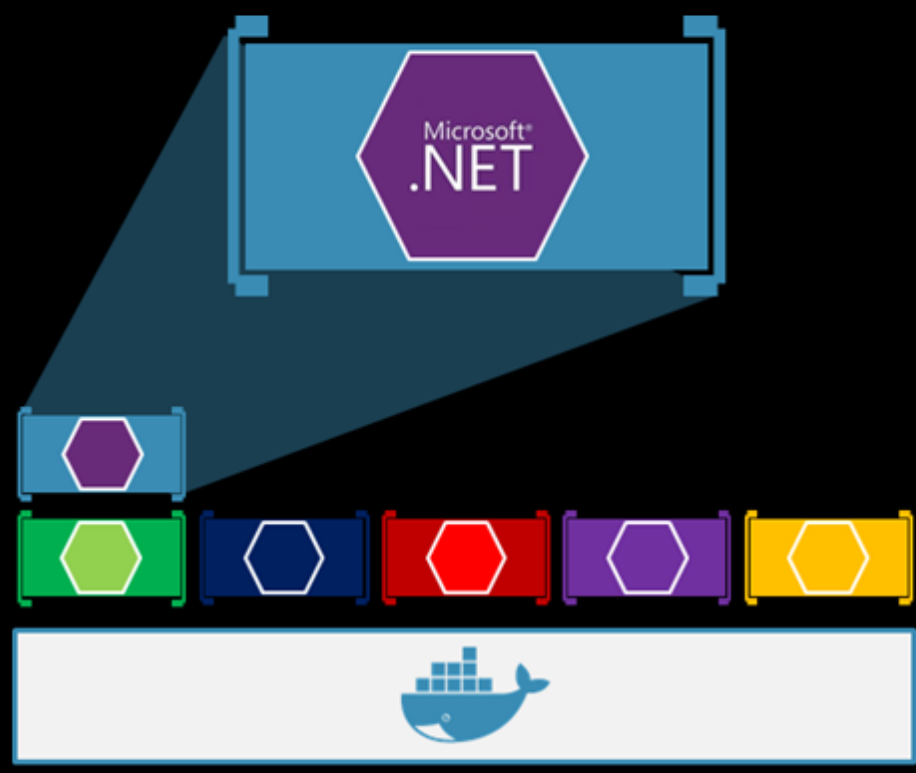Early Draft

Microsoft

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

Co-Authors:

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

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

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

Participants and reviewers:

Full name, Title, Team, Company

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

---------------------- *To be completed when finished the review process* -----------------------

----

Contents

Introduction

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

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

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

About this guide

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

You can think of this document as providing foundational design and development guidance at a development environment and architectural design level with clear focus on two technologies: Docker and .NET Core. Our intention is that you would read this document before making decisions about any specific cloud infrastructure like Microsoft Azure. After you've studied this guide, your next step would be to read about production-ready microservices on Microsoft Azure.

What we don't cover

This guide doesn't focus on the application lifecyle, DevOps, CI/CD pipelines, and team work. The complementary guide Containerized Docker Application Lifecycle with Microsoft Platform and Tools (https://aka.ms/dockerlifecycleebook) covers those topics in more detail.

Who should use this guide

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

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

How to use this guide

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

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

Introduction to containers and Docker

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

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

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

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

Docker Host

OS on a VM or
a server

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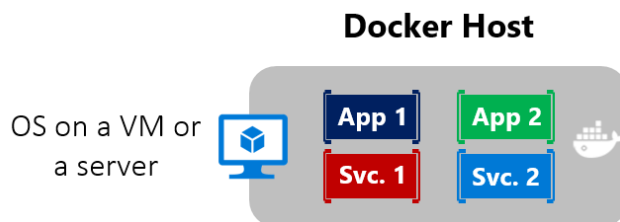
Figure 2-1. Multiple containers running on a container host

When running containers on regular Docker hosts, the isolation is not as strong as when using plain VMs. If you need further isolation than that provided by regular containers, Microsoft offers an additional choice: 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 than regular Windows Containers.

Another benefit of containerization is scalability. You can scale-up fast by 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 as instantiating a process (like a service or web app). For reliability, however, 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.

What is Docker?

[Docker](#) is an [open-source project](#) for automating the deployment of applications as portable, self-sufficient containers that can run on the cloud or on-premises. [Docker](#) is also a [company](#) that promotes and evolves this technology, working in collaboration with cloud, Linux, and Windows vendors, including Microsoft.
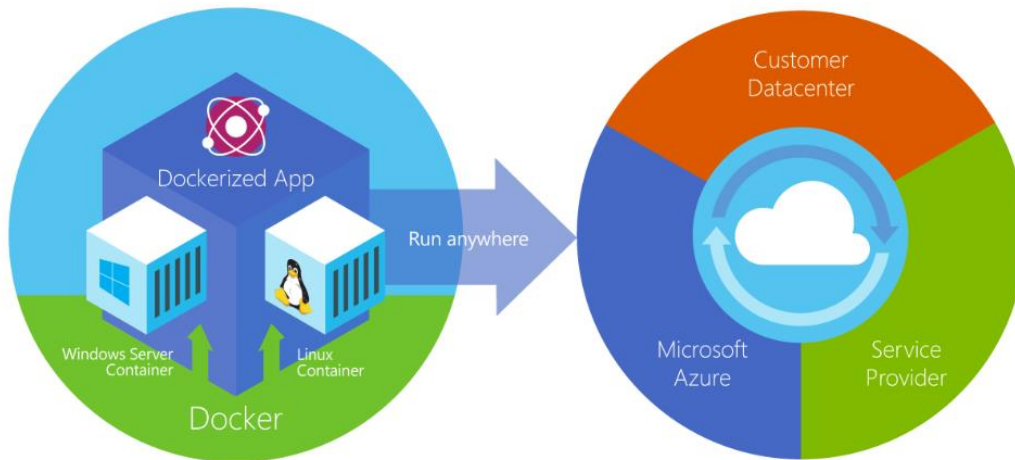
Figure X-X. *Docker deploys containers at all layers of the hybrid cloud*

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.).

Docker image containers can run natively on Linux and Windows. However, Windows images can only run on Windows and Linux images can only run on Linux.

In terms of development environment, you can use Windows, Linux or MacOS. When using Windows, you can develop for Windows containers and Linux Containers. However, MacOS is a development environment alternative just for Linux containers, not Windows Containers. Thus, you can edit code or run the Docker CLI from MacOS, but (at the time of this writing) containers do not run directly on MacOS.

When targeting Linux containers but using Windows or MacOS development machines, you will need a Linux host (typically a Linux VM) to run those Linux containers.

To host containers in development environments and provide additional developer tools, Docker ships [Docker Community Edition (CE)](#) for Windows and Mac (Aka *[Docker for Mac](#)* and *[Docker for Windows)](#)*. These products install the necessary VM to host Linux containers.

Docker also ships [Docke Enterprise Edition (EE)](#) designed for enterprise development and IT teams who build, ship and run business critical applications in production at scale.

