will be “transparent” when working with VS, but it is 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 with multiple services

With the “`docker-compose.yml`” file you can define a set of related services to be deployed as a composed application with the deployment commands explained in the next step section.

You need to create that file in your main or root solution folder, with a similar content to the following file `docker-compose.yml`.

```
docker-compose.yml * ✘
1 version: '2'
2
3 services:
4   webmvc:
5     image: eshop/web:latest
6     environment:
7       - CatalogUrl=http://catalog.api
8       - OrderingUrl=http://ordering.api
9     ports:
10      - "800:80"
11     depends_on:
12       - ordering.api
13       - basket.api
14
15   ordering.api:
16     image: eshop/ordering.api:latest
17     environment:
18       - ConnectionString=Server=ordering.data;Database=
19     ports:
20      - "81:80"
21     depends_on:
22       - ordering.data
23
24   ordering.data:
25     image: eshop/ordering.data.sqlserver.linux
26     ports:
27      - "1433:1433"
28
29   basket.api:
30     image: eshop/basket.api:latest
31     environment:
32       - ConnectionString=basket.data
33     depends_on:
34       - basket.data
35
36   basket.data:
37     image: redis
```

Figure X-XX. Example “`docker-compose.yml`” file for a multi-container based app

In this particular case, this `docker-compose.yml` file defines five services. The “`webmvc`” service (a web app), two microservices (`ordering.api` and `basket.api`) and two datasource containers, one (named `ordering.data`) based on SQL Server for Linux running as a container and a second container (named `basket.data`) with a Redis cache service. Each service will be deployed as a container, so we need to use a concrete Docker image for each.

For instance, for that particular “`webmvc`” service:

- Builds from the Dockerfile in the current directory.
- Uses two environment variables initialized in this file
- Forwards the exposed port 80 on the container to port 8000 on the host machine.
- Explicitly links the web service to the basket and ordering service with “depends_on” so it will wait for those services until they are started.

We will re-visit this docker-compose.yml file in the next sections when tackling on microservices and multi-container apps.

Working with docker-compose.yml from Visual Studio 2017

When you are using Visual Studio and add “Docker Solution Support” to a service project in your solution, VS is not just adding a dockerfile file to your project but it is also adding a service section in your solution docker-compose.yml file (or creating the file if it didn’t exist). It is precisely a very easy way to start composing your multiple-container solution and you can then open the docker-compose.yml file and update it with additional features.

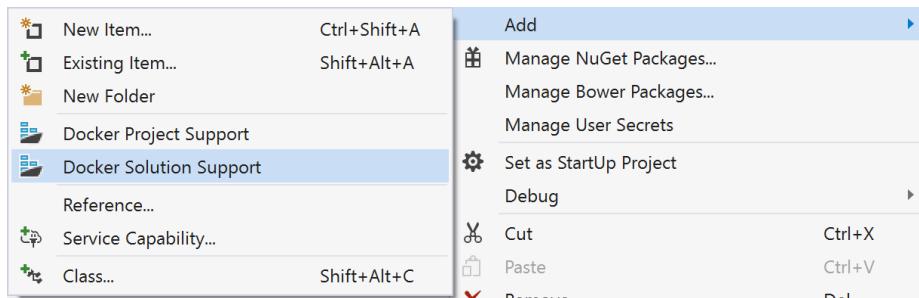


Figure X-XX. Enabling Docker Solution support in a Visual Studio 2017

That simple action not only will add the *dockerfile* to your project (as Docker Project Support does, as well) but also will add the required configuration lines of code to a global docker-compose.yml set at the solution level, similar to the docker-compose.yml previously shown.



Step 5. Build and run your Docker app

If your app only has a single container, you just need to run it by deploying it to your Docker Host (VM or physical server). However, if your app is made by multiple services, you need to “compose it”, too. Let’s see the different options.

Option A. Running a single container

Running a single container with Docker CLI

You can run the Docker container using “**docker run**” command, as the following execution.

```
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 X-XX. Code example – running a Docker container using the "docker run" command

Note that for this particular deployment, I'll be redirecting requests sent to port 80 to the internal port 5000. So now, the application is listening on the external port 80 at the host level.

Running a single container with Visual Studio

When using Visual Studio 2015 (with Docker tools installed) or Visual Studio 2017, it is even simpler. You just need to hit F5 or press the "Docker Play" button on the tool bar. Under the covers, VS will create the Docker image, deploy and run it in your Docker host.

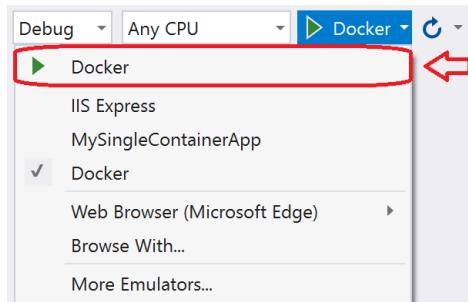


Figure X-XX. Running a Docker container using Visual Studio

Option B. Running a multi-container application

In most enterprise scenarios, a Docker application will be composed by multiple services which means you need to run a multi-container application as in figure X-XX.

Running a multi-container application with Docker CLI

In this case, you can execute the command "**docker-compose up**" that will use the "docker-compose.yml" file that you might have at the solution level, so it deploys a composed application with all its related containers, as in the following example when running the command from your main project directory containing 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 5-21. Example results when running the "docker-compose up" command

After running "docker-compose up", you would have your application and its related container/s deployed into your Docker Host, like illustrated in Figure 5-20 in the VM representation.

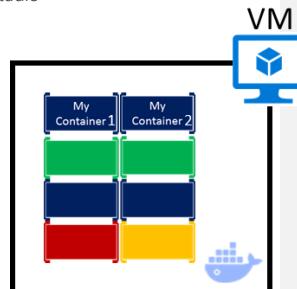


Figure X-XX. VM with Docker containers

Running and debugging a multi-container application with Visual Studio

Again, when using Visual Studio 2017 it cannot get simpler and you are not only running the multi-container application but being able to debug all of its containers at once.

As mentioned before, each time you add “Docker Solution Support” to a specific project within a solution, you will get that project configured in the global/solution docker-compose.yml, so you will be able to run or debug the whole solution at once because Visual Studio will spin up a container per project that has Docker solution Support enabled while creating 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 5-26, in **Visual Studio 2017** you have an additional F5 button that we have added so you can run or debug a whole multiple container application by running all the containers that are defined in the docker-compose.yml file at the solution level that was modified by Visual Studio while adding “Docker Solution Support” to each of your projects. This means that you could set several breakpoints up, each breakpoint in a different project/container and while debugging from Visual Studio you will be stopping in breakpoints defined in different projects and running on different containers.

For further details on the services implementation and deployment to a Docker host, read the following articles.

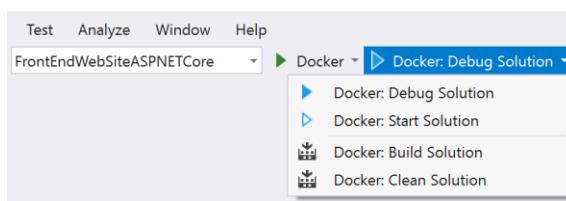


Figure X-XX. Running multi-container apps in Visual Studio 2017

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/>

IMPORTANT NOTE: “`docker-compose up`” and “`docker run`” (or running/debugging the containers from VS which is under the covers using the same techniques) might be enough for testing your containers in your development environment, but might not be used at all if you are targeting Docker clusters and orchestrators like **Docker Swarm**, **Mesosphere DC/OS** or **Kubernetes**, in order to be able to scale-up. If using a cluster, like **Docker Swarm mode** (available in *Docker for Windows and Mac* since version 1.12), you need to deploy and test with additional commands like “`docker service create`” for single services or when deploying an app composed by several containers, using “`docker compose bundle`” and “`docker deploy myBundleFile`”, by deploying the composed app as a “stack” as explained in the article [Distributed Application Bundles](#), from Docker.

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 (locally, in your local CD VM)

This step will vary depending on what is your app doing.

In a very simple .NET Core Web API hello world deployed as a single container/service, you'd just need to access the service by providing the TCP port specified in the dockerfile, as in the following simple example.

If "localhost" is not enabled, to navigate to your service, find the IP address for the machine with this command:

```
docker-machine ip your-container-name
```

Open a browser on the Docker host and navigate to that site, and you should see your app/service running.



Figure 5-22. Example of testing your Docker application locally using localhost

Note that it is using the port 80 but internally it was being redirected to the port 5000, because that's how it was deployed with "docker run", as explained in a previous step.

It can also be tested with CURL, from the terminal. In a Docker installation on Windows, the default IP is 10.0.75.1.

```
PS C:\dev\dotnetcore-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 5-23. Example of testing your Docker application locally using CURL

Testing and Debugging containers with Visual Studio

As mentioned, when running/debugging the containers with VS you'll be able to debug the .NET application running on containers in a similar way than you could do when running on the plain OS.

For further details on how to debug containers, read the following articles.

Build, Debug, Update and Refresh apps in a local Docker container:
<https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-docker-edit-and-refresh/>

Simplified workflow when developing containers with Visual Studio

Effectively, the workflow when using Visual Studio is a lot simpler than a regular Docker container development process because most of the steps required by Docker related to dockerfile and docker-compose.yml are hidden or simplified by VS, as shown in the image X-XX.

VS development workflow for Docker apps

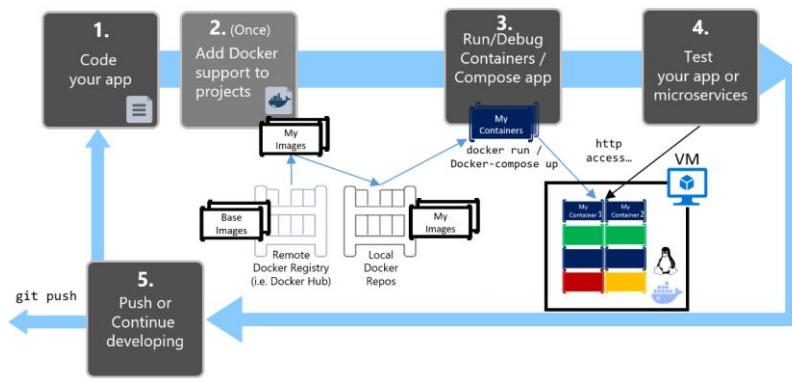


Figure X-XX. Simplified workflow when developing with Visual Studio

Even further, the step number 2, “Add Docker support to your projects” needs to be done just once. So usually that process or workflow remains pretty similar to your usual development tasks when using plain .NET. However, you still need to know what’s going on under the covers (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* when customizing the behaviors. But, for the most part of the work, it’ll be greatly simplified by Visual Studio, making you a lot more productive.

Using PowerShell commands in Dockerfile to setup Windows Containers (Docker standard based)

[Windows Containers](#) allows you to convert your existing Windows applications into Docker images and deploy them with the same tools as the rest of the Docker ecosystem.

In order to use **Windows Containers**, you just need to write **PowerShell commands** in the Dockerfile, as the following example.

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

Figure 5-27. Code example – running Dockerfile PowerShell commands

In this case, it is using a Windows Server Core base image plus installing IIS with a PowerShell command. In a similar way, you could also use PowerShell commands and setup additional components like the traditional ASP.NET 4.x and .NET 4.6 or any other Windows software. For example: **RUN powershell add-windowsfeature web-asp-net45**

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 being following a microservice architecture.

In a similar page, another question to be answered is, "is Docker necessary when building a microservice architecture?". The answer is a clear "No". Docker is an enabler and can give 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 architecture based application (using microservice patterns as explained in the microservice introduction section in this paper) that is also based on Docker containers as its unit of deployment, you will be able to design and develop any other simpler application model, like a 3-tier application that also requires a multi-container approach. Because of that fact and because microservice architectures are an important trend within the container's world, this section focuses on a microservice architecture implementation using Docker containers.

Designing a microservice oriented application

Application context

You have to develop a server-side enterprise application. It must support a variety of different clients including desktop browsers running SPA (Single Page Applications) or traditional web apps, mobile web apps and native mobile apps. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either http services or a message bus. The

application handles requests by executing business logic, accessing databases and returning HTML/JSON/XML responses.

The application will consist of multiple types of components:

- Presentation components - responsible for handling the UI plus consuming remote services
- Domain/business logic - the application's domain logic
- Database access logic - data access components responsible for access to databases (SQL or NO-SQL)
- Application integration logic - messaging layer, possible service buses, etc.

The application will have requisites of high scalability but probably, certain sub-systems will need higher 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) very easily.

Development team context

- You have multiple dev teams focusing on different business areas of the application
- New team members must quickly become productive and the application must be easy to understand and modify
- The application will have a long-term evolution with "ever-changing business rules"
- You need a good long-term maintainability which means having agility when implementing new changes in the future while being able to update multiple sub-systems with minimum impact on the other sub-systems. The application must be easy to understand and modify.
- 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 in the long term, but you don't want to make full migrations of the application when moving to new technologies as that would bring high costs and impact predictability and stability of the application.

Problem

What is going to be the application deployment architecture?

Solution

Architect the application decomposing it in many autonomous sub-systems in the form of collaborating microservices and containers (each microservice would be a container).

Each service/container implements a set of 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 is possible (especially when propagating changes/updates).

Microservices are developed and deployed as containers independently of one another which means that a development team can be developing and deploying a certain microservice/container without impacting other sub-systems.

Each microservice has its own database in order 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) like handled in CQRS (Command and Query Responsibility Segregation). Because of that, the business constraints have to embrace eventual consistency between the multiple microservices and related databases.

eShopOnContainers - Reference app for .NET Core and microservices/containers

For the sake of a well-known business domain, so you can focus on the architecture and technologies instead of wondering about the business domain, we have selected a simplified ecommerce or e-shop application that visualizes a catalog of products, takes orders from customers, verifies inventory and other business features. This container-based application's source code is available at GitHub.

Source code – eShopOnContainers reference app (.NET Core & microservices/containers)
<https://aka.ms/eShopOnContainers/>

The application consists of multiple sub-systems including several store UI front-ends (Web app and native mobile app) along with the backend microservices/containers for all the required server-side operations, as shown in figure X-XX.