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

Virtual Machines	Docker Containers
App 1 / App 2 / App 3 — Bins/Libs / Bins/Libs / Bins/Libs — Guest OS / Guest OS / Guest OS — Hypervisor — Host Operating System — Infrastructure	App 1 / App 2 / App 3 — Bins/Libs / Bins/Libs / Bins/Libs — Container Engine — Operating System — Infrastructure
Virtual machines include the application, the required libraries/binaries, and a full guest operating system. Full virtualization requires more resources than containerization.	Containers include the application and all of its dependencies but share the OS kernel with other containers, running as isolated processes in user space on the host operating system (except in "*Hyper-V containers*" where each container runs inside of a special virtual machine per container).

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

Docker containers are easy to deploy and quick to start up. As a side effect of running on the same kernel, you get less isolation than VMs, but also use far fewer resources. That allows you to have

Introduction to Containers and Docker

"higher density", meaning that it allows you to run more services on the same hardware unit, thereby reducing costs.

A Container Image is a way to package an app or service and deploy it in a reliable and reproducible way. So, you could say that Docker is not only a technology, but also a philosophy and a process.

When using Docker, you won't get the typical developer's excuse "*it works on my machine*". 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 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, like the basic cloud principles). In addition, you need no upfront infrastructure design, implementation and investment per project.

In a CaaS infrastructure, developers can build, test and deploy applications and IT operations can run, manage and monitor those applications in production.

From its original principles, it is partially like Platform as a Service (PaaS) in 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 image containers and those images are immutable once created.

In regards to host OS related updates, it is usually the responsibility of the person/organization owning the container image to perform the updates; 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 you should be familiar with 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/

Container Image: 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, you would 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 tags that correspond to historical versions of a specific image. Some repos contain multiple variations based on the SDK, runtime, fat vs. thin, etc. 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. Most companies will want to keep their images private and their network close to their deployment infrastructure, they can 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, and integration with GitHub and Bitbucket. Docker Hub is the public instance of a registry, similar to the public GitHub offering compared to the GitHub enterprise offering where customers store their code in their own environment.

Azure Container Registry: Centralized public resource for working with Docker Images and their components in Azure, a registry network close to your deployments with control over access, making it 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 Windows provides both Windows and Linux container development environments.

Docker for Windows and Docker for Mac replace 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 creates a container per image on a single Docker Host and does everything that

needs to be done to get the whole application 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 refer to it as a swarm instead of a cluster.

Orchestrator: A Docker Orchestrator simplifies management of clusters and Docker hosts. Orchestrators enable users to manage their images, containers and hosts through a user interface, either a command line interface (CLI) or graphical UI. This interface allows users to administer container networking, configurations, load balancing, service discovery, High Availability, Docker host management and much more. An orchestrator is responsible for running, distributing, scaling and auto-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.
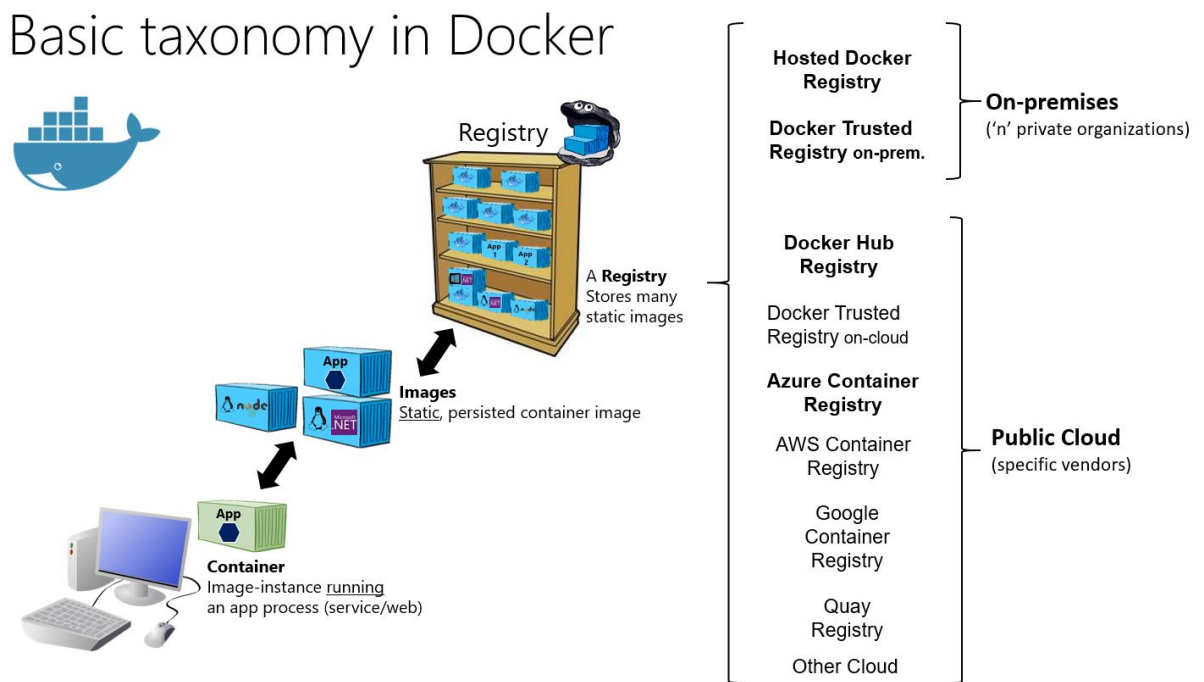
Figure X-X. Taxonomy of Docker terms and concepts

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

You should use a private registry (for example, using *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 image distribution pipeline.
- Integrate image storage and distribution tightly into your in-house development workflow

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

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

The explanation on "the why" for the following bullets are explained in the upcoming pages.

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

- You have cross-platform needs. For example, when you want to use both Linux and Windows containers.
- Your application architecture is based on microservices.
- You need to start containers fast and want a small foot-print per container.
- In summary, when creating new containerized .NET applications, you should try .NET Core as your "by default choice for Docker", as it has many benefits and fits much better with the containers philosophy and way of work.

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

- Your application currently uses .NET Framework and has strong dependencies on Windows
- You need to use Windows APIs not supported by .NET Core.
- You need to use third-party .NET libraries or NuGet packages not available for .NET Core.
- In summary, the ability of using .NET Framework on Docker can improve your deployment experiences minimizing deployment to production issues, so this "lift and shift" scenario is important for "dockerizing" legacy applications (probably not based on microservices).

Note than an additional benefit from .NET Core is that you can run side by side .NET versions for applications within the same machine, however this benefit is more important for plain servers or VMs not using containers because when using containers each container's image could also use a different .NET framework as long as they are compatible with the underneath OS (Linux or Windows).

When to choose .NET Core for Docker containers

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, like monolithic.

However, the modularity and lightweight nature of .NET Core makes it perfect for containers. When deploying and starting a container, the size of its image is far smaller with .NET Core than with .NET Framework because if using the full .NET Framework not only the framework is a lot heavier but in addition to that you have to use a Windows Server Core image which is a lot heavier than the Windows Server Nano or Linux images.

Additionally, .NET Core is cross-platform, so you can deploy server apps as Linux container images. But if using the full .NET Framework, you can only deploy as Windows containers images (and Windows Server Core, only).

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 is able to run on multiple platforms supported by Docker (Linux and Windows), the right choice is to use .NET Core, as .NET Framework only supports Windows.

.NET Core also supports MacOS as a development platform, but when deploying containers to a Docker host, that host currently must be based on Linux or Windows. For example, in a development environment you could use a Linux VM running on a Mac.

[Visual Studio](#) provides an Integrated Development Environment (IDE) for Windows and Mac. Visual Studio for Mac is an evolution of Xamarin Studio. You can also use [Visual Studio Code](#) on MacOS, Linux and Windows. Visual Studio Code fully supports .NET Core, including IntelliSense and debugging. You can also target .NET Core with most third-party editors like Sublime, Emacs, VI, and the open source Omnisharp project which also provides Intellisense support. In addition to the IDEs and editors you can also use the .NET Core comman-line tools (dotnet CLI), 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 can 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 on Containers

First of all, it is worth to say that you can use the full .NET framework for microservices based applications when using plain processes, as the .NET Framework is already installed and shared across

processes. However, if using containers, the required image for .NET Framework (Windows Server Core plus the full .NET Framework within each image) is probably too heavy for a "microservices on containers" approach.

On the other hand, .NET Core is the best candidate if you are embracing a microservices oriented system which is at the same time running on containers, because.NET Core is lightweight and most of all, its related container images, either its Linux image or its Windows Nano image are lean and small images.

A microservice is meant to be as small as possible, light when spinning up, small foot-print, a small Bounded Context, a small area of concerns and being able to start/stop microservices fast. For those requirements, you will want to use small and fast to instantiate container images like the .NET Core container images.

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

There are many orchestrators you can use when targeting microservices and containers.

For large and complex microservice systems being deployed as *Linux containers*, Azure Container Service with its multiple orchestrator offering (Mesos DC/OS, Kubernetes and Docker Swarm) is a great and more mature choice. You can also use Azure Service Fabric for Linux which also supports Docker Linux containers (Note: At the time of writing this offering was still in Preview. Check the Azure Service Fabric for the latest status).

For large and complex microservice systems being deployed as *Windows containers*, most orchestrators are currently in a less mature state, 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 density in scalable systems

When your container-based system needs the best possible density, granularity and performance, .NET Core and ASP.NET Core are your best options. ASP.NET Core outperforms the traditional 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 container images based on Linux or Windows Nano images you can run your system with a much lower number of servers/VMs, ultimately saving costs in infrastructure and hosting.

When to choose .NET Framework for Docker containers

While .NET Core offers significant benefits for new applications and application patterns, 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

You may want to use Docker containers simply for reasons related to deployment issues, as explained previously, nothing to do with microservices. It could be simply because you want to improve safety of your DevOps workflow and eliminate deployment issues caused by missing 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 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, for example 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 deciding.

However, consider 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 (.NET Core 2), but others don't apply to the new application patterns targeted by .NET Core and may never be available. The following list shows 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 nor server), although

plans are to include it 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 January 2017, WCF server implementation is only available on the .NET Framework. This scenario is being considered for future releases of .NET Core.
- Workflow related services: Windows Workflow Foundation (WF), Workflow Services (WCF + WF in a single service) and WCF Data Services (formerly known as "ADO.NET Data Services") are only available on 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 for .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 features 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/API that doesn't support .NET Core

Some Microsoft or third-party platforms don't support .NET Core. For example, some Azure services provide an SDK not yet available for consumption on .NET Core. This is temporary, as all of Azure services will eventually 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 from the Azure service instead of the client SDK.

Decision table - .NET frameworks to use for Docker

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

Consider 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 on Containers	**.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 and WebForms)	--	**.NET Framework**
SignalR services	**.NET Core** in upcoming releases	**.NET Framework** **.NET Core** in upcoming releases
WCF, WF and other traditional frameworks	WCF in **.NET Core** (In the Roadmap)	**.NET Framework** WCF in **.NET Core** (In the Roadmap)
Consumption of Azure services	**.NET Core** (Eventually all Azure services will provide Client SDKs for .NET Core)	**.NET Framework** **.NET Core** (Eventually all Azure services will provide Client SDKs for .NET Core)

What OS to target with .NET Containers

Given the diversity of Operating systems supported by Docker and the "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 figure X-X you can see the recommended OS version depending on the .NET frameworks.

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

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

When adding the image name to your dockerfile file, you can select the Operating System and version depending on the tag you use, as in the following examples.

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

Official .NET Docker images

The Official .NET Docker images are Docker images created and optimized by Microsoft and publicly available at Docker Hub within Microsoft's repositories.

Each repository may 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's 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) are intended to be used for console apps based on .NET Core. For example, batch processes, Azure WebJobs and other console scenarios should use .NET Core, which doesn't include the unnecessary ASP.NET Core stack, resulting in a smaller container image.

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

For further information about the official .NET Docker images provided by Microsoft, see the Official .NET Docker Images reference.

.NET Core and Docker image optimizations per variant

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

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

Why multiple images? When developing, building and running containerized applications, you usually have different priorities.

Development and Building: When developing, what's important is how fast you can iterate changes, and the ability to debug the changes. The size of the image isn't as important as the ability to make changes to your code and see them quickly. Some of our tools, like yo docker for use in Visual Studio Code, use this image during development time. When building "inside a Docker container", what's important is what's needed to compile your app. This includes the compiler and any other .NET dependencies plus web development dependencies like NPM, Gulp, Bower, etc.

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

Production: What's important in production is how fast you can deploy and start your containers based on a "production .NET Core image". Therefore, 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 compilation from C# code like when running "dotnet build". 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 contains the compiled .NET binaries, images, .js and .css files. Over time, you'll see images that contain pre-jitted packages.

Although there are multiple versions of the .NET Core 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 the images share the same base layer, and potentially others.

Therefore, 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-runtime-deps**	.NET Core 1.1, with runtime and framework dependencies for self-contained apps, on Linux
microsoft/dotnet**:1.1.0-sdk-msbuild**	.NET Core 1.1 with SDK included, on Linux

Architecting container and microservice based applications

Vision

Architect and design scalable solutions with containers in mind.

There are many great-fit use cases for containers, not just for microservices oriented architectures but also for regular services or web applications where you want to reduce friction between development and deployment to production environments.

Architecting Docker applications

In the first section of this document you learned 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 of multiple services instead of a single service/container. For those optional use cases, you need to understand further architectural approaches such as Service Orientation and the more advanced Microservices and container orchestration concepts. However, the scope of this document is not limited to "microservices on containers" but also taking into account any containerized application.

Common container design principles

Instance of container image equals a process

In the container model, a container image instance represents a single process. By defining a container image as a process boundary, you start to create the primitives used to scale, or batch off processes. When designing a container image, you'll see an ENTRYPOINT definition. This defines the process whose lifetime controls 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 formerly might 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.

Architecting microservices and containers

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.