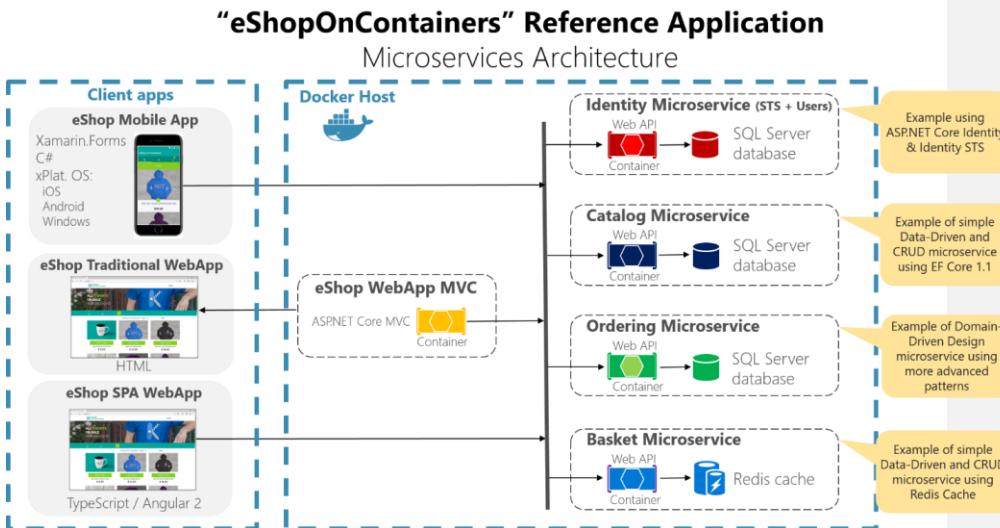


Figure X-XX. eShopOnContainers reference app – Using Direct Client-to-Microservice Communication

Hosting environment: In the architecture diagram shown you see several containers deployed within a single Docker Host. That would be the case when deploying to a single Docker Host with "docker-compose up". However, if using any 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

explained in the architecture section when introducing orchestrators and clusters like the ones available in *Azure Container Service (Docker Swarm, Kubernetes or DC/OS)* or *Azure Service Fabric*.

Communication architecture – Initially using “Direct Client-to-Microservice Communication”

The application will be deployed as a set of microservices in the form of containers and client apps can communicate with those containers as well as communicate between microservices/containers. Something important to highlight is that 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 will have a public endpoint like <https://servicename.applicationname.companynname>, or even using a different TCP port per microservice. In production, that URL would map to the microservice’s load balancer, which distributes requests across the available instances.

As mentioned and explained in the preliminary architecture section of this document, the “*Direct Client-To-Microservice communication architecture*” can have possible drawbacks when building a large and complex microservice-based application, but it can be good enough for a small application as the “eShopOnContainers” example 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 tens of microservices, we strongly recommend to consider the “API Gateway pattern” explained in the architecture section.

Data Sovereignty Per Microservice

In regards data, each microservice will “own” its own database or data source. Each database or data source will be deployed as another container. This design decision is made only because this application is a *sample reference application* and an important goal is that any developer should be able to just grab the code from GitHub, clone it, open it in Visual Studio or simply take a look with VS Code and compile/generate the custom Docker images using .NET Core CLI and Docker CLI and be able to deploy/run it in a Docker development environment in a matter of minutes without having to provision any external database like SQL Server, Azure SQL Database or any other data source with hard dependencies on infrastructure (in the cloud or on-premises). However, consider that in a real production environment this design decision should change and because of high availability and scalability reasons, the databases should be based on database servers in the cloud or on-premises.

Therefore, the units of deployment for microservices (and even for databases in this application) are Docker containers and it will be indeed a multi-container application which is also embracing the microservices principles.

Benefits

A microservice based solution like this has a number of benefits:

- **Each microservice is relatively small, easy to manage and evolve:**
 - Easier for a developer to understand and get started quickly with good productivity
 - The container starts faster, which makes developers more productive, and speeds up deployments

- The IDE is faster loading and managing smaller projects, making developers more productive
- **Each service can be developed and deployed independently of other services** - easier to deploy new versions of services frequently.
- **It is now possible to scale-out just certain areas of the application.** For instance, just the catalog service or the basket service need to scale-out much more than the ordering process. The resulting infrastructure will be much more efficient in regards resources being used when scaling out.
- **It enables you to organize the development effort around multiple teams.** Each service is owned by a single dev team. Each team can develop, deploy and scale their service independently of all of the other teams.
- **Improved issues isolation.** For example, if there is a bug or issue in one service then only that service will be initially impacted. The other services will continue to handle requests. In comparison, one malfunctioning component in a monolithic deployment architecture can bring down the entire system when it is related to resources, like memory leaks. Additionally, when the bug/issue is solved, you can deploy just the affected microservice without impacting the rest of the already running microservices.
- **Can use the latest technologies.** Because you can start developing autonomous services independently and running side-by-side, that allows you to start using the latest technologies and frameworks instead of being stuck on an older stack/framework for the whole application.

Drawbacks

A microservice based solution like this has a number of possible drawbacks:

- It is a **distributed system**, that brings additional complexity to be handled by developers when designing and building the applications.
 - Developers must implement inter-service communications which adds complexity in regards testing, exception handling and also adds additional latency to the system
- **Deployment complexity.** In production, there is also the operational complexity of deploying and managing a system comprised of many different service types. If not using a microservice oriented infrastructure (like the orchestrators and schedulers introduced in previous sections in this document), that additional complexity can take more development efforts than the business application itself.
- **Atomic transactions** between multiple microservices is not usually possible. The business requirements have to embrace eventual consistency between the multiple microservices.
- **Increased global resources consumption** (memory, drives, network). The microservices architecture replaces a number N of monolithic application instances (i.e. 10 monolithic instances) with NxM services instances (i.e. 8 microservices per application instance). If each service runs in its own .NET Core framework, which is usually good in order to isolate the instances, then there is the overhead of M times as many .NET Core runtimes (80 vs 10). However, given the cheap cost of resources in total and the benefit of being able to scale-out

just certain areas of the application compared to long-term costs when evolving monolithic applications, this is usually something that can be assumed by large and long-term applications

- **Issues in the “Direct Client-to-Microservice communication” approach:** When the application is large with tens of microservices, there are challenges and limitations with this option. One problem is the mismatch between the needs of the client and the fine-grained APIs exposed by each of the microservices. In certain cases, the client app might need to make many separate requests per page/screen. While a client could make that many requests, it would probably be too inefficient over the public Internet and would definitely be impractical over a mobile network, so requests from the client app to the backend system should be minimized.
 - Another problem with the client directly calling the microservices is that some microservices might be using non-web-friendly protocols. One service might use a binary communication while another service might use AMQP messaging protocol. Those protocols are not firewall-friendly and is best used internally. An application should use protocols such as HTTP and WebSocket outside of the firewall.
 - Another drawback with this approach is that it makes it difficult to refactor the contracts of those microservices. Over time we might want to change how the system is partitioned into services. For example, we might merge two services or split a service into two or more services. If, however, clients communicate directly with the services, then performing this kind of refactoring can break compatibility with client apps.

As mentioned in the architecture section, when designing and building a large and complex application based on microservices you would probably want to consider the “API Gateway” pattern instead of the simpler “Direct Client-to-Microservice communication” approach.

Finally, another challenge that happens no matter what approach you take is deciding how to partition the system into microservices. This is very much an art, but there are a number of strategies that can help. Basically, you need to identify areas of the application that are decoupled from the other areas with a low number of hard dependencies. In many cases this is aligned to partition services by use case. For example, in our e-Shop application we have the Ordering service that is responsible for all the business logic related to the order process. Then you also have the catalog service and the basket service implementing other differentiated capabilities. Ideally, each service should have only a small set of responsibilities kind of 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 might be a bit larger than a single class and most of all has to be completely autonomous, end to end, including responsibility on its data sources.

External vs. Internal Architecture and Design Patterns

This is another important subject to discuss. The external architecture is precisely the microservice architecture composed by multiple service and following the already explained principles in the architecture section of this document. However, depending on the nature of each microservice and independently of your chosen high-level microservice architecture, it is common and advisable to have a different internal architecture and patterns implementation per microservice, potentially even using different technologies and programming languages (C# vs. F# vs. any platform) as illustrated in figure X-XX.

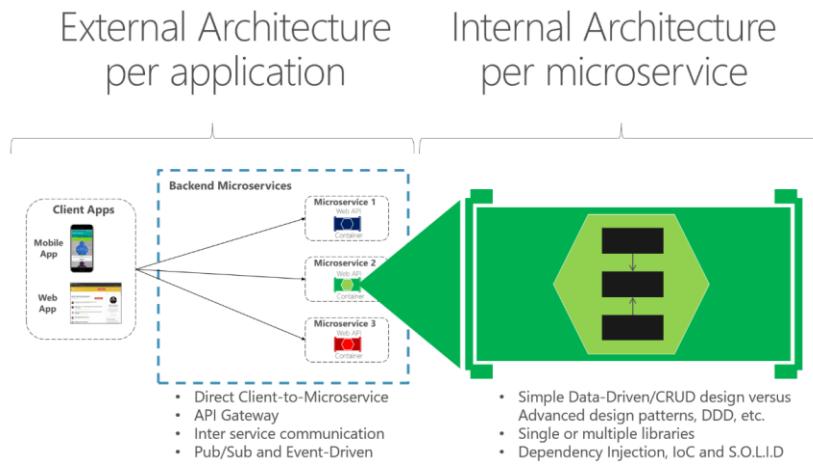


Figure X-XX. External vs. Internal Architecture and Design

For instance, in our initial eShop sample system, the catalog, basket and user profile microservices are pretty simple and basically CRUD sub-systems, therefore, its internal architecture and design will be pretty straight forward. However, you might have other microservices like in our case the Ordering microservice which can hold further complexity and "ever-changing business rules" with a high degree of domain/business complexity. In those cases, you might want to implement more advanced patterns within that particular microservice, like the ones defined in Domain-Driven Design approaches, as we are precisely doing in the eShop Ordering microservice. You will be able to review these DDD patterns in the section explaining the implementation of the eShop Ordering microservice.

Another example of different implementation and technology per microservice could be related to the nature of the microservice. For certain particular domain logic, might be a much better implementation if you use a functional programming language like F# or even a language like R when targeting AI and machine learning domains, instead of a more object-oriented programming language like C# which could be good for many other microservices.

The bottom line is that each microservice could follow a different internal architecture and different design patterns. Not all microservices should be implemented using advanced DDD patterns as that would be overengineered, and in a similar way, complex microservices with a lot of ever-changing business logic shouldn't be implemented as CRUD components or you will end up with spaghetti code with very low quality.

Creating a simple data-driven/CRUD microservice

Designing a simple data-driven/CRUD microservice

From a design point of view, this type of containerized microservice should be as simple as possible while providing good development productivity.

An example of this kind of service is the Catalog microservice from the eShopOnContainers sample application. This type of service implements all its functionality within a single ASP.NET Core Web API project, including classes for its data model and any required business logic and data access code. In addition to that, you can have its related data and database running in a SQL Server container as shown in the design diagram in figure X-XX.

Data-Driven/CRUD microservice container

Sample design

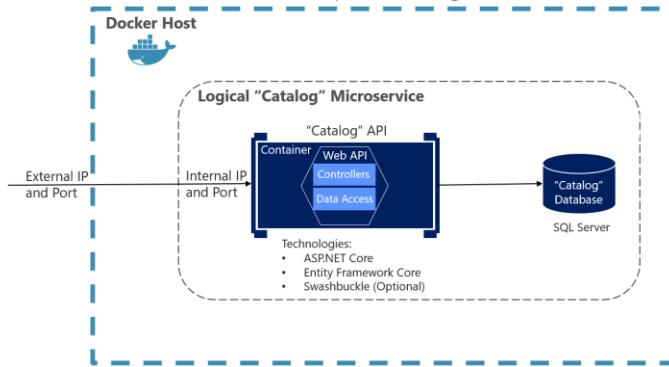


Figure X-XX. Simple data-driven/CRUD microservice design diagram

When developing this API you just need to use [ASP.NET Core](#) and any data access API or ORM like [Entity Framework Core](#). You could also generate [Swagger](#) metadata automatically through [Swashbuckle](#), so you provide a description of what your service offers, as explained in the next section when developing this service.

Note that running a database server like SQL Server within a Docker container is great for development environments as you can have all your dependencies up and running without needed to provision any database in the cloud or in an on-premises server and very convenient when running integration tests. However, for production environments running a database server on a container is not a recommended environment as you usually won't have 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 HA and HS, like DocumentDB if using a NO-SQL approach.

Finally, by editing the `dockerfile` and `docker-compose.yml` metadata files you are able to configure how the image of this container will be created and what base image it will use plus design settings like internal and external names and TCP ports used.

Implementing a simple CRUD microservice with ASP.NET Core

When implementing this type of service using .NET Core and Visual Studio, you just need to start by creating a simple ASP.NET Core (running on .NET Core so it can run on the Linux Docker host) Web API as shown in figure X-X.

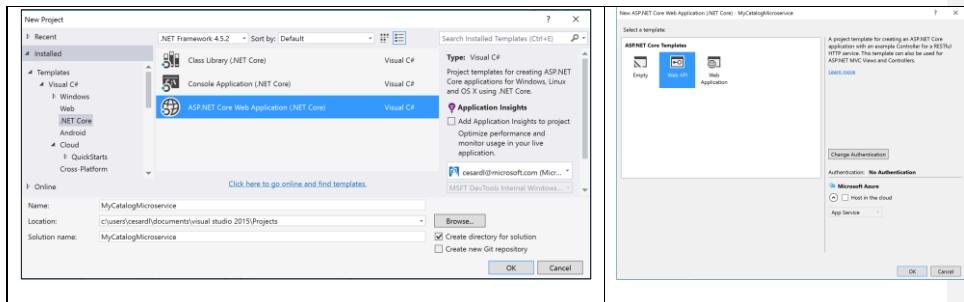


Figure X-XX. Creating an ASP.NET Core Web API project in VS 2015

After creating the project, you can implement your MVC controllers like in any other Web API project, using 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 about ASP.NET Core itself, Entity Framework and Swashbuckle:

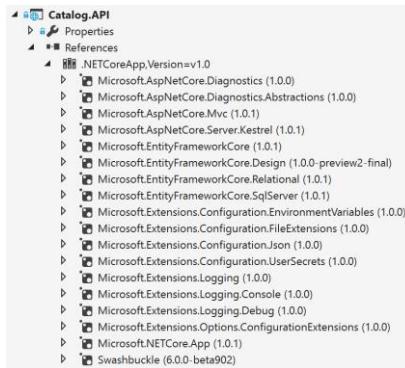


Figure X-XX. Dependencies in a simple CRUD Web API microservice

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 (O/RM) that enables .NET developers to work with a database using .NET objects.