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.
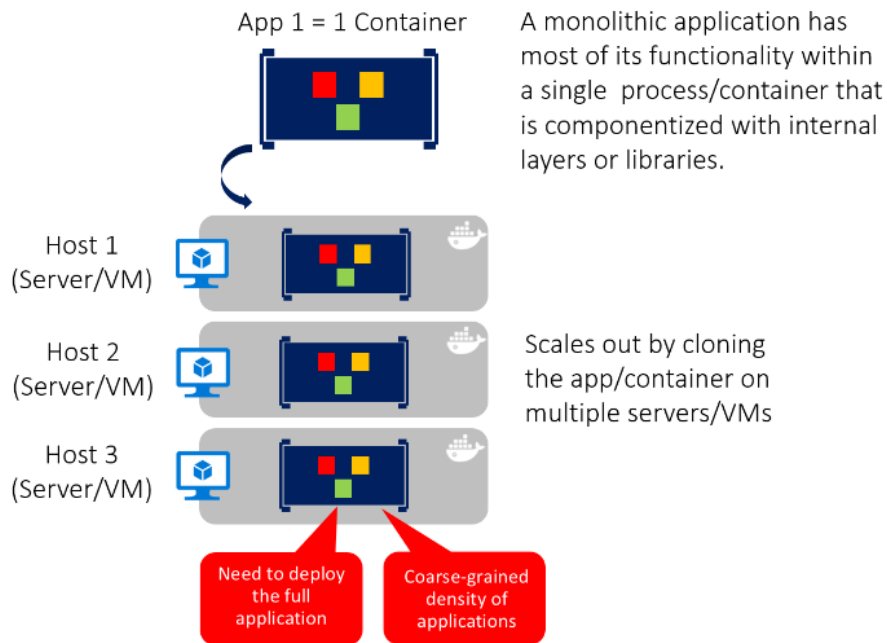
Figure X-X. Monolithic application architecture example

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

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 to scale is the product information component. Many more customers browse products than purchase them. More customers use their basket than use the payment pipeline. Fewer customers add comments or view their purchase history. And you likely 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.

fThe 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 they didn't see the need - until the app grew.
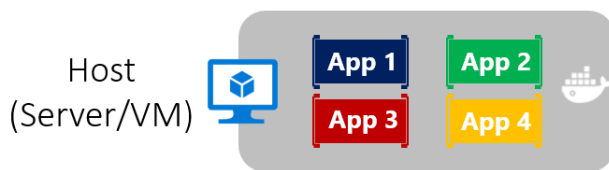
Figure X-X. Host running multiple apps/containers

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 X-X.

Deploying monolithic applications in Microsoft Azure can be achieved using dedicated VMs for 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.
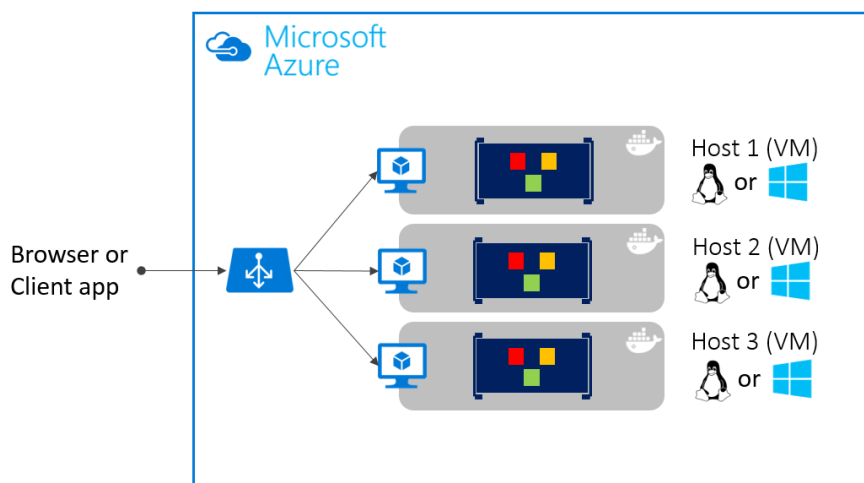
Figure 5-3. Example of multiple hosts scaling-out a single container application apps/containers

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, or through automation such as Continuous Delivery (CD) pipelines.

Monolithic application deployed as a container

There are benefits of using containers to manage monolithic application deployments. Scaling the instances of containers is far faster and easier than deploying additional VMs. Even using VM Scale Sets to scale VMs, 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 is far faster and network efficient. Docker Images typically start in seconds, speeding rollouts. Tearing down a Docker instance is as easy as issuing a `docker stop` command, typically completing in less than a second.

As containers are inherently immutable by design, you never need to worry about corrupted VMs, whereas update scripts might forget to account for some specific configuration or file left on disk.

While monolithic apps can benefit from Docker, we're only touching on the potential benefits. Larger benefits of managing containers come from deploying with container orchestrators which manage the various instances and lifecycle of each container instance. Breaking up the monolithic application into 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

Whether you want to get a quick validation of a container deployed to Azure or when an 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 with. It provides great git integration to take your code, build it in Visual Studio and directly deploy it to Azure.
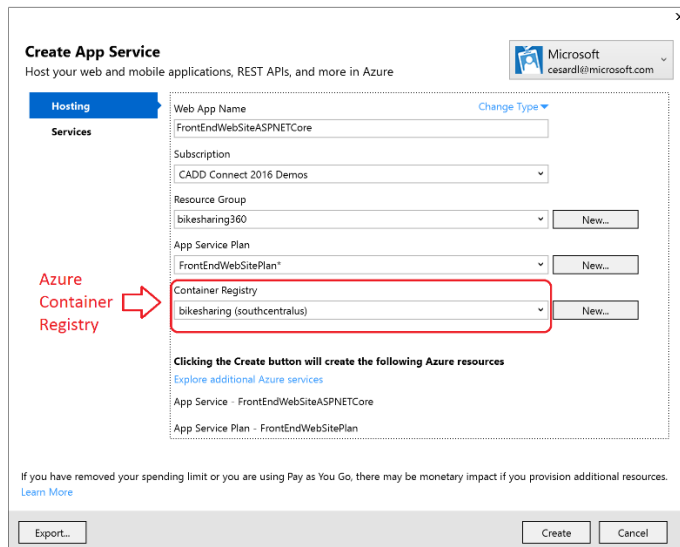
Figure X-X. Publishing a Container to Azure App Service from Visual Studio apps/containers

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

Container support in Visual Studio 2017 gives you the ability to include whatever you want in your app environment, as shown in Figure X-X. Since you are running it in a container, if you add a dependency to your app, you now have the capability of including the dependency in your dockerfile or Docker image.

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

State and data in Docker applications

Think of a container as an instance of a process. A process doesn't maintain durable state. While a container can write to its local storage, assuming that an instance will be around indefinitely would be like assuming that a single location in memory will be durable. Container images, like processes, should be assumed to have multiple instances or killed, or when managed with a container orchestrator, they may get moved from one node/VM to another.

Docker uses a feature known as an Overlay File System to implement a copy-on-write task 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 or if its related Docker host or node in a cluster gets out of order with all its local storage. Thus, while it's possible to save the state of a container, designing a system around this would 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.
- Data volume containers which provide shared storage across containers, using an external container.
- Volume Plugins which mount volumes to remote services, providing long term persistence.
- Remote data sources like SQL, NoSQL databases or cache services like Redis.
- Azure Storage which provides geo distributable storage, providing a good long term persistence solution for containers.

Data volumes. A data volume is amaped directory from the host OS to a directory in the container so that when code in the container access to the directory, the access is really to a directory on the host OS. This directory is not tied to the lifetime of the container itself and can be accessed from code running directly on the host OS or by another container that maps the same host directory into itself. Thus, data volumes are designed to persist data independent of the container's life cycle. Docker never automatically deletes 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, which 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 outside of the containers themselves but within the host server/VM physical boundaries. Docker volumes can't access a volume from one host server/VM to another.
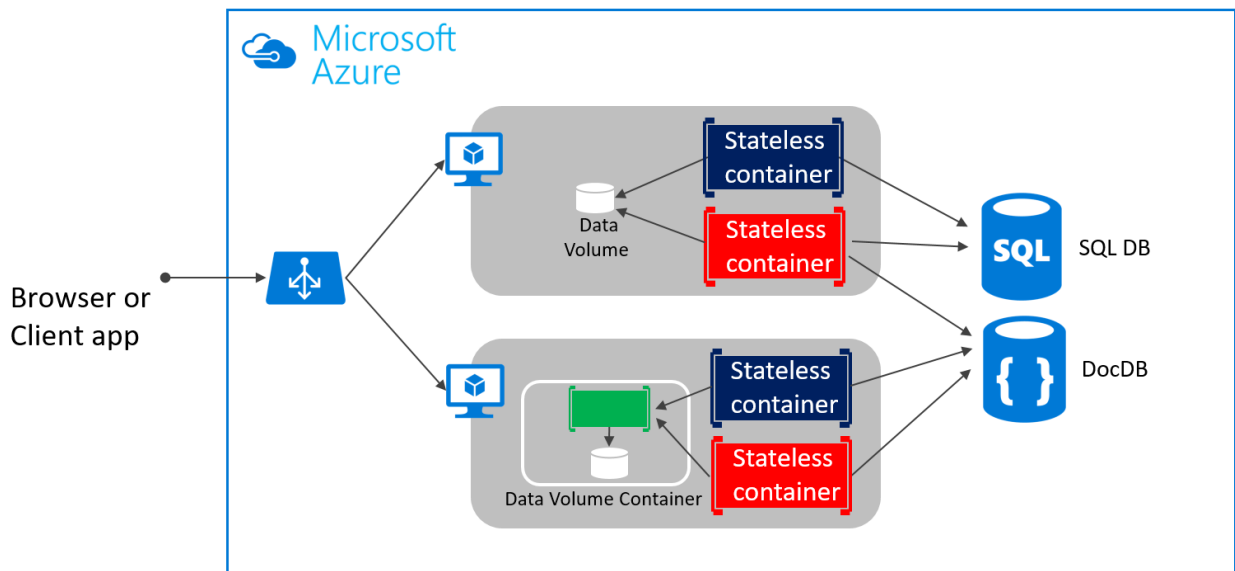
Figure X-X. Data Volumes and external data sources for containers apps/containers apps/containers

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. When using Docker containers in an orchestrator, containers are expected to be moved between hosts 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 killed/started across multiple hosts or cluster nodes.

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

Azure Storage provides the following four services in the cloud: Blob storage, Table storage, Queue storage, and File storage.

- *Blob Storage* stores unstructured object data. A blob can be any type of text or binary data, such as a document or media files (images, audio and video files). Blob storage is also referred to as Object storage.
- *File Storage* offers shared storage for legacy applications using the standard SMB protocol. Azure virtual machines and cloud services can share file data across application components via mounted shares, and on-premises applications can access file data in a share via the File service REST API.
- *Table Storage* stores structured datasets. Table storage is a NoSQL key-attribute data store, which allows for rapid development and fast access to large quantities of data.

Service-oriented architecture applications

Service-oriented architecture (SOA) was an overused term and meant many different things to different people. But as minimum and common denominator, SOA or service orientation means that you structure the architecture of your application by decomposing it into 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 now be deployed as Docker containers, which 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 scalability and availability challenges if you are deploying based on single Docker Hosts. 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 to 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

As the name implies, a 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(S), WebSockets or AMQP. Each microservice implements a specific end-to-end domain/business capability within a certain *Bounded Context* and must be developed autonomously and be deployable independently. Finally, each microservice should own its related domain data model and domain logic (sovereignty and decentralized data management), and can employ different data storage technologies (SQL, NoSQL) 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 set 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 a microservices architecture? In short, it provides long term agility. 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 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. That means saving costs as you need less hardware.
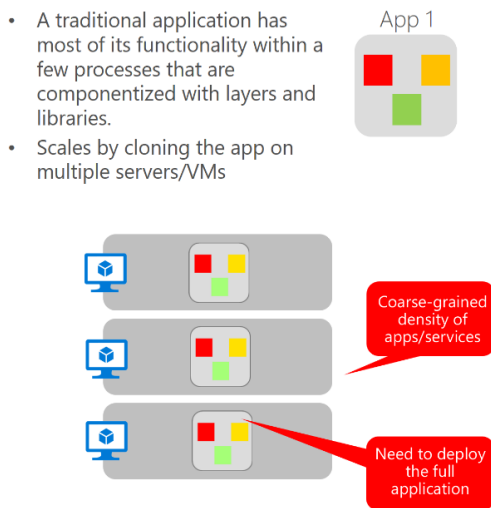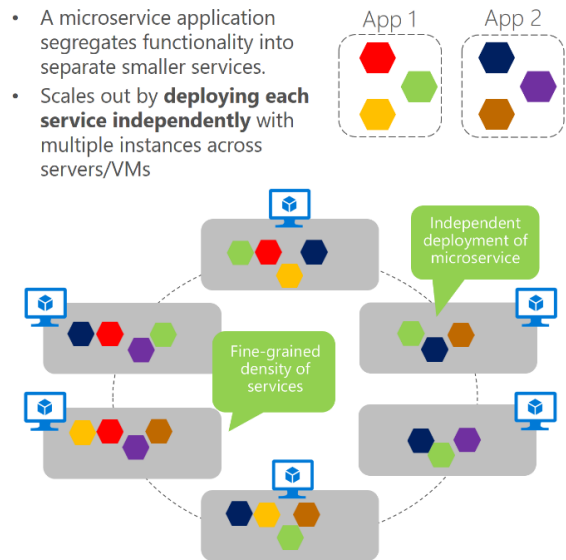
Figure X-X. Monolithic deployment vs. the Microservices approach

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.

Architecting fine-grained microservice based applications enables continuous integration and continuous delivery 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.

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

- Basic monitoring of the services and infrastructure
- Rapid provisioning of infrastructure for the services
- Rapid application delivery
- Devops culture

Microsoft Platform and Tools for Docker

From those topics, only the first one is covered by this book. The rest of them which are related to application lifecycle are covered in the additional eBook named *Containerized Docker Application Lifecycle with Microsoft Platform and Tools*, referenced in the table below.

References – Microservices 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 **Chunk Cloud Computing** https://www.infoq.com/articles/CCC-Jimmy-Nilsson **Containerized Docker Application Lifecycle with Microsoft Platform and Tools** https://aka.ms/dockerlifecycleebook

Data Sovereignty Per Microservice

An important rule to follow 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) or autonomous 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 data) approach used in many applications is to have a single centralized database (or just a few databases), often a normalized SQL database, for the whole application and all its internal subsystems, as shown in figure X-X.