In the case of the "Catalog" microservice, it is using 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, like Windows on-premises or Azure SQL DB. The only thing you would need to change is the connection string being used from 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 Visual Studio UI or with the NuGet console:

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

The model

With EF Core, data access is performed 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, hand 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). In the case of the "Catalog" microservice we are using the last approach. You can see an example of the Product entity class in figure X-X which is a simple [POCO](#) (Plain Old CLR Class) entity class.

```
14 references | Unix, 25 days ago | 3 author, 1 change
public class CatalogItem
{
    1 reference | Unix, 25 days ago | 1 author, 1 change
    public int Id { get; set; }

    8 references | Unix, 25 days ago | 1 author, 1 change
    public string Name { get; set; }

    4 references | Unix, 25 days ago | 1 author, 1 change
    public string Description { get; set; }

    5 references | Unix, 25 days ago | 1 author, 1 change
    public decimal Price { get; set; }

    5 references | Unix, 25 days ago | 1 author, 1 change
    public string PictureUrl { get; set; }

    6 references | Unix, 25 days ago | 1 author, 1 change
    public int CatalogTypeId { get; set; }

    1 reference | Unix, 25 days ago | 1 author, 1 change
    public CatalogType CatalogType { get; set; }

    6 references | Unix, 25 days ago | 1 author, 1 change
    public int CatalogBrandId { get; set; }

    1 reference | Unix, 25 days ago | 1 author, 1 change
    public CatalogBrand CatalogBrand { get; set; }

    4 references | Unix, 25 days ago | 1 author, 1 change
    public CatalogItem() { }
}
```

Figure X-XX. Sample POCO Entity class: CatalogItem

You also need the mentioned **DbContext** that represents a session with the database. In the case for the "Catalog" microservice, it is the CatalogContext class deriving from the DbContext base class, as shown below in figure X-XX.

```

using EntityFrameworkCore.Metadata.Builders;
using Microsoft.EntityFrameworkCore;

9 references | Unai, 25 days ago | 1 author, 3 changes
public class CatalogContext : DbContext
{
    0 references | Unai, 25 days ago | 1 author, 1 change
    public CatalogContext(DbContextOptions options) : base(options)
    {
    }
    7 references | Unai, 25 days ago | 1 author, 1 change
    public DbSet<CatalogItem> CatalogItems { get; set; }
    3 references | Unai, 25 days ago | 1 author, 1 change
    public DbSet<CatalogBrand> CatalogBrands { get; set; }
    3 references | Unai, 25 days ago | 1 author, 1 change
    public DbSet<CatalogType> CatalogTypes { get; set; }

```

Figure X-XX. Sample DbContext class: CatalogContext

You can have additional code within the DbContext implementation like the method "OnModelCreating()" being used in the CatalogContext class that manages the sample data it will automatically populate the first time it tries to access the database, useful for demo data.

Querying data from Web API controllers

Instances of your entity classes are usually retrieved from the database using Language Integrated Query (LINQ). See [Querying Data](#) to learn more.

```

[Route("api/v1/[controller]")]
1 reference | Unai, 14 days ago | 2 authors, 5 changes
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _context;

    0 references | glennnc, 75 days ago | 1 author, 1 change
    public CatalogController(CatalogContext context)
    {
        _context = context;
    }

    // GET api/v1/[controller]/items/?pageSize=3&pageIndex=10
    [HttpGet]
    [Route("[action]")]
    0 references | Unai, 14 days ago | 1 author, 2 changes
    public async Task<IActionResult> Items(int pageSize = 10, int pageIndex = 0)
    {
        var totalItems = await _context.CatalogItems
            .LongCountAsync();

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

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

        return Ok(model);
    }
}

```

Figure X-XX. Querying data from a Web API controller

Saving data

Data is created, deleted, and modified in the database using instances of your entity classes. See [Saving Data](#) to learn more. You can add code similar to the following to your Web API controllers.

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

Figure X-XX. Saving Data

Dependency Injection in ASP.NET Core and Web API controllers

Note that in ASP.NET Core you can use Dependency Injection (DI) out-of-the-box. There's no need to setup any third party IoC container (IoC means Inversion of Control), although you can also plug your preferred IoC container into the ASP.NET Core infrastructure, if you'd like. In this case, it means that you can directly inject the needed EF DBContext or additional repositories through the controller constructor. In the figure X-XX above we are injecting an object of CatalogContext type.

An important configuration to set up at the Web API project is the DbConext class registration into the services IoC container. You usually do that at the **Startup.cs** class and the method **ConfigureServices()**, with the method **services.AddDbContext()**, as shown in figure X-XX.

```
0 references | Carlos Calizares Estévez, 5 days ago | 4 authors, 6 changes  
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddSingleton< IConfiguration>(Configuration);  
  
    services.AddDbContext< CatalogContext >(c =>  
    {  
        c.UseSqlServer(Configuration["ConnectionString"]);  
        c.ConfigureWarnings(wb =>  
        {  
            //By default, in this application, we don't want to have client evaluations  
            wb.Log< RelationalEventId.QueryClientEvaluationWarning>();  
        });  
    });  
}
```

Figure X-XX. Registering a DbContext class for DI use

The DB connection string and environment variables used by Docker containers

Basically, you can use the ASP.NET Core settings and add a "ConnectionString" property to your **settings.json**. Then you can reference that from your docker-compose.yml file configuration, so

```
services:  
  catalog.api:  
    image: eshop/catalog.api  
    build:  
      context: .  
      dockerfile: Dockerfile  
    environment:  
      - ConnectionString=Server=catalog.data;Initial Catalog=CatalogData;  
    expose:  
      - "80"  
    ports:  
      - "5101:80"  
    depends_on:  
      - catalog.data
```

Docker will set it up as an environment variable for you, as shown in the docker-compose.yml file below.

Finally, you can get that value from your code with `"Configuration["ConnectionString"]"` like shown in the method `"ConfigureServices()"` in the code from the image X-XX above.

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

Since this feature should probably be common in any kind of microservice, either Data-Driven microservices or more advanced Domain-Driven microservices (explained in following section), that is why it is placed within the simpler Data-Driven microservice implementation. But you should also implement this feature in the more complex microservices.



[Swagger](#) is a very much used open source framework backed by a large ecosystem of tools that helps you design, build, document, and consume your RESTful APIs. It is probably becoming as the main standard for this domain (APIs description metadata).

The heart of Swagger is the Swagger Specification (API description metadata which is a JSON or YAML file). The specification creates the RESTful contract for your API, detailing all of its resources and operations in 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.

This specification defines the structure for how a service can be discovered and its capabilities understood. More information, a Web Editor, and examples of Swaggers from companies like Spotify, Uber, Slack, Microsoft and many more can be found at <http://swagger.io>

Why to use Swagger

The main reasons why you would want to generate Swagger metadata about your APIs are basically the following:

- **Ability to automatically consume and integrate your APIs with tens of products** and [commercial tools supporting Swagger](#) plus many [libraries and frameworks](#) serving the Swagger ecosystem. Microsoft has high level products and tools that can automatically consume Swagger based APIs, like the following:
 - o [Microsoft Flow](#) – Ability to automatically [use and integrate your API](#) into a high-level Microsoft Flow workflow, with no programming skills required.
 - o [Microsoft PowerApps](#) – Ability to automatically consume your API from [PowerApps mobile apps](#) built with [PowerApps Studio](#), with no programming skills required.
 - o [Azure App Service Logic Apps](#) - Ability to automatically [use and integrate your API into an Azure App Service Logic App](#), with no programming skills required.
- **APIs documentation automatically generated** - When creating large scale RESTful APIs, like when building complex microservice based applications, you will need to handle a large number of endpoints with different data models used in the request/response payloads.

Proper documentation and having a solid API explorer is a crucial pillar for your API success and likability by developers.

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

How to automate API Swagger metadata generation with Swashbuckle NuGet package

Generating Swagger metadata manually (JSON or YAML file) can be a tedious work if you have to write it manually. 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 is seamlessly and automatically adds Swagger metadata to ASP.NET Web API projects. Depending on the package version, it supports ASP.NET Core Web API projects and the traditional ASP.NET Web API and any other "flavor" like Azure API App, Azure Mobile App, Azure Service Fabric microservices based on ASP.NET or plain Web API on containers, as in this case.

Swashbuckle combines ApiExplorer and Swagger/swagger-ui to provide a rich discovery and documentation to 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 slick discovery UI to assist developers with their integration efforts. Best of all, it requires minimal coding and maintenance because it is automatically generated, allowing you to focus on building your API. The final result for the API explorer will look as the image below:

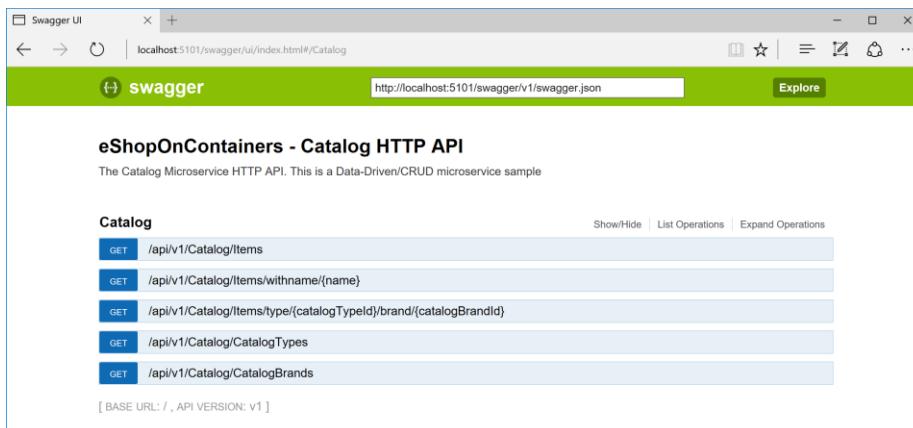


Figure X-XX. Swashbuckle UI based on Swagger metadata – eShop Catalog microservice example

But that UI explorer is not the most important thing here, as mentioned, once you have a Web API that can describe itself in Swagger metadata, your API can seamlessly be used from Swagger-based tools including client proxy classes code generator that can target many platforms, like using [swagger-codegen](#), for example, which allows 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.

Once you have installed those Nuget packages on your Web API project, you will need to configure Swagger in your Startup.cs 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 = "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 should be able to spin up your app and browse the following Swagger JSON and UI endpoints respectively.

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

You previously showed the generated UI created by Swashbuckle with the URL "<http://<your-root-url>/swagger/ui>", but in image X-XX you can also see how you can test any specific API method.

The screenshot shows the Swashbuckle UI for the Catalog/Items API. It includes a table for parameters, a section for response messages (HTTP 200 Success), a 'curl' command, a 'Request URL' field, and a 'Response Body' preview. The response body shows a JSON object with fields like 'pageIndex', 'pageSize', 'count', and 'data' containing vehicle items.

Figure X-XX. Swashbuckle UI testing the Catalog/Items API method

Now, the following image X-XX is the Swagger JSON metadata generated from the eShopOnContainer microservice (which is really what the tools use underneath) when you test it and request that <your-root-url>/swagger/v1/swagger.json URL using the convenient [Postman](#) tool.

The screenshot shows the Postman application interface. A search bar at the top has 'http://localhost:5101/' entered. Below it, a 'GET' request is selected with the URL 'http://localhost:5101/swagger/v1/swagger.json'. The 'Authorization' tab is active. The 'Body' tab shows the raw JSON response, which is the generated Swagger metadata for the API.

Figure X-XX. Swagger JSON metadata

It is that simple, and because it is automatically generated, the Swagger metadata will grow while you add more functionality to your API.

NOTE: Currently, [Swashbuckle 6.0.0](#) version is what you need to use for .NET Core Web API projects which is the normal case when building Docker containers with .NET Core. If using the traditional .NET Framework for Windows Containers, you need to use a different NuGet package version.

References – Swagger and Swashbuckle

ASP.NET Web API Help Pages using Swagger

<https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger>

Creating microservices based on Domain-Driven Design (DDD) and Command-Query Separation (CQS) patterns

Most of the techniques or practices explained for simple data-driven microservices like 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 patterns. Therefore, this section is an extension of the previous sections as most of the practices explained before can also apply here.

However, this section focuses on more advanced microservices that you might want to implement when you need to tackle complexity of certain sub-systems or microservices derived from the knowledge of the Domain experts with a considerable amount of "ever-changing business rules".

DDD vs. DDD patterns

Make no mistake about it, this present guidance is not really covering DDD (Domain-Driven Design) and CQS/CQRS. It is much less ambitious. This section is just covering how you can design certain DDD and CQS architectural approaches and implement them with .NET Core, within a microservice or bounded-context.

There are many DDD patterns like Domain Entity, Aggregates and Aggregate Root, Value Object, Repositories, Factories and so on. But merely applying these patterns doesn't mean you are creating a DDD application or service. It only means that you are applying DDD patterns, really.

DDD is first and foremost about a Domain Model expressed as software. That Domain Model is an attempt to bridge the gap between the software and the real domain and domain experts' knowledge by applying patterns that help transfer a domain reality to a domain model. Techniques like the Ubiquitous Language attempt to help on the fidelity between the real conceptual domain and the software domain model. But, building a robust Ubiquitous Language requires extensive conversations with the domain experts so the developers can learn about the domain. That is really DDD, the process or journey, not the patterns.

Pattern examples are great and it is what this section and sample application ([eShopOnContainers](#)) show you, but that is not DDD. If you're truly looking for "how to do DDD", it's not in any code repository, nor in this short guidance section. This is not capturing real brainstorming or whiteboarding sessions with domain experts.

In order to learn DDD and how to apply it, you can start by reading books like Eric Evan's book, 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>

Commented [CDIT2]: Should we call it CQS or CQRS?
– CQRS is usually related to a more complex design based on separate physical databases and messaging/events and related artifacts to keep consistency between the Writes-DB and the Reads-DB. However, the approach we're taking for this microservices is simpler, just segregating the Queries from the Domain Model, so developers can create queries in a flexible way not limited by the DDD patterns constraints. That's why I'm just saying CQS (Command-Query Separation) although I'm also mentioning CQRS and why it is also pretty similar and you could also say that our approach is a simplified CQRS.. Thoughts?

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](#)

Applying DDD and CQS patterns within a microservice

A simplified example of this kind of service is the “Ordering” microservice from the *eShopOnContainers* reference application. This type of service implements a microservice based on the CQS principle and DDD patterns, as shown in the design diagram in figure X-X.

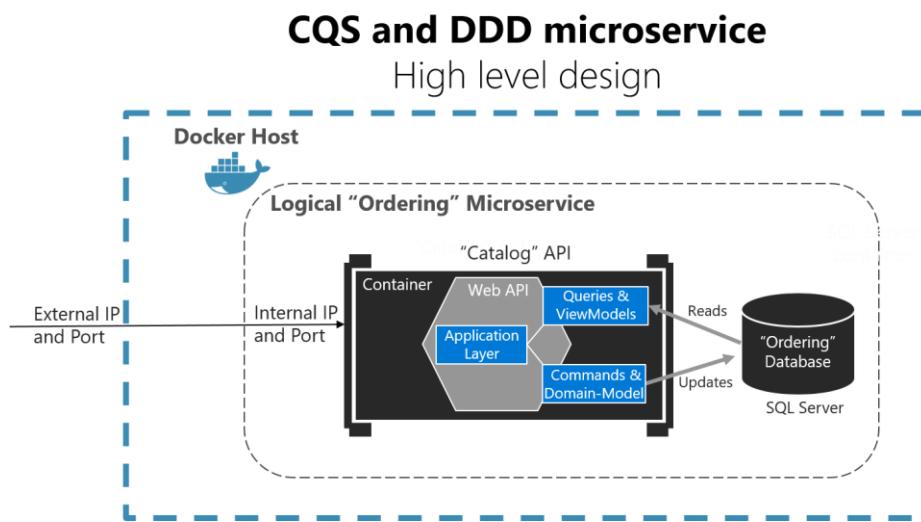


Figure X-X. CQS and DDD based microservice design

The “Application Layer” can be the Web API itself. The important design decision taken 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 ([CQS or Command-Query Separation](#)). 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.

CQS and CQRS approaches in a DDD microservice

The 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 (are free of side effects).
- **Commands:** Change the state of a system.

CQS is a simple concept and can be considered as a principle.

[CQRS \(Command and Query Responsibility Segregation\)](#) was introduced by Greg Young and also promoted by Udi Dahan and other advocates. It is based on the CQS principle, although it is more precise and could be considered a pattern based on messages, commands and events as opposed to CQS that simply suggests to separate queries from commands and the Domain Model. CQRS is usually related to more advanced scenarios like having a different physical database for the Reads/Queries than for the Writes/Updates. Even going further, a more evolved CQRS system would be when implementing [Event-Sourcing \(ES\)](#) for your Updates/Writes database, so you would only store events in the Domain Model instead of the "current state data".

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.

The present microservice's design of this guide is based on CQS and 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 (which is also a possible approach in CQRS). Since it is the simplest approach, this document refers to it as CQS, although it could also be said that it is following the CQRS pattern in its simplest form.

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 just querying. On the other hand, commands (which will trigger transactions and data updates) are what impact your system, so this area related to commands or updates is where you need to be careful when dealing with a complexity and "ever-changing business rules" and therefore this is the area where you might want to apply Domain-Driven Design patterns to have a more solid and better modelled system.

However, as introduced in the following sections, Domain-Driven Design presents many restrictions and constraints based on patterns like Aggregates, Domain Entities, Repositories, etc. Those patterns are very beneficial for your system so you can evolve your it in the long term with quality, but honestly, they usually just matter for the transactional/updates area which can be triggered by commands. If that is the case, why should you limit yourself and use the same constraints, limitations and even unnecessary complexity when still using those patterns for the queries if that can turn to worse performance and lack of flexibility in your queries?

For example, when using Aggregates for your model plus Entity Framework Core for your infrastructure, if you use that approach also for your queries you will have constraints derived from the fact that an Aggregate might not have info about other additional entities that you'd like to include in a specific query. That will make your end-to-end query more complex, you might need to aggregate data from multiple Aggregates and do convolute operations that you shouldn't need to do for a query. Not taking into account that when using Entity Framework Core you might not get the

best performance possible for your queries because of many reasons compared to a plain SQL data access like when using a Micro ORM.

This is why, as shown in image X-XX, this guide suggests to implement DDD patterns only to your transactional/updates area of your microservice (triggered by Commands) but when dealing with queries, you can just forget about DDD patterns and design those queries separated from the commands/updates, following a CQS approach, by implementing straight queries using a Micro ORM like [Dapper](#) or any other Micro ORM which offers great flexibility for the queries as you can implement any query based on SQL sentences while getting the best performance because of being a very light framework with very little overhead.

Commented [CDIT3]: Image at the beginning of this section "CQS and DDD based microservice design"

CQS/CQRS and DDD patterns are not top-level architectures

Something really important to highlight is that CQS, CQRS and most DDD patterns (like DDD Layers or a Domain Model with Aggregates) are not architectural styles but only architectural patterns and therefore should not usually be used as top-level architectures.

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/CQS and DDD patterns describe something inside a single system or component, in this case, something inside a microservice.

This is very important to understand. Most architectural patterns like CQRS or most DDD patterns are not good to apply everywhere. If you see architectural patterns applied as a top-level architecture, you probably have a problem. For example, to say "all microservices must use DDD or CQRS" is wrong and bad. It will be a large failure if you try to 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 about. It is its own set of tradeoffs and decisions that have been made per bounded-context, microservice or any boundary you can have per sub-systems. Do not try to apply the same architectural patterns like CQRS or DDD everywhere.

References – CQS and CQRS

CQS vs. CQRS (by Greg Young)

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

CQRS Documents (Greg Young)

https://cqrsrcosmosis.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://cqrsrcosmosis.files.wordpress.com/2010/11/cqrs_faq.pdf

Event-Sourcing (ES)

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

Implementing the Reads/Queries in a CQS/DDD microservice

As a possible implementation example, the "Ordering" microservice from the *eShopOnContainers* reference application has implemented the queries independently from the Domain-Driven Design model and transactional area. The image X-XX shows where the queries are defined in the "Ordering" microservice.

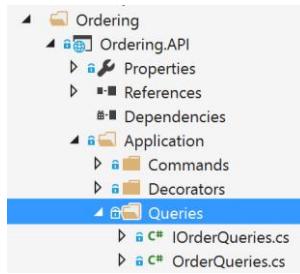


Figure X-XX. Queries in the Ordering microservice from eShopOnContainers

ViewModels especially made for the client apps, independent from the Domain Model constraints

Since the queries are performed to obtain the data needed by the client applications, the model to return to the client apps can be especially made for them and based on the data returned by the queries. Because of that, this especial model or DTOs can be called ViewModels, as they are the data models needed by the views from the client apps.

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 independently from the Domain Model, the Aggregates boundaries and constraints would be 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 could be pre-defined in classes, or could also be created dynamically based on the queries performed, which is very agile for developers.

Dapper: Selected Micro ORM as mechanism to query in eShopOnContainers Ordering microservice

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

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

Dapper is an open source project (original created by Sam Saffron) and part of the building blocks used in Stackoverflow.

When using Dapper you can just write any SQL query which could be accessing and joining multiple tables.