Data in Traditional approach

- Single monolithic database
- Tiers of specific technologies

Web Tier

Services Tier

Cache doesn't help much for massive data ingress (Events, IoT, etc.)

Cache Tier

Database servers are usually the bottleneck

Data Tier

SQL DB
or
No-SQL

Monolithic Databases are shared across services.
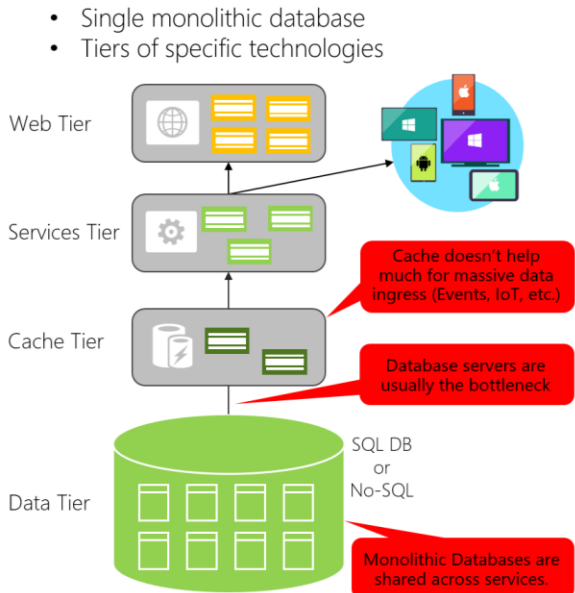
Data in Microservices approach

- Graph of interconnected microservices
- State typically scoped to the microservice
- Remote Storage for cold data

Mobile apps

Web presentation services

Stateless services

SQL
[...]

Stateless services with separate store

Stateful services

Each microservice owns its model/data!
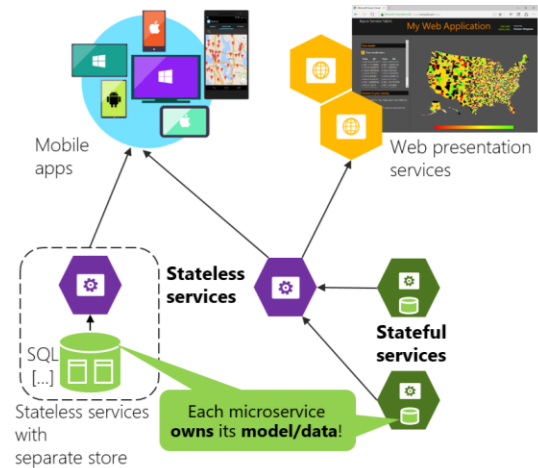
Figure X-X. Data Sovereignty Comparison: Microservices vs. Monolithic DB

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, or learning geography.

A monolithic application with typically a single relational database has two important benefits. First, ACID transactions and second the SQL language, both working across all the tables and data related to your app. This provides a very simple way to easily write a query that combines data from multiple tables. However, data access becomes much more complex when you move to a microservices architecture. 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 would require coordinated updates to all the services and that would eliminate the microservice lifecycle autonomy. But this type of distributed data structures provokes that you cannot take a single ACID transaction across microservices. This means that eventual consistency must be used and this is much harder for developers to implement because SQL joins and many more relational database features are also not available across multiple microservices, making the development work much harder.

Going even further, different microservices often use different kinds of databases. Modern applications store and process diverse kinds of data and a relational database is not always the best choice. For some use cases, a particular NoSQL database (such as Azure DocumentDB or MongoDB) might have a more convenient data model and offer much better performance and scalability than a SQL database like SQL Server or Azure SQL DB. In other cases, a relational DB is still the best

approach. Therefore, microservices-based applications often use a mixture of SQL and NoSQL databases, the so-called polyglot persistence approach.

A partitioned, polyglot-persistent architecture for data storage has many benefits, including loosely coupled services, better performance, scalability, costs and manageablilty. However, it does introduce some distributed data management challenges that will be explained in a later section named "Identifying domain-model boundaries".

Relationship between Microservices 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. While each BC must have its own model and database, likewise each microservice owns its related data. In addition, each Bounded Context usually has its own Ubiquitous Language to help communication between software developers and domain experts.

Those terms (mainly Domain Entities) in the Ubiquitous Language can be named differently between different Bounded Contexts even when different Domain Entities might share the same Identity (i.e. the unique id value with which the entity would be retrieved from persistence). 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 separate process per Bounded Context and must use the mentioned distributed protocols like HTTP(S), WebSockets or AMQP. The Bounded Context pattern, however, doesn't specify whether it is a distributed service or if it is simply a logical boundary within a monolithic-deployment application. Ultimately, both patterns are very much related.

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

References – Data Sovereignty per Microservice and Bounded Context patterns
"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   **Context Map:** https://www.infoq.com/articles/ddd-contextmapping

Identifying domain-model boundaries per microservice

The goal when identifying model boundaries and size for each microservice is not to get to the most granular separation possible, although you should tend toward small microservices, if possible. Instead, your goal should be to get to the most meaningful separation guided by your domain knowledge. The emphasis is not on the size, but instead on the business capabilities. Also, if there is clear cohesion needed for a certain area of the application, that would probably is going to be a microservice, too. Cohesion is a way to identify how to break apart or group together microservices.

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 earlier in this book.

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

A DDD technique that can be used for this is the Context Mapping pattern. Via Context Mapping, 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 number of attributes when dealing with a CRM expert who wants to store every detail about the customer than for an ordering domain expert who just needs partial data about the customer. It requires lots of coordinated efforts to disambiguate all terms across all domains. And worse, if you try to have a single unified database for the whole application, this 'unified vocabulary' 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 are clearly defined and that you don't have to switch between models too often.

So perhaps 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 figure X-X you can see how multiple microservices (multiple Bounded Contexts) with its own model for each microservice and how their entities can be defined depending on your specific requirements for each of the identified Domains in your application.

Identifying a domain model per microservice or Bounded Context
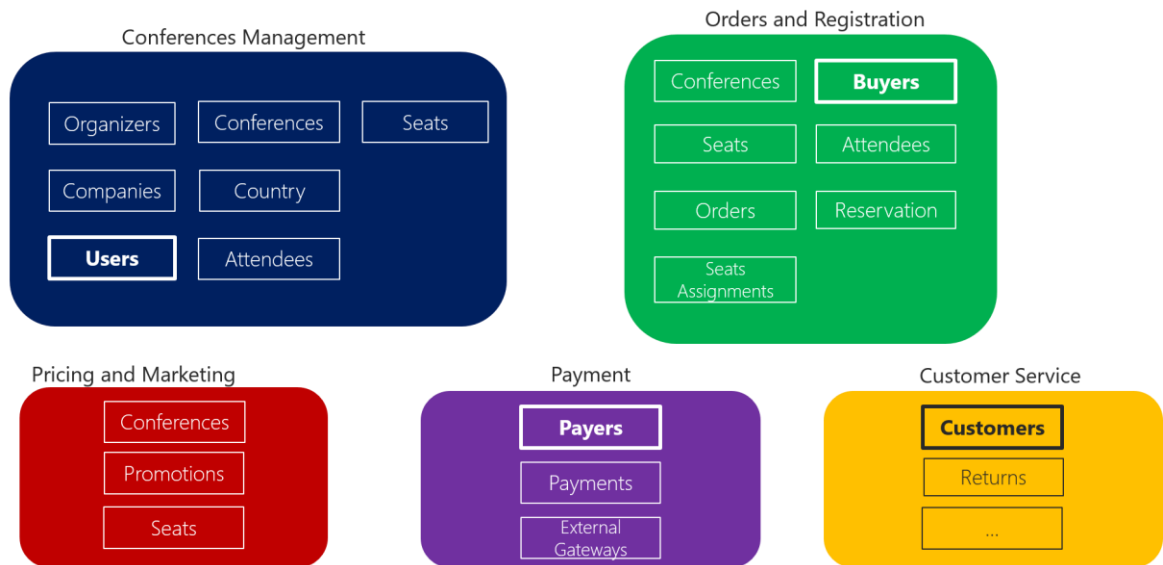
Figure X-X. Identifying entities and microservice's model boundaries