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



Then you need to place 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 "System.Data.SqlClient" and through the "QueryAsync<>()" 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 would be dynamically added to the returned ViewModel.

```
using Dapper;
using Microsoft.Extensions.Configuration;
using System.Data.SqlClient;
using System.Threading.Tasks;
using System.Dynamic;
using System.Collections.Generic;

public class OrderQueries : IOrderQueries
{
    public async Task<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 a very straightforward code and very productive. However, you could also pre-define ViewModels (like pre-defined DTOs) if you would like to have the 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

Note that given that the selected approach for a sample microservice is CQS/CQRS, the DDD implementation will be only related to the transactional/updates area of that 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 steps/areas of problems as bounded contexts (each bounded context correlates to a microservice), and emphasizes a common language to talk about these problems, plus it 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. The design and implementation of those internal patterns is precisely what this section is introducing.

It is important to highlight that sometimes these DDD technical rules and patterns are perceived as hard barriers implementing the DDD, but at 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". As described previously, if your microservice is simple, like a CRUD service, using DDD internal patterns doesn't make sense and it would be better if you just implement a simple CRUD service with straightforward code like writing directly Entity Framework Core code in an ASP.NET Core project.

When designing and defining a microservice, where do you draw the boundaries? Domain Driven Design community helps you deal with this 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. These components within those boundaries end up being your microservices. Microservices is about boundaries and so is DDD.

Keep the microservice's context boundaries relatively small

In regards business functionality to be implemented in a DDD microservice, any microservice should be reasonably small when implementing a specific Bounded-Context. Do not try to implement the whole application or the whole "Core-Domain" within a single DDD microservice or it won't really be a microservice oriented application. Try to design a microservice's size as small as possible as long as it makes sense. On the other hand, if you realize that your microservices are having a too chatty communication, that might be a symptom of a "too small" microservices design.

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 of the emergent complexity. Distinct layers imply that translation must happen between some of the layers in order 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 via perhaps a REST Web API service. The entity is contained within the domain layer and should not be dragged 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 image xx-xx shows how that design is implemented in the *eShopOnContainers* app.

Layers in a Domain-Driven Design Microservice

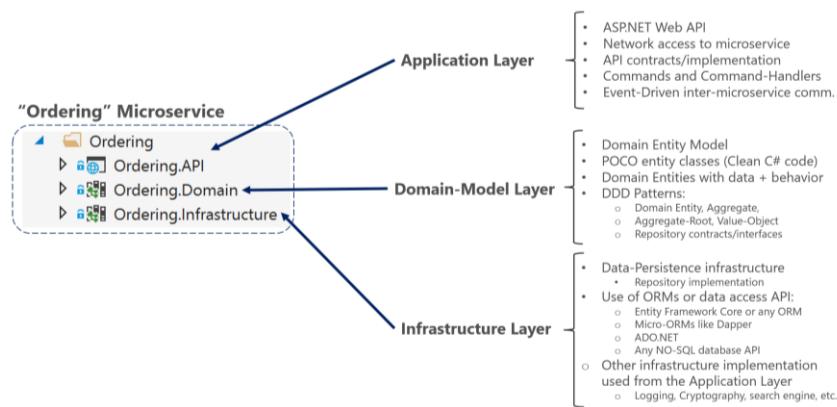


Figure X-XX. DDD Layers in the Ordering microservice from eShopOnContainers

A layer is simply a set of classes that you could group in a project folder or you can also put each layer in a different class library. A layer is something logical, a group of classes, you don't need to implement it as a class library if you don't want to. However, implementing each major layer as a library provides a better control of dependencies between each layer. 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 the screenshot below about the Ordering.Domain layer library which only has dependencies with the .NET Core libraries.

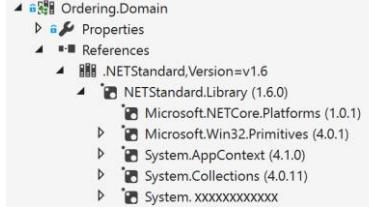


Figure X-XX. Layers implemented as libraries allow a better control of dependencies

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 the infrastructure level.

Luckily, nowadays most modern ORM (Object-Relational Mapping) frameworks allow this approach so your domain model classes are not coupled to the infrastructure (like Entity Framework Core). However, having POCO entities is not always possible when using certain NO-SQL persistence and frameworks like Actors and Reliable Collections in Azure Service Fabric, but it is a "good to have" goal and certainly possible if using relational databases and Entity Framework Core.

You could also, of course, implement data access without any ORM, as well, but that can require more custom code and a larger effort.

"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."

When implementing a microservice's "Application Layer" in .NET, that "layer" would be coded as an application project that varies depending on what you are building. For instance, a common application layer project type can be an ASP.NET 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, Queries if using a CQS approach (explained in the previous section) Commands accepted by the microservice and even the event-driven communication between microservices. However, the ASP.NET Web API must not contain business rules or domain knowledge (especially domain rules in regards transactions or updates) which should be owned by the Domain Model class library.

The Application Layer (in this case an ASP.NET Web API project) must only coordinate tasks and must not hold/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 must not depend on any infrastructure framework.

Infrastructure Layer: How that data initially held in domain entities in-memory will be persisted in databases or any other persistent store is a different matter and it will be implemented in the “Infrastructure Layer” like when using Entity Framework Core implementing the Repository pattern classes which use DbContext to persist data in a relational database. There’s a section within this section named “Infrastructure Layer” that explains the data persistence.

Because of the 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 you might be using to persist data (EF or any other framework) by not taking hard dependencies on frameworks. Your Domain-Model layer class library should have just “your domain code”, just [POCO](#) entity classes implementing the heart of your software completely decoupled from invasive infrastructure technologies.

As a result, your layers or class libraries and projects should ultimately depend on your Domain Model layer/library, not vice versa, as shown in the figure X-XX.

Dependencies between Layers in a Domain-Driven Design service

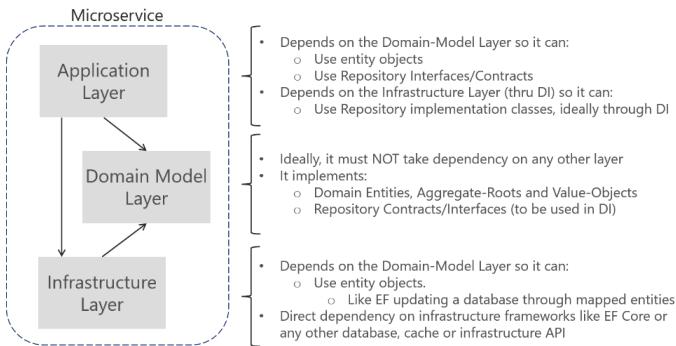


Figure X-XX. Dependencies between Layers in DDD

That layer’s design would be independent per microservice, and as mentioned previously, you could implement your most complex microservices following DDD patterns while implementing in a much more simpler way (simple CRUD in a single layer) your 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

In a similar way than in DDD each Bounded-Context has to have its own Domain Model, each microservice has to have and own its model, as introduced previously in this guide.

However, a Domain Model as defined in DDD, it is not just a data-model but a model that captures more than data entities. It also captures entity's rules, behavior, business language and constraints of a specific domain's problem (Bounded-Context). That special "Rich Domain Model" is what you try to model and implement by following Domain-Driven Design patterns.

The Domain Entity pattern

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

According to Eric Evans' definition, "*An object primarily defined by its identity is called 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 might be implemented as a different group of attributes depending on each microservice's context and domain model. For instance, the Customer entity might have most of the person's attributes in the Profile or Membership microservice. However, the Buyer entity (which shares the identity with the Customer entity) in the Ordering microservice might have less attributes because you just care about certain user's data related to the order process. The context of each microservice impacts on 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 (object accessed in memory). For example, as part of an "Order" entity class we must have business logic like and operations like "add order item to order", or data validation, total calculation, etc. implemented as methods within the same entity class.

The image X-XX shows a diagram of a Domain Entity which clearly implements not only data attributes but also operations/methods with related domain logic.

Domain Entity pattern

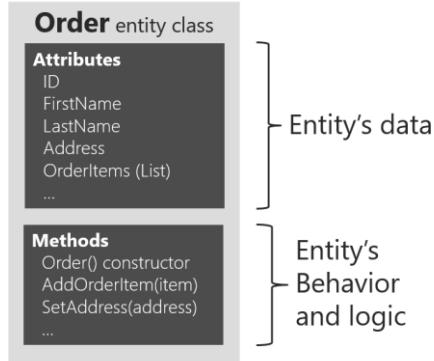


Figure X-XX. Example of Domain Entity Design implementing data plus behavior

Of course, we could also have entities that do not implement any logic as part of the entity class, but this case should only happen if that entity really doesn't have related 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

An Anemic Domain Model is basically a data model implemented as a bunch of classes with attributes or properties, just that. There are "entity" objects most of them based on the nouns in the domain space, and these objects are connected with 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 on these objects, making them little more than a DTO or data class with getters and setters. Of course, these data models will be used from a set of service objects (in many cases named "Business Layer") which capture all the domain or business logic. These services (Business Layer) sits on top of the data-model and use that data-model just for data.

The anemic domain model is really just a procedural style design. Those "anemic entity objects" are not real objects because they lack of behavior (methods). They just 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 the domain model can bring.

Don't make mistake about it, if your microservice (or Bounded-Context) is very simple, data-driven or CRUD, that anemic domain model (entity objects with just data properties) might be "good enough" and it might not be worth to be implemented using more complex DDD patterns.

Some people may say that the Anemic Domain Model is an anti-pattern. Well, 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 and 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 such an identity like 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 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 like groups of attributes.

What may be an Entity in a microservice may not be in another microservice because in that second 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 addresses could be very important for the business domain and therefore the address must have an identity because the billing system can be directly linked to the addresses. In this last 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 you need to have. A classic example is an order that also contains 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 root-entity, usually called Aggregate Root.

Identifying Aggregates can be pretty 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, then you can identify the aggregates in your model. Thinking about transaction operations is probably the best way to identify aggregates.

Aggregate-Root or Root-Entity Pattern

An aggregate will be composed at least by one entity, the Aggregate Root (AR), also called root-entity or primary entity. Additionally, it can have multiple child entities and some Value-Objects, as well, all those 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/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 within your aggregate. If you change a child entity or VO independently, the Aggregate Root cannot ensure the aggregate is in a valid state. It would be like a table with a loose leg. Maintaining the consistency is the main purpose of the Aggregate Root.

In figure X-XX, you can see represented sample aggregates, like the "Buyer" aggregate which is made of a single entity (the Aggregate Root "Buyer") compared to the "Order" aggregate which is made of multiple entities and a Value-Object.

Aggregate pattern

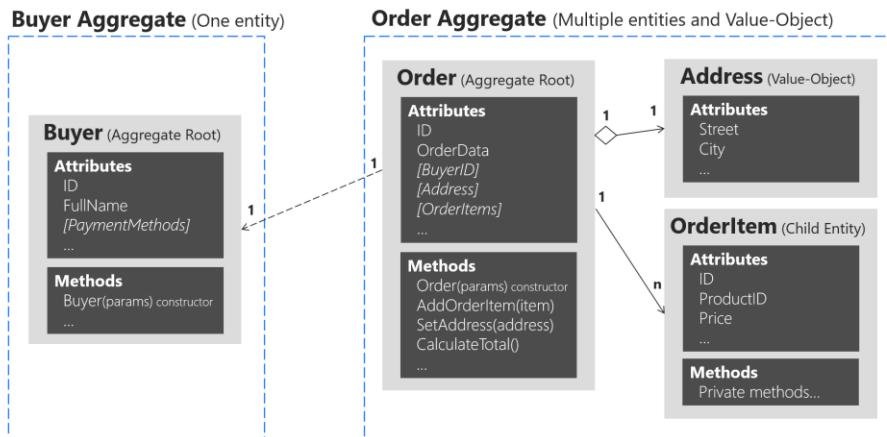


Figure X-XX. Aggregate pattern examples

Note that the "Buyer" aggregate could have additional child entities depending on your Domain, as it has in fact in the sample Ordering microservice from *eShopOnContainers* reference application. The

figure X-XX 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 there 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://devg.com/aggregate-pattern/>

Effective Aggregate Design - Part I: Modeling a Single Aggregate

https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGRAGATES_PART_1.pdf

Effective Aggregate Design - Part II: Making Aggregates Work Together

https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGRAGATES_PART_2.pdf

Effective Aggregate Design - Part III: Gaining Insight Through Discovery

https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGRAGATES_PART_3.pdf

DDD Tactical Design Patterns

<https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part>

Implementing a microservice's Domain Model with .NET Core and Entity Framework Core

In the previous section, it was explained the fundamental design principles and patterns to design a domain model. Now, it is time to drill down and show you 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/references to EF Core in your Domain Model).

Domain Model structure in a .NET Core Standard Library

The way you structure your model within certain folders is completely up to you. The way it is implemented in the Ordering microservice from the *eShopOnContainers* application is designed to try to show you DDD model concepts in a clear way. But you are free to group your classes (Aggregate-Roots, Entities, Value-Objects and Repository Interfaces) in a different way, of course.

As you can see in the screenshot from image X-XX, in the Ordering Domain-Model there are two identified Aggregates, the Order aggregate and the Buyer aggregate. Each aggregate is usually a group of domain entities and value-objects, although you could have an aggregate composed by a single domain entity (the Aggregate-Root or Root Entity), as well.

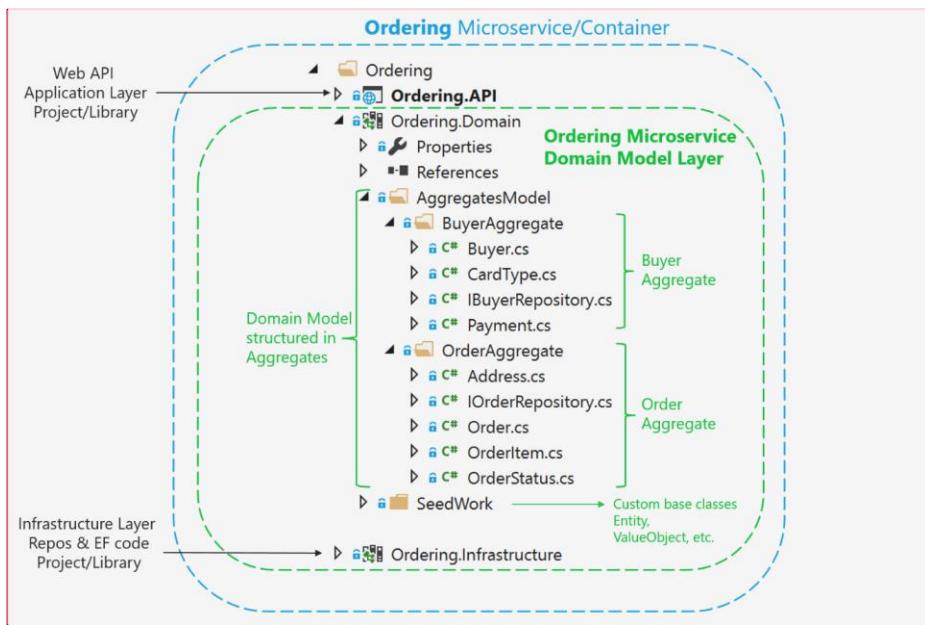


Figure X-XX. Domain Model structure for the Ordering microservice

Additionally, in the Domain-Model layer you usually place the Repository contracts/interfaces that are just the "infrastructure requirements" of your model, but not the infrastructure implementation of those repositories that 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 additional Value-Objects, if any.

Transactional consistency simply means that whatever is comprised within 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 by the following elements shown in the figure X-XX extracted from the eShopOnContainers Ordering microservice domain model.

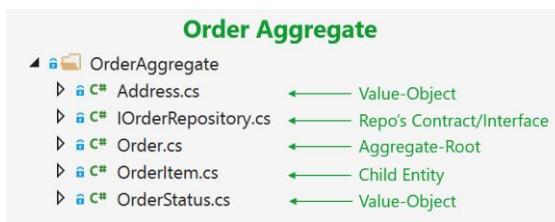


Figure X-XX. The "Order" aggregate in the VS solution

In order to check out what kind of entity or object is each class within an aggregate, you need to open its code and see how it is marked with your custom base classes or Interfaces implemented in the folder "Seedwork".

Implementing Domain Entities as a POCO classes

As introduced in the previous design section, the way you implement a domain model in .NET is simply by creating POCO classes that implement your domain entities, as in the following code where you can see that the Order class is marked as an entity but also an aggregate root. Because of the Order class is deriving from the custom base "Entity" class, it is able to re-use common code related to entities. Keep in mind that those base classes and interfaces are custom, so it is "your code", not infrastructure code from any ORM like EF.

Entity Framework Core 1.0

```
public class Order : Entity, IAggregateRoot //Entity is a custom base class with the Id
{
    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 this first 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 "marked" with an interface named IAggregateRoot. That interface is an empty 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 (like AddOrderItem() when adding an OrderItem to the Aggregate). You should not create/update OrderItems independently or directly, the AggregateRoot class must keep the control and consistency of any update operation against its child entities.

For instance, you shouldn't do the following from any CommandHandler method or application class:

Wrong according to DDD

```

//My code in CommandHandlers or Web API controllers
//... Code with validations and business logic ...
OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName, pictureUrl,
                                         unitPrice, discount, units);

myOrder.OrderItems.Add(myNewOrderItem);

//...

```

In this "bad case", the Add() operation is a pure "adding data" operation 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.

For doing that approach, you would have needed to mark the OrderItems collection with a "public setter" in its property definition. That is a "No do" in DDD. Entities must not have public setters in any entity's property.

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 that will keep consistency in a more controlled and object-oriented way instead of doing a "transactional script" code implementation.

In the following code snippet, you see how you would operate when adding an OrderItem to the Order aggregate.

Right according to DDD

```
//My code in CommandHandlers or WebAPI controllers, only related to application stuff  
// NO code here related to OrderItem validations/logic  
myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units);  
// The code related to OrderItem params validations or domain rules will be within AddOrderItem()  
//...
```

The important point here is that most of the validations/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, plus, if there are different amounts of discounts but the product Id is the same, you should apply the higher discount, or any other domain logic to be applied.

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/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. This is extremely useful when properties only must have a get accessor. Previously, with properties get and set accessors were required and the only choice was to put the setters as private.

Now, you can use simple fields instead of properties and implement any update to the field through methods and read access through public getter properties.

Because in DDD you want to update the entity only through methods in the entity (or the constructor) so you can control any invariant and consistency of the data, just properties with only get accessor are defined. The properties are backed by private fields. Private members can only be accessed from within the class. Of course, there's an exception: EF Core needs to set these fields as well.

Entity Framework Core 1.1 or later

```
public class Order : Entity, IAggregateRoot //Entity is a custom base class with the Id
{
    private bool _someOrderInternalState;

    private int _buyerId;
    public int BuyerId => _buyerId;

    private DateTime _orderDate;
    public DateTime OrderDate => _orderDate;

    private int _statusId;
    public int StatusId => _statusId;

    private ICollection<OrderItem> _orderItems;
    public ICollection<OrderItem> OrderItems => _orderItems;

    private Address _shippingAddress;
    public Address ShippingAddress => _shippingAddress;

    private int _paymentId;
    public int PaymentId => _paymentId;

    protected Order() { }

    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
     */
}
```

Mapping properties with only get accessors to the field in the EF Core Context

With the context you need to map the properties with only get accessors to the field. This is done with the HasField method of the PropertyBuilder. This is explained in the Infrastructure Layer section of this guide.

Commented [CDIT5]: Make sure we explain this in the infrastructure section

Mapping Fields without Properties

With this new feature in EF Core 1.1 to map columns to fields, it is also possible to not use any properties, 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 has no property related, neither for setter or getter. That field will be calculated within the order's business logic and used from the order's methods, too. 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 related property to a column in the database. **This is also explained in the Infrastructure Layer section of this guide.**

Commented [CDIT6]: Make sure we explain this in the infrastructure section

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 is called “Seedwork” instead of framework because it is just a small subset of reusable classes, but it cannot be considered as a framework. [Seedwork](#) is a term introduced by Martin Fowler, but you could also name that folder as “Common” or any other similar concept.

Below you can see 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.

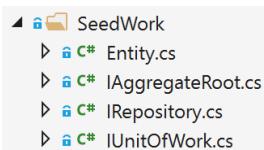


Figure X-XX. A sample Domain Model “Seedwork” with base classes and

This is the kind of copy and paste reuse that many developers use between projects, not a formal framework. You can have Seedworks wthin any layer or library, however, when it gets big enough, you might create a single class library just for itself.

The custom “Entity” base class

For instance, 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, like the entity Id, [equality operators](#), etc.:

```
Entity Framework Core 1.0
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;
        else
            return item.Id == this.Id;
    }

    public override int GetHashCode()
    {
        if (!IsTransient())
        {
            if (!_requestedHashCode.HasValue)
                _requestedHashCode = this.Id.GetHashCode() ^ 31; // XOR for random distribution
            (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);
    }
}
```

Commented [CDIT7]: This code will change when migrated to EF 1.1. So it'll be using a field with just a getter, etc.

```
public static bool operator !=(Entity left, Entity right)
{
    return !(left == right);
}
```

Repository contracts/interfaces in the Domain Model Layer

The Repository contracts are simply .NET interfaces and express the contract requirements of the Repositories to be used per each Aggregate. But, the Repositories themselves with EF Core code or any other infrastructure dependencies and code must not be implemented within the Domain Model, only the contacts 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*". Doing that way, from the application layer (in this case the Web API project of the microservice) when using Dependency Injection you will have a dependency to the "requirements" defined in the Domain Model but not a direct dependency to the infrastructure/persistence layer which is where you are implementing the actual Repositories.

For example, the following code snippet with the `IOrderRepository` interface defines what operations need to have the `OrderRepository` to be implemented in the infrastructure layer library, which in its current implementation of the application it just needs to add/store the order to the database since queries are split following the CQS approach and updates to Orders are not implemented in this particular implementation.

```
public interface IOrderRepository : IRepository
{
    Order Add(Order order);
}
```

References – Repository Contracts

Separated Interface pattern (By Martin Fowler)

<http://www.martinfowler.com/eaaCatalog/separatedInterface.html>

Implementing Value-Objects

TBD – To be written when the code is implemented/updated in eShopOncontainers

Commented [CDIT8]: TBD – To be written when the code is implemented/updated in eShopOncontainers

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 be "always-valid entities". There is a certain number of invariants for an object that should always be true (as an example that an `OrderItem` object always has to have a quantity or 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 even be able to exist without being valid. Invariant rules are simply expressed as contracts and exceptions or notifications are thrown when these are violated.

The reasoning behind this is that a lot of bugs comes from the fact that objects are in a state they should never have been. This 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 and within methods that can update the entity. There are multiple ways to implement those variations like sentences of control verifying data and raising exceptions if the validation cannot succeed or more advanced patterns like using the Specification pattern for validations and the Notification pattern to be able to return a collection of errors instead of returning an exception as soon as something was not validated.

Validating conditions and returning exceptions

The code example below shows the simplest approach in a Domain Entity by raising an exception, but in the references table at the end of this section you can see more advanced implementations based on the mentioned patterns.

```
public void SetAddress(Address address)
{
    if (address == null)
    {
        throw new ArgumentNullException(nameof(address));
    }
    ShippingAddress = address;
}
```

A similar approach raising an exception could be done in the entity's constructor so it makes sure that the entity is valid when you create it.

Using Validation attributes in the model based on Data Annotations

Another approach is to use validation attributes (Data Annotations). Validation attributes are a way to configure model validation so it's 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, phone number, or email address. Validation attributes make enforcing these requirements very easy to use.

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 have to 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. So, use it or not depending on how coupled you'd like your model to be with that infrastructure:

```
using System.ComponentModel.DataAnnotations;
//Other usings
```

Commented [CDIT9]: Is it possible to use Validation Attributes from Data Annotations without using the ModelState.IsValid() from the MVC controllers? Could it be invoked from the entity itself to check if it is valid at the end of a constructor, for instance?...

```

public class Product : Entity //Entity is a custom base class which has the Id
{
    [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's logic and constructor...
}

```

However, from a DDD point of view, the domain model is best kept lean with use of your plain-old exceptions in your entity's behavior methods or by implementing the Specification and Notification patterns to enforce validation rules. But validation frameworks like Data Annotations in ASP.NET Core or any other validation frameworks like FluentValidation carry a requirement to invoke the application framework like when calling the "ModelState.IsValid()" you need to invoke in ASP.NET controllers when using Data Annotations.

Validating Entities by implementing the Specification pattern and the Notification pattern

Finally, more elaborated approaches to implement validations at the domain model is by implementing the **Specification pattern** in conjunction with the **Notification pattern**, as explained in some of the references below.

It is worth to mention that you can also use only one of those patterns, like 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 mentioned Specification pattern and the Deferred Validation approach described by Ward Cunningham in his Checks pattern language. If you have the Implementing Domain-Driven Design book by Vaughn Vernon, you can also read from pages 208-215.

Commented [CDIT10]: Change / Extend

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 have to have validation at the Domain Model level, validation can be both at 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 be a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

Also, just like the view model and the domain model are different, view model validation and domain validation might be similar but serve a different purpose. And if you’re still concerned about being DRY (“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 more than to follow the DRY principle everywhere.

You could also validate your commands or input DTOs in the server side, especially if your system doesn’t have any client UI application, like if you are building just a public API. But if you have any client application, from a UX perspective, it is good to be proactive and not allow the user to type in stuff that makes no sense.

Therefore, in the client side you will be usually validating the ViewModels being used in the client app or you could also validate the client output DTOs or commands (if you choose to create commands in the client side) to be sent to the server side before you send it to the services.

The implementation of client side validation depends on what kind of client application you are building, as 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 there are a few references of validations depending on each type 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>

As a summary in regards validations, here are the most important topics:

Entities and Aggregates should enforce their own consistency and be "always-valid". Aggregate-Roots are in fact responsible for multi-entity consistency within the same aggregate. What is the purpose of an aggregate if not to enforce its own consistency?

If you think that an entity needs to enter into an "invalid state" consider that you could probably use a different object model, like a temporal DTO until you create the final domain entity.

Validation frameworks are best used at specific layers like in the presentation layer or application/service layer but probably not into the Domain Layer as you need to take a strong dependency with 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

TBD - REFERENCES - <http://www.tonytruong.net/domain-events-pattern-example/>

Domain Events – Take 2: <http://udidahan.com/2008/08/25/domain-events-take-2/>

Commented [CDIT11]: TBD, might not include DOMAIN EVENTS in this eBook, as that is a pretty large subject..
However, APPLICATION OR INTEGRATION EVENTS (to communicate microservices with each other) will be covered, though.

Designing the Infrastructure-Persistence Layer

The Repository pattern

Repositories are classes/components that encapsulate the logic required to access data sources. Therefore, they centralize common data access functionality so the application can have a better maintainability and decouple the infrastructure or technology used to access databases from the Domain layer. If we use an ORM like Entity Framework, the code to be implemented is highly simplified thanks to Linq and strongly types, so we can focus on the data persistence logic rather than on data access plumbing like when using plain ADO.NET.

"Repository" 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".

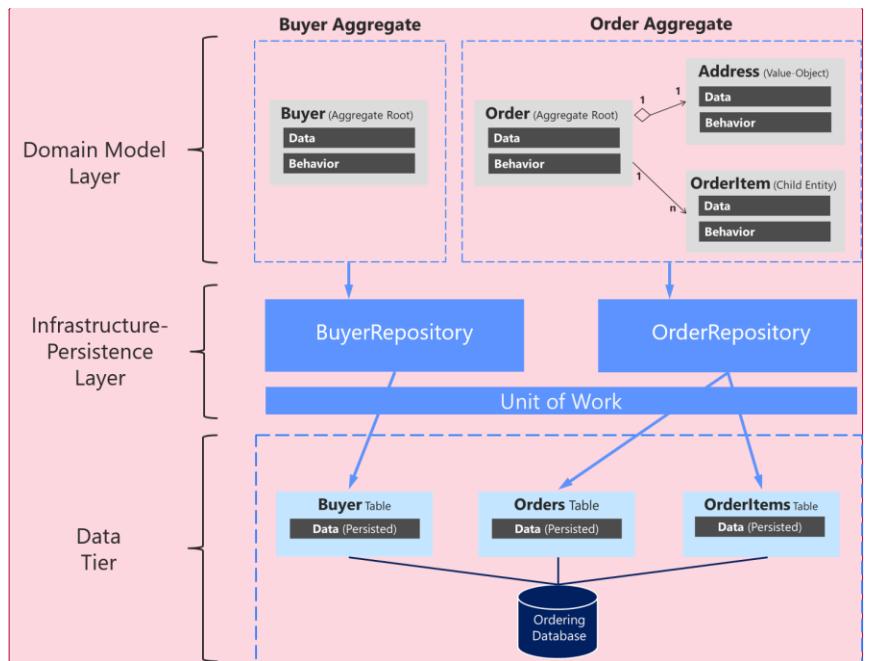
Define one Repository per Aggregate

Hence, per each aggregate (or per each Aggregate-Root as they are 1:1) 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 and also allows you to persist updated data in those 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 be just to create/update/delete the database from your Domain Model, but you won't need to have any method for queries in such a Repository.

It is important to re-emphasize that Repositories should only be defined per 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 one Repository per each table in the database but per each aggregate-root.

In a microservice based on DDD, the only channel you should use to update the database should be through the Repositories because they have a 1:1 relationship with the Aggregate-Root which is who controls the aggregate's invariants and transactional consistency. It is okay to query the database through other channels (like you can do following a CQS/CQRS approach) because queries are idempotent and no matter how many queries you do, the database won't change. But, the transactional area, the updates, must always be controlled by the Repositories and the Aggregate-Roots.



Commented [CDIT12]: TO DO: Create a simplified newer image with no attributes, but showing the relationship between:
 Aggregates → Repos → Database Tables
 Aggregates above
 Repositories in the middle
 Database tables below

Figure X-XX. Relationship between Repositories, Aggregates and Database Tables

The Repository pattern makes it easier to test your application logic

The Repositories allow you to easily make your application "testable" with unit tests.

As introduced in a previous section, it is recommended to define and place the Repository interfaces in the domain layer so the application layer (like 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 needing any connectivity to the database.

Connection to databases can fail and most of all, running hundreds of tests against a database is a bad thing because 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 Tests but an Integration Test. You should have many Unit Tests running fast but fewer Integration Tests against the databases.

Difference between the Repository pattern and the legacy Data Access class (DAL class)

It is important to differentiate between a Repository and the legacy "Data Access" object (aka. DAL). A Data Access object directly performs data access and persistence operations against the storage. However, a repository "marks" the data with the operations you want to do in the memory of a Unit of Work object (like 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 operations of insert/update/delete. To say it in simple words, it means that for a specific user action (say registration on a website), all the transactions like insert/update/delete are done in one single transaction, rather than doing multiple database transactions in a more chatty way.

These multiple persistence operations will be performed at a later time in a single action when your code from the Application layer commands it. That decision about "Applying changes" in memory into the real database storage is usually based on the Unit of Work pattern (In Entity Framework 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 which will be more efficiently run in comparison to many isolated operations against the database. Therefore, the selected ORM will be able to optimize the execution against the database (e.g., 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://devg.com/repository-pattern/>

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

What does Entity Framework Core provide?

EF Core offers the possibility of implementing your Domain Entities as POCO Entities with some new possibilities since EF Core 1.1 like plain fields instead of properties.

TBD – Search EF Core narrative

Implementing 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, like the following class:

TBD

In regards the persistence methods that should be placed in each Repository class, you usually should place methods that are updating an entity contained by its related Aggregate.

Mapping properties with only get accessors to the field in the EF Core Context

With the context you need to map the properties with only get accessors to the field. This is done with the HasField method of the PropertyBuilder. This is explained in the Infrastructure Layer section of this guide.

Commented [CDIT13]: Make sure we explain this in the infrastructure section

Mapping Fields without Properties

With this new feature in EF Core 1.1 to map columns to fields, it is also possible to not use any properties, 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 has no property related, neither for setter or getter. That field will be calculated within the order's business logic and used from the order's methods, too. 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 related property to a column in the database. This is also explained in the Infrastructure Layer section of this guide.

Commented [CDIT14]: Make sure we explain this in the infrastructure section

REFERENCES

Implementing Repositories with Entity Framework Core

Designing the microservice's Application Layer and Web API

Use S.O.L.I.D. principles and Dependency Injection

TBD

Designing Commands and Command-Handlers as the only input gate to your Domain Model

TBD

REFERENCES

Related to Command classes and input DTOs....

At the Boundaries, Applications are Not Object-Oriented (by Mark Seemann)

<http://blog.ploeh.dk/2011/05/31/AttheBoundariesApplicationsareNotObject-Oriented/>

Implementing the microservice's Application Layer and Web API

TBD

Using Dependency Injection and Autofac as IoC container in ASP.NET Core

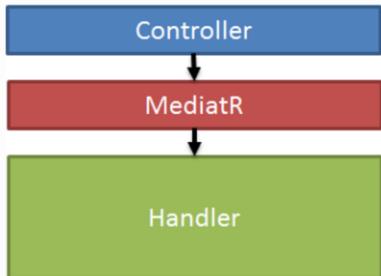
TBD

Implementing Commands and CommandHandlers

TBD

Using MediatR - An in-memory Command Bus with Decorators

TBD



Implementing event based communication between microservices

Defining an Event Bus interface

TBD

Multiple implementations of an Event Bus

TBD

Implementing a simple Event Bus with a SignalR Hub service

TBD

Implementing an Event Bus with Azure Service Bus

TBD

Intro to an Event Bus implementation with NServiceBus

TBD

Intro to Event Bus implementation with RabbitMQ

TBD

(MOVE Multi-Container Application Configuration before Domain-Driven Design sections)

Composing your multi-container application with docker-compose.yml

In the [docker-compose.yml file](#) you can explicitly describe how you would like to deploy your multi-container application. Basically, you can define each of the containers you want to deploy plus certain characteristics for each container deployment but once you have this "multi-container deployment description file" you can deploy the whole solution in a single composed step by using the CLI command "[docker-compose up](#)". Otherwise, you would need to specify when using Docker CLI when deploying container-by-container with "docker run". Therefore, each service defined in docker-compose.yml must specify exactly one of image or build. Other keys are optional, and are analogous to their "docker run" command-line counterparts, as mentioned.