In that same figure X-X you can see a sample scenario related to an online Conference 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 have a different flavor or shape but share the same identity 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 Buyers in the Ordering microservice, or named as Payer in the Payment microservice and even present in the Customer Service microservice 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 named 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)
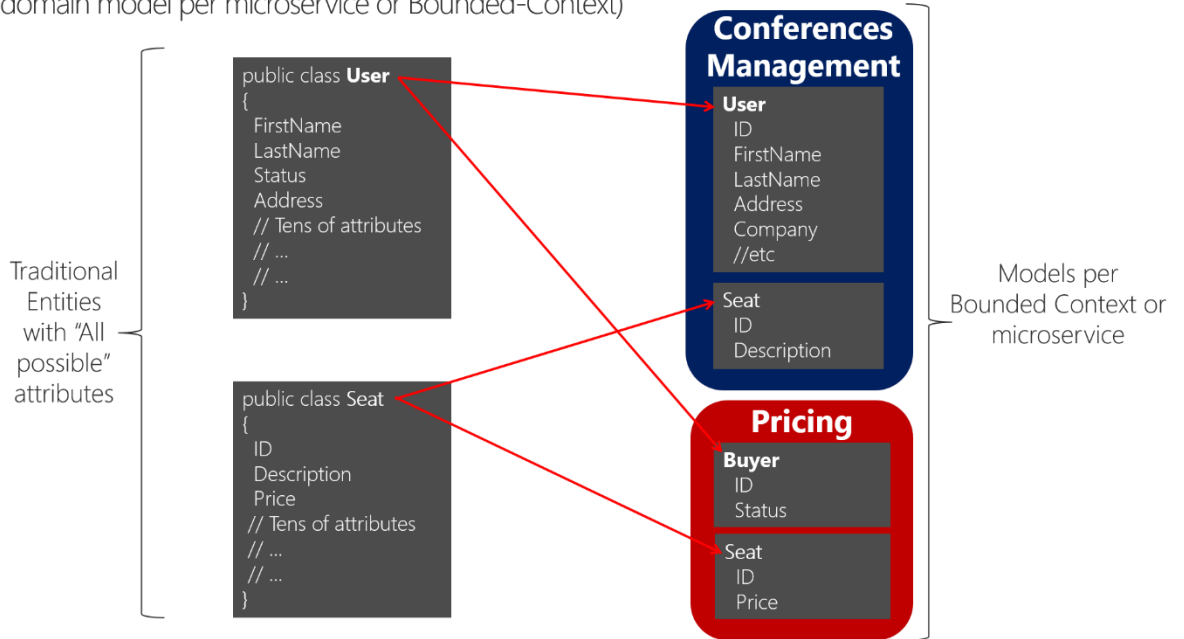
Figure X-X. Decomposing traditional data models into multiple domain-models

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 in the Pricing microservice, with alternate attributes or details about the user when it is actually a buyer. Each microservice or Bounded Context might not need all the data related to a User but just part of it, depending on the problem to solve or the context. For instance, in the pricing microservice's model you don't need the address or the id/passport number of the user but just his ID (as identity) and the Status 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.

Basically, there is a shared concept of user that exists in multiple services (domains), sharing the identity of the user but in each domain model there may be some additional or different details about the user entity. Therefore, there needs to be a way to map a user entity from one domain (microservice) to another domain (microservice).

The potential benefits of not sharing the exact same user entity with the same number of attributes across domains (services) are about reducing redundancy so each microservice's model doesn't have data that does not need and also about having a "master" microservice owninmg certain type of data per entity so updates and queries for certain type of data must be driven only by that specific microservice.

Challenges and solutions for Distributed Data Management

Challenge #1: How to maintain consistency across multiple services

As stated previously, the data owned by each microservice is private to that microservice and can only be accessed via its microservice API. The first challenge presented by this approach is how to implement business transactions that maintain consistency across multiple microservices.

To analyze this problem, let's look at an example from the eShopOnContainers reference application. The Catalog microservice maintains information about all the products, including their stock level. The Ordering microservice manages orders and must verify that a new order doesn't exceed the available catalog product's stock. In a hypothetical monolithic version of this app, the Ordering subsystem could simply use an ACID transaction to check the available stock, create the order in the Orders table and update the available stock in the Products table.

In contrast, in a microservices architecture the Order and Product tables are private to their respective services, as shown in image X-X.
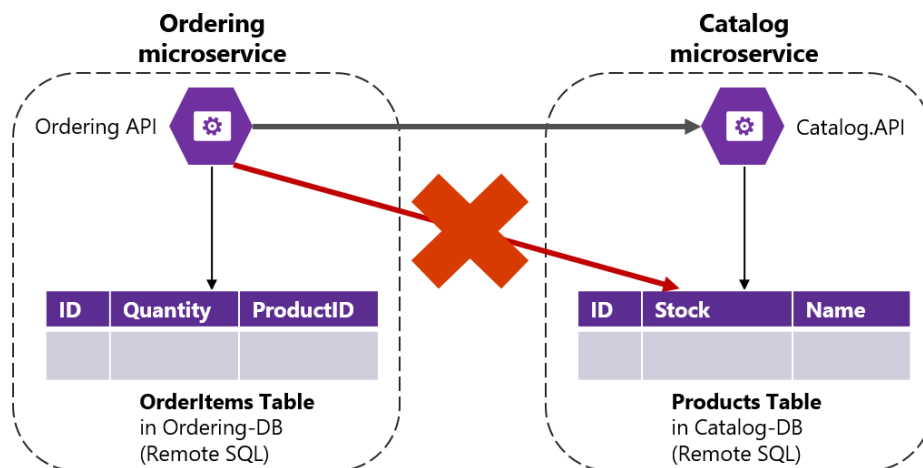
Figure X-X. Cannot access directly Tables from other microservices

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. Many scenarios demand the availability as opposed to strong consistency. Mission-critical apps must remain up and running and developers can work around strong consistency by using techniques for working with weak/eventual consistency, which is precisely the approach taken by microservice-based architectures. Moreover, ACID-style or two-phase commit transactions are not just against microservices principles, but most NoSQL databases (like Azure Document DB, 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 can I propagate changes

across multiple microservices when certain data needs to be redundant? – i.e. When having the product's name or description in the Catalog microservice and the Basket microservice".