Commented [CDIT15]: MOVE Multi-Container Application Configuration before Domain-Driven Design sections

In this document, 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 a few additional interesting definitions and details that is worth to dig into.

The following yaml code is the definition of a possible global but single docker-compose.yml for the eShopOnContainers solution.

```
version: '2'
services:
  webmvc:
    image: eshop/webmvc
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
    ports:
      - "5100:80"
    depends_on:
      - catalog.api
      - identity.data
      - basket.api
  webspa:
    image: eshop/webspa
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
    ports:
      - "5104:80"
    depends_on:
      - catalog.api
      - identity.data
      - basket.api
  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"
    depends_on:
      - catalog.data
  catalog.data:
    image: microsoft/mssql-server-linux
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"
  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=ordering.data;Database=Microsoft.eShopOnContainers.Services.OrderingDb;User Id=sa;Password=your@password
    ports:
      - "5102:80"
  # (Go to Production): For secured/final deployment, remove Ports mapping and
  # leave just the internal expose section
  #   expose:
  #     - "80"
  #   extra_hosts:
  #     - "CESARDLBOOKVHD:10.0.75.1"
  depends_on:
    - ordering.data
  ordering.data:
    image: eshop/ordering.data.sqlserver.linux
    ports:
      - "5432:1433"
```

```

basket.api:
  image: eshop/basket.api
  environment:
    - ConnectionString=basket.data
  ports:
    - "5103:80"
  depends_on:
    - basket.data
basket.data:
  image: redis

```

First of all, the root key in this file is “services” and under that key you define the multiple services you want to deploy and run when running the “docker-compose up” by using this docker-compose.yml file. In this particular case, this docker-compose.yml file has multiple services defined, as described in the following table.

Service name in docker-compose.yml	Description
webmvc	Container with ASP.NET Core MVC app consuming the microservices from server-side C#
webspa	Container with Web SPA approach app consuming the microservices from remote JavaScript running on browsers
catalog.api	Container with the Catalog ASP.NET Core Web API microservice
catalog.data	Container running SQL Server for Linux, with the Catalog database
ordering.api	Container with the Ordering ASP.NET Core Web API microservice
ordering.data	Container running SQL Server for Linux, with the Ordering database
basket.api	Container with the Basket ASP.NET Core Web API microservice
basket.data	Container running REDIS Cache service, with the Basket database as REDIS cache

A simple Web Service API container

The catalog.api container-microservice has a simple and 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"
  depends_on:
    - catalog.data

```

This containerized service has the following basic configuration in place:

- It is based on the custom “eshop/catalog.api” image. In this particular case, because there is not the “build:” key in the file, the image has to be previously built (with “docker build”) or be available locally by downloading it with “docker pull” from any Docker registry before running the docker-compose up using this docker-compose.yml file.
- Builds from the Dockerfile in the current directory (by convention, as there is no an explicit Docker file key for this service).

- It defines an environment variable named "*ConnectionString*" with the connection string to be used by Entity Framework to access the SQL Server container related to the Catalog data model. Note that the SQL server name is "*catalog.data*" which is the same name/id used for the container that is running the SQL Server for Linux with the Catalog database. This is very convenient as being able to use this name it will internally resolve the network name and address so you don't need to know what is the IP for the data-container running SQL.
Important: Since the connection string is defined by an environment variable, you could set that variable through a different mechanism and at a different time, like setting a different value when deploying to production in the final hosts or by doing it from your CI/CD pipelines in VSTS or your chosen DevOps system.
- It exposes the port 80 for internal access within the Docker host (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, too).
- Forwards the exposed port 80 on the container to port 5101 on the Docker host machine (The mentioned Linux VM).
- Links the web service to the Catalog.data service which is a SQL Server for Linux running on a container. This is useful as by specifying this dependency, the Catalog.API container won't start until the Catalog.Data container is already started, as we need to have the SQL database up and running in the first place.

There are, however a few other interesting possible configuration settings at the docker-compose.yml level worth to be mentioned.

[Additional settings from docker-compose.yml](#)

TBD-CDLTLL

```
expose:
  - "80"
ports:
  - "5101:80"
depends_on:
  - catalog.data

extra_hosts:
  - "CESARDLBOOKVHD:10.0.75.1"
```

A database server running as a container

[SQL Server running as a container with a microservice-related database](#)

As mentioned, the related catalog.data container would run SQL Server for Linux with the Catalog database. That is configured with the following yaml code at your docker-compose.yml file and executed when running with "docker-compose up" which will use it.

<pre>catalog.data: image: microsoft/mssql-server-linux environment:</pre>

```
- SA_PASSWORD=your@password  
- ACCEPT_EULA=Y  
ports:  
- "5434:1433"
```

A similar command could be run directly with “docker run”.

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

However, if deploying a multi-container application like eShopOnContainers, using “docker-compose up” is a much more convenient method.

When starting this container for the first time, it will initialize SQL Server with the SA password that you are providing. At this time and once you have SQL Server running as a container, you can update new data into the database by connecting through any regular SQL connection, either from SQL Server Management studio, Visual Studio or from C# code.

The eShopOnContainers application is initializing the database with sample data by seeding with Test Data on the first Startup, as explained in the following section.

Having SQL Server running as a container is not just useful for a demo where you might don't have a SQL Server ready. It is also great for development and testing environments so you can easily run integration tests starting from a clean SQL Server image and know state in regards data by seeding new sample data.

In order to get further insights about SQL Server for Linux running as a container, check the following references.

References – SQL Server for Linux running on Docker containers

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 the Web API Startup

To add data to the database when the application starts up, you can do so by adding some code to the Configure() method at the Startup.cs class from 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...
}

}
```

Then, in our custom CatalogContextSeed it is where data gets populated from code.

```
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 running integration Tests, having a similar way to generate data consistent with your integration tests is something very useful, but being able to create everything from scratch, including a SQL Server running on a container is something great for test environments.

EF Core In-Memory-Database vs. SQL Server running as a container

Another good choice when running tests is to use the Entity Framework Core In-Memory-Database provider. You can do so by specifying that configuration at the Startup:ConfigureServices() method in your Web API project.

```

public class Startup
{
    //Other Startup code...
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddSingleton< IConfiguration>(Configuration);

        //DbContext using an In-Memory-Database provider
        services.AddDbContext< CatalogContext >(opt => opt.UseInMemoryDatabase());

        //((Versus commented 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 doesn't hold any constraints that would be specific to any particular DB. For instance, you could add a unique index on a column and write a test against your in-memory DB to check that it does not let you to add a duplicate value, but when using the in-memory-database, you cannot handle that. So, the in-memory-database does not behave 100% the same way than a real SQL Database. It doesn't emulate any DB-specific constraints. However, it's still useful for testing and prototyping scenarios, but if you want to create accurate integration tests being able to take into account the behavior of a specific database implementation, then you would need to use a real database, like SQL Server. For that purpose, running SQL Server as a container is a great choice and more accurate than the in-memory-database provider from EF.

[Redis cache service running in a container](#)

TBD

Testing ASP.NET Core services and web apps

Controllers are a central part of any ASP.NET Core API service and 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, there are several main differentiated types of tests you should have for your microservices. Unit Tests, Integration Tests, Functional Tests (per microservice) and Service Tests.

- *Unit Tests* - Ensure that individual components/classes of the app work as expected.
Assertions test the component API.

- *Integration Tests* - Ensure that component collaborations work as expected against external artifacts like databases. Assertions may test component API, UI, or side-effects (such as database I/O, logging, etc.)
- *Functional Tests (per microservice)* - Ensure that the app works as expected from the user's perspective, like a use-case.
- *Service Tests* – Ensure that end-to-end service tests, including testing multiple services at the same time are tested. For this type of testing you need to prepare the environment first which in this case means to spin up the services/containers (like using "docker-compose up" first).

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 unit testing controller logic, only the contents of a single action is tested, not the behavior of its dependencies or of the framework itself. As you unit test your controller actions, make sure you focus only on its 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. However, unit tests do not detect issues in the interaction between components, which is the purpose of integration testing.

When writing a unit test of a Web API controller, you directly instance the controller class through the "new" C# language keyword, so it will run as fast as possible, like in the following example.

```
public class ApiIdeasControllerTests
{
    [Fact]
    public async Task Create_ReturnsBadRequest_GivenInvalidModel()
    {
        // Arrange & Act
        var mockRepo = new Mock<IBrainstormSessionRepository>();
        var controller = new IdeasController(mockRepo.Object);
        controller.ModelState.AddModelError("error", "some error");

        // Act
        var result = await controller.Create(model: null);

        // Assert
        Assert.IsType<BadRequestObjectResult>(result);
    }
}
```

Implementing Integration and Functional Tests per isolated microservice

As introduced, Integration Tests and Functional Tests have different goals and purposes. However, the way you implement both when testing ASP.NET Core controllers is pretty similar, so below it is only explained how to implement an Integration Tests.

Integration testing ensures that an application's components function correctly when assembled together. 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 in this case you won't use fakes or mock objects but including the infrastructure, like database access or services invocation from the outside.

Integration tests, because they exercise larger segments of code and because they rely on infrastructure elements, tend to be orders of magnitude slower than unit tests. Thus, it's a good idea to limit how many integration tests you write.

ASP.NET Core includes a *test host* 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, serving test requests without the need for a real web host.

As you can see in the following code, when creating integration tests of ASP.NET Core controllers, you would instantiate the controllers through the Test Host so it is comparable to an HTTP request but running 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);
    }
}
```

For additional details on how to create unit tests and integration tests for ASP.NET Core Web API and MVC applications, read the following references.

References – Testing ASP.NET Core Web APIs and MVC Apps

Testing Controllers in ASP.NET Core:

<https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/testing>

Integration Tests in 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>

Implementing Service Tests on a multi-container application

As introduced before, when testing multi-container applications ou need to have running all the microservices/containers within the Docker host (or container cluster). These end-to-end service tests which include multiple operations involving several microservices/containers requires you to spin-up the whole application in the first place deploying it to the Docker host, by running "docker-compose up" (or comparable mechanism to run the whole application if using an orchestrator/cluster). Once

the whole application and all its services are up and running is when you will be able to execute end-to-end integration and functional tests for your multi-container or microservice based application.

There are a few of approaches you can use. In the docker-compose.yml that you would use to deploy the whole application and test afterwards (like one named as docker-compose.ci.build.yml file that you would use in your CI pipeline), at the solution level, you would expand the entrypoint to use "[dotnet test](#)". You could also use another compose file that would run your tests in the same image you are targeting.

By using another compose file for integration tests that includes your microservices, databases on containers that always resets to its original state, website, and test project you could be getting breakpoints and exception breaks throughout if running in Visual Studio, or you could run those integration tests automatically in your CI pipeline in Visual Studio Team Services or any other CI/CD system that supports Docker containers.

TBD – Include more info or URL references or step-by-step walkthroughs about Tests for multi-container/microservices apps

Developing and deploying a .NET Framework application with monolithic deployment on Windows containers

TBD

TBD

Introduction to the Docker application lifecycle

Containers as the foundation for DevOps collaboration

The lifecycle of containerized applications is like a journey which starts with the developer. The developer chooses and begins with containers and Docker because it eliminates frictions in deployments and with IT Operations, which ultimately helps them to be more agile, more productive end-to-end, faster. Then by the very nature of the Containers and Docker technology, developers are able to easily share their software and dependencies with IT Operations and production environments while eliminating the typical "it works on my machine" excuse. Containers solve application conflicts between different environments. Indirectly, Containers and Docker bring developers and IT Ops closer together. It makes easier for them to collaborate effectively. Adopting the container workflow provides many customers with the continuous they've sought, but had to implement complex release build and config as code management systems.

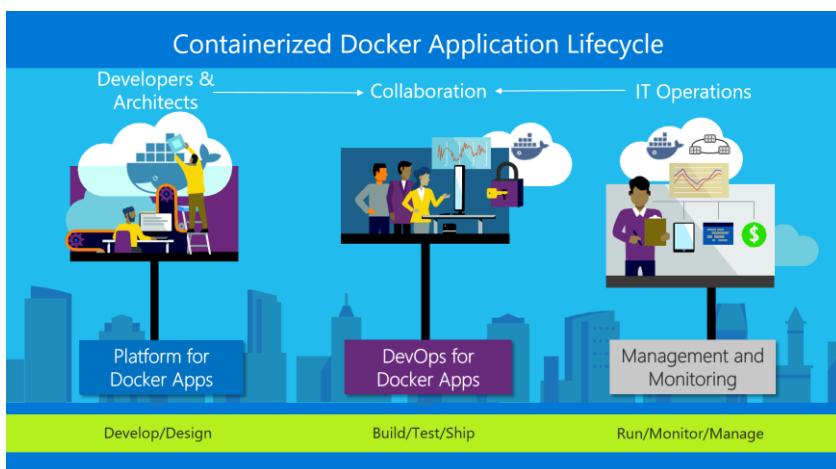


Figure 3-1. Main workloads per “personas” in the lifecycle for containerized Docker applications

With Docker Containers, developers own what's inside the container (application/service and dependencies to frameworks/components) and how the containers/services behave together as an application composed by a collection of services. The interdependencies of the multiple containers are defined with a docker-compose.yml file, or what could be called a deployment manifest. Meanwhile, IT Operation teams (IT Pros and IT management) can focus on the management of production

environments, infrastructure, scalability, monitoring and ultimately making sure the applications are delivering right for the end-users, without having to know the contents of the various containers. Hence the "container" name because of the analogy to shipping containers in real-life. In a similar way than the shipping company gets the contents from a-b without knowing or caring about the contents, in the same way developers own the contents within a container.

Developers on the left of the image 1.1 are writing code and running their code in Docker containers locally using Docker for Windows/Mac. They define their operating environment with a dockerfile that specifies the base OS they run on, and the build steps for building their code into a Docker image. They define how the one or more images will inter-operate using a deployment manifest like a docker-compose.yml file. As they complete their local development, they push their application code plus the Docker configuration files to the code repository of their choice (i.e. Git repos).

The **DevOps** pillar defines the build-CI-pipelines using the dockerfile provided in the code repo. The CI system pulls the base container images from the Docker registries they've configured and builds the Docker images. The images are then validated and pushed to the Docker registry used for the deployments to multiple environments.

Operation teams on the right are managing deployed applications and infrastructure in production while monitoring the environment and applications so they provide feedback and insights to the development team about how the application must be improved. Container apps are typically run in production using Container Orchestrators.

The two teams are collaborating through a foundational platform (Docker containers) that provides a separation of concerns as a contract, while greatly improving the two teams' collaboration in the application lifecycle. The developer owns the container contents, its operating environment and the container interdependencies. While ops takes the built images, the manifest and runs the images in their orchestration system.

Introduction to a generic E2E Docker application lifecycle workflow

From a different perspective, the more detailed workflow for a Docker application lifecycle can be represented as in Figure 3.2. In this case the diagram focuses on specific DevOps activities and assets.

E2E Docker application lifecycle workflow

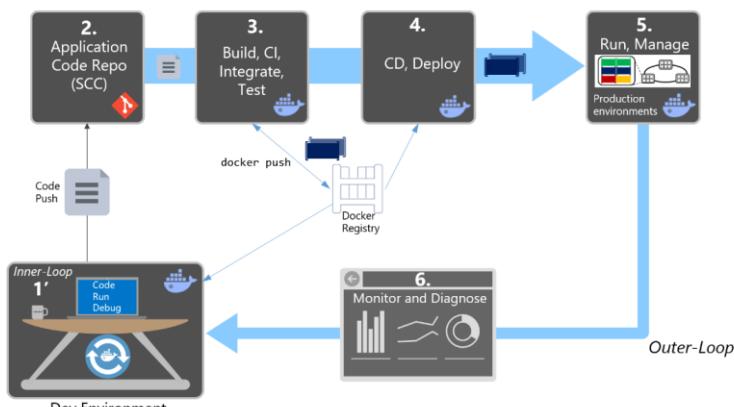


Figure 3-2. High level workflow for the Docker containerized application lifecycle

It all starts from the developer who starts writing code in the inner-loop workflow. Inner-loop defines everything that happens before pushing code into the code-repository (Source Control system, like Git). Once committed, the repo triggers CI (Continues Integration) and the rest of the workflow.

That mentioned initial inner-loop basically consists on typical steps like "Code", "Run", "Test", "Debug", plus additional steps right before "Running" the app locally because the developer wants to run and test the app as a Docker container. That inner-loop workflow will be explained in the following sections.

Taking a step back and looking the E2E workflow, the Development-Operations workflow is more than a technology or a tool set. It's a mindset that requires cultural evolution. It is people, process and the right tools to make your application lifecycle faster and more predictable. Organizations that adopt a containerized workflow typically restructure their orgs to represent people and process that match the containerized workflow.

Practicing DevOps can help teams respond faster together to competitive pressures by replacing error prone manual processes with automation for improved traceability and repeatable workflows. Organizations can also manage environments more efficiently and enable cost savings with a combination of on-premises and cloud resources, as well as tightly integrated tooling.

When implementing your DevOps workflow for Docker applications, you'll see that Docker's technologies are present in almost every stage of the workflow, from your development box while working in the inner-loop (code, run, debug), to the build, test CI phase, and of course at the production/staging environments and when deploying your containers to those environments.

Improvement of quality practices helps to identify defects early in the development cycle, which reduces the cost of fixing them. By including the environment and dependencies in the image, adopting a philosophy of deploying the same image across multiple environments, you adopt a discipline of extracting the environment specific configurations making deployments more reliable.

Rich data obtained through effective instrumentation (Monitoring and Diagnostics) provides insight into performance issues and user behavior to guide future priorities and investments.

DevOps should be considered a journey, not a destination. It should be implemented incrementally through appropriately scoped projects, from which to demonstrate success, learn, and evolve.

Benefits from DevOps for containerized applications

The most important benefits provided by a solid DevOps workflow are:

- Deliver better quality software faster and with better compliance
- Drive continuous improvement and adjustments earlier and more economically
- Increase transparency and collaboration among stakeholders involved in delivering and operating software
- Control costs and utilize provisioned resources more effectively while minimizing security risks
- Plug and play well with many of your existing DevOps investments, including investments in open source

Introduction to the Microsoft platform and tools for containers lifecycle

Vision

Create an adaptable, enterprise-grade, containerized application lifecycle that spans your development, IT operations, and production management.

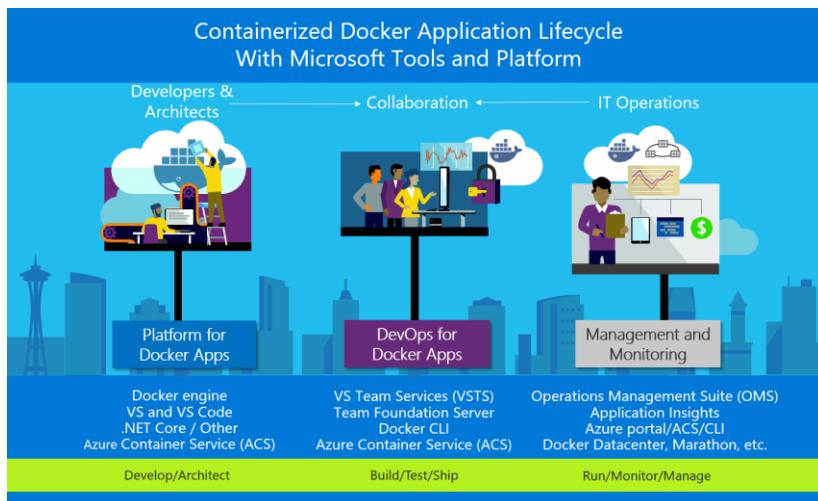


Figure 4-1. Main pillars in lifecycle for Containerized Docker Applications with Microsoft Platform & Tools

Figure 4-1 shows the main pillars in the lifecycle of Docker apps classified by the type of work delivered by multiple teams (app-development, DevOps infrastructure processes and IT Management and Operations). Usually, in the enterprise, the profiles of "the persona" responsible for each area are different. So are their skills.

A containerized Docker lifecycle workflow can be initially prescriptive based on “by default product choices” so it makes it easier for developers to get started faster, but it is fundamental that under the covers there must be an open framework so it will be a flexible workflow capable of adjusting to the different contexts from each organization/enterprise. The workflow infrastructure (components and products) must be flexible enough to cover the environment that each company will have in the future, even being capable of swapping development or DevOps products to others. This flexibility, openness and broad choice of technologies in the platform and infrastructure are precisely the Microsoft priorities for containerized Docker applications, as explained in the following sections.

As shown in figure 4-2, the intention of the Microsoft DevOps for Containerized Docker applications is to provide an open DevOps workflow so you can choose what products to use for each phase (Microsoft or third-party) while providing a simplified workflow which provides “by-default-products” already connected, so you can quickly get started with your enterprise-level DevOps workflow for Docker apps.

	Microsoft technologies	3rd party – Azure pluggable
Platform for Docker Apps	<ul style="list-style-type: none"> ▪ Visual Studio & Visual Studio Code ▪ .NET ▪ Azure Container Service ▪ Azure Service Fabric ▪ Azure Container Registry 	<ul style="list-style-type: none"> ▪ Any code editor (i.e. Sublime, etc.) ▪ Any language (Node, Java, Go, etc.) ▪ Any Orchestrator and Scheduler ▪ Any Docker Registry
DevOps for Docker Apps	<ul style="list-style-type: none"> ▪ Visual Studio Team Services ▪ Team Foundation Server ▪ Azure Container Service ▪ Azure Service Fabric 	<ul style="list-style-type: none"> ▪ GitHub, Git, Subversion, etc. ▪ Jenkins, Chef, Puppet, Velocity, CircleCI, TravisCI, etc. ▪ On-premises Docker Datacenter, Docker Swarm, Mesos DC/OS, Kubernetes, etc.
Management & Monitoring	<ul style="list-style-type: none"> ▪ Operations Management Suite ▪ Application Insights 	<ul style="list-style-type: none"> ▪ Marathon, Chronos, etc.

Figure 4-2. Open DevOps workflow to any technology

The Microsoft platform and tools for containerized Docker applications, as defined in Figure 4-2, has the following components:

- **Platform for Docker Apps development.** The development of a service, or collection of services that make up an “app”. The development platform provides all the work a developer requires prior to pushing their code to a shared code repo. Developing services, deployed as containers, are very similar to the development of the same apps or services without Docker. You continue to use your preferred language (.NET, Node.js, Go, etc.) and preferred editor or IDE like *Visual Studio* or *Visual Studio Code*. However, rather than consider Docker a deployment target, you develop your services in the Docker environment. You build, run, test, debug your code in containers locally, providing the target environment at development time. By providing the target environment locally, Docker containers enable what will drastically help you improve your Development and Operations lifecycle. *Visual Studio* and *Visual Studio Code* have extensions to integrate the container build, run, test your .NET, .NET Core and Node.js applications.
- **DevOps for Docker Apps.** Developers creating Docker applications can leverage *Visual Studio Team Services* (VSTS) or any other third party product like *Jenkins*, to build out a comprehensive automated application lifecycle management (**ALM**).

With VSTS, developers can create container-focused DevOps for a fast, iterative process that covers source-code control from anywhere (VSTS-Git, GitHub, any remote Git repository or Subversion), continuous integration (CI), internal unit tests, inter container/service integration tests, continuous delivery CD, and release management (RM). Developers can also automate their Docker application releases into Azure Container Service, from development to staging and production environments.

- **IT production management and monitoring.**

Management - IT can manage production applications and services in several ways:

- *Azure portal.* If using OSS orchestrators, Azure Container Service (ACS) plus cluster management tools like Docker Datacenter and Mesosphere Marathon help you to set up and maintain your Docker environments. If using Azure Service Fabric, the Service Fabric Explorer tool allows you to visualize and configure your cluster.
- *Docker tools.* You can manage your container applications using familiar tools. There's no need to change your existing Docker management practices to move container workloads to the cloud. Use the application management tools you're already familiar with and connect via the standard API endpoints for the orchestrator of your choice. You can also use other third party tools to manage your Docker applications like Docker Datacenter or even CLI Docker tools.
- *Open source tools.* Because ACS expose the standard API endpoints for the orchestration engine, the most popular tools are compatible with Azure Container Service and, in most cases, will work out of the box—including visualizers, monitoring, command line tools, and even future tools as they become available.

Monitoring - While running production environments, you can monitor every angle with:

- *Operations Management Suite (OMS).* The "OMS Container Solution" can manage and monitor Docker hosts and containers by showing information about where your containers and container hosts are, which containers are running or failed, and Docker daemon and container logs. It also shows performance metrics such as CPU, memory, network and storage for the container and hosts to help you troubleshoot and find noisy neighbor containers.
- *Application Insights.* You can monitor production Docker applications by simply setting up its SDK into your services so you can get telemetry data from the applications.

Docker DevOps lifecycle workflow with Microsoft Tools

Microsoft tools can automate the pipeline for specific implementations of containerized applications (Docker, .NET Core, or any combination with other platforms).

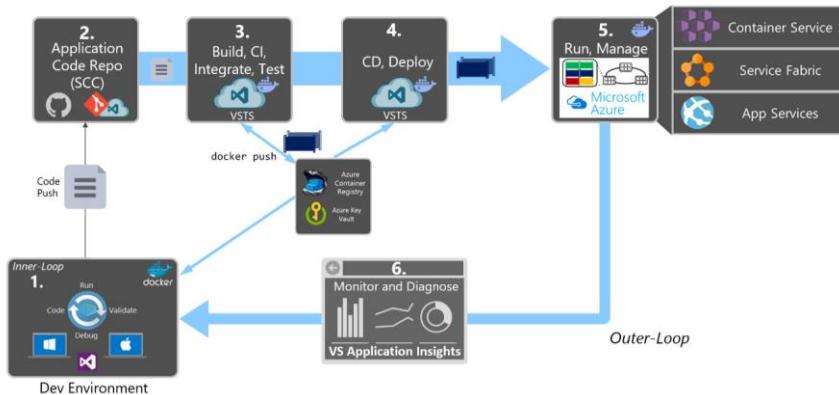


Figure X-X. DevOps outer-loop workflow for Docker applications with Microsoft Tools

You can implement from global builds and Continuous Integration (CI) and tests with VSTS/TFS, to Continuous Deployment (CD) to Docker environments (Dev/Staging/Production), and to provide analytics information about the services back to the development team through *Application Insights*. Every code commit can trigger a build (CI) and automatically deploy the services to specific containerized environments (CD).

Microsoft therefore offers a complete foundation for an end-to-end Containerized Docker application lifecycle. However, it is a **collection of products and technologies which allow you to optionally select and integrate with existing tools and processes**. The flexibility in a broad approach and the strength in the depth of capabilities place Microsoft in a strong position for containerized Docker application development.

For further information about the end-to-end DevOps lifecycle, download the following related eBook.

eBook: Containerized Docker Application Lifecycle with Microsoft Platform and Tools

<http://aka.ms/dockerlifecyclebook>

Conclusions

Key takeaways

- Container based solutions provide important benefits of cost savings because containers are a solution to deployment problems caused by the lack of dependencies in production environments, therefore, improving DevOps and production operations significantly.
- Docker is becoming the “de facto” standard in the container industry, supported by the most significant vendors in the Linux and Windows ecosystems, including Microsoft. In the future Docker will be ubiquitous in any datacenter in the cloud or on-premises.
- A Docker container is becoming the standard unit of deployment for any server-based application or service.
- Docker orchestrators like the ones provided in Azure Container Service (Mesos DC/OS, Docker Swarm, Kubernetes) and Azure Service Fabric are fundamental and indispensable for any microservice-based or multi-container application with significant complexity and scalability needs.
- An end-to-end DevOps environment supporting CI/CD connecting to the production Docker environments provides agility and ultimately improves the time to market of your applications.
- Visual Studio Team Services greatly simplifies your DevOps environment targeting Docker environments from your Continuous Deployment (CD) pipelines, including simple Docker environments or more advanced microservice and container orchestrators based on Azure.