A good solution for both questions is based on eventual consistency between microservices articulated through event-driven communication and a publish/subscription system, topics which are covered in the section named "Asynchronous Event-Driven communication" later in this book.

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

The second challenge is the question of how you can implement queries that retrieve data from multiple services while avoiding a super-chatty communication from remote client apps taking to those microsrevices. An example could be a single screen from a mobile app that needs to show user's info owned by the basket, catalog and user identity microservices. Another example would be a complex report involving many tables located in multiple microservices. The right solution 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 recommended approach would be to handle the aggregation in an "Aggregation microservice". This approach is known as the API Gateway pattern which is explained in the following section when talking about inter-microservice communication.

B. **CQRS "Query-Tables":** This solution is also known as the [Materialized view pattern](#) that pre-joins data owned by multiple microservices. Think about a similar case but using a single database. In that case, you would use a very complex join that you implement with a SQL query involving multiple tables. However, when handling multiple databases, each database owned by a different microservice, you cannot query those databases to make a SQL join, therefore, you can address that necessity with a CQRS approach by creating a de-normalized "Query-Table" in a different database used just for queries. That table will be designed based on 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 the query-table. This approach not only solves this 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 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 and the fact that you have you data split in multiple databases might require these types of approaches and solutions.

C. **"Cold-Data" in central databases**: For complex reports and queries that might not require real-time data, a common approach is to export your "hot data" (transactional data from the microservices) into large databases only used for reporting. That central database system can be a Big Data database like Hadoop, a Data Warehouse based like Azure SQL Data Warehouse or even a single SQL database used just for reports if size won't be an issue. Keep in mind that this centralized database would be used only for queries and reports that don't need real-time data. The original updates and transactions, as "*your source of truth, 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 process would be pretty similar to the way you propagate data in the mentioned "CQRS Query Tables".

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

Direct Client-to-Microservice communication vs API Gateway pattern

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

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 some of the microservices, as shown in figure X-XX.
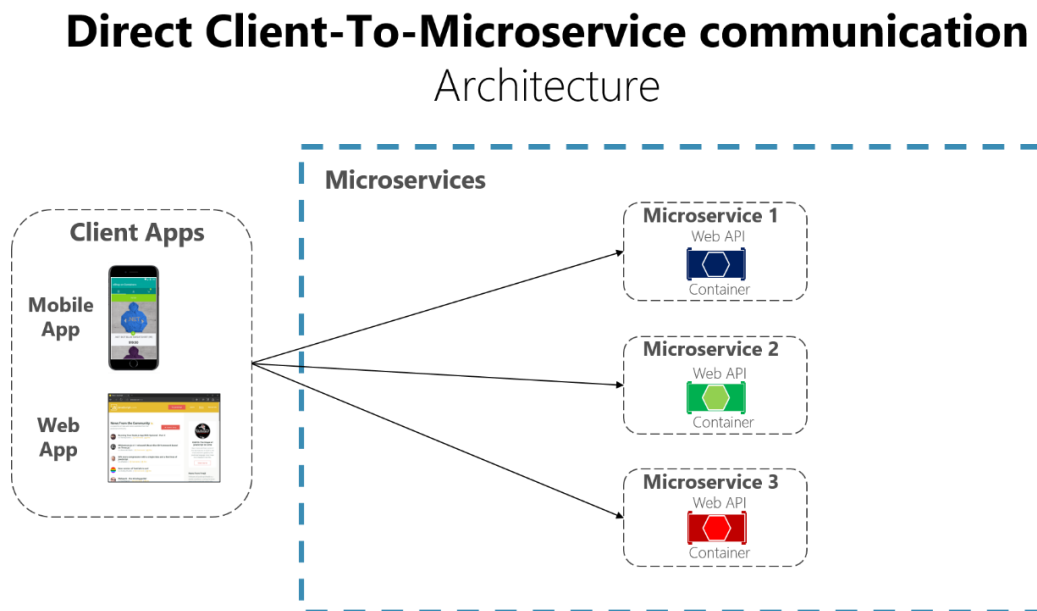
Figure X-XX. Using the Direct Client-To-Microservice communication architecture

Microsoft Platform and Tools for Docker

Each microservice has a public endpoint like https://applicationname.cloudprovider.com, sometimes with a different TCP port per microservice, like this URL in Azure: http://eshoponcontainers.westus.cloudapp.azure.com:88/.

In a production environment based on a cluster, that URL would map to the microservice's load balancer, which distributes requests across the available instances.

This *Direct Client-To-Microservice communication architecture* is good enough for a small microservice-based application, however when building large and complex microservice-based application (for example, when handling tens of microservice types) that approach faces possible issues as explained in the following cases.

You need to consider the following questions when developing a large application based on microservices:

- *How 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. This approach actually reduces latency since multiple pieces of data come back in parallel and some UI can show data as soon as its ready.

- *How to allow client apps 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 to protocols like HTTP(S).

- *How can you handle cross-cutting concerns such as authorization, data transformations, and dynamic request dispatching*? – Implementing security and cross-cutting concerns on every microservice can require significant development effort. A possible approach would be to have those services within the Docker host or internal cluster restricting access from the outside and implementing those cross-cutting concerns, like security and authorization, in a centralized place, like an API Gateway.

- *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 good approach to be considered for your architecture is known as an API Gateway. An API Gateway is a service that is the single-entry point into the internal application's microservices. 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:

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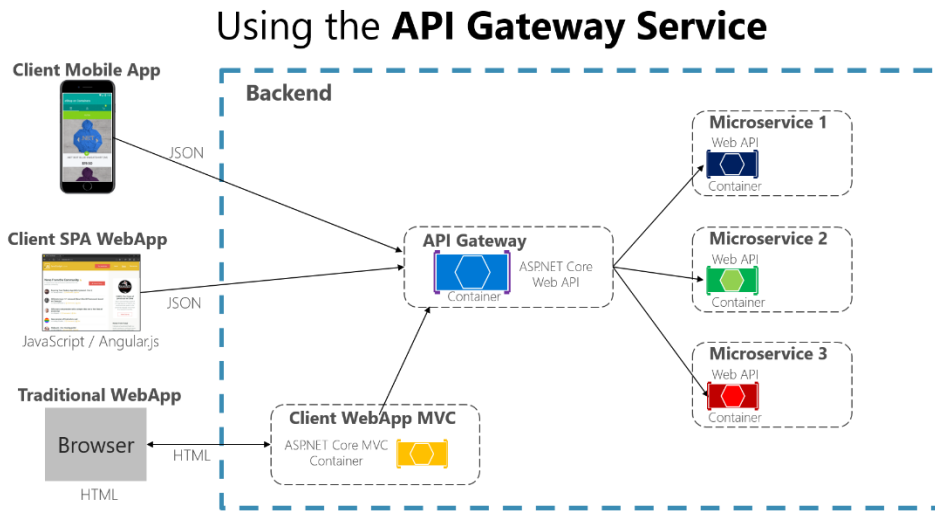
Figure X-XX. Using the API Gateway pattern in a microservice based architecture

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 that mentioned API Gateway service.

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. This allows you to get a better understanding of how your APIs are being used and performing by viewing near real-time analytics reports and identifying 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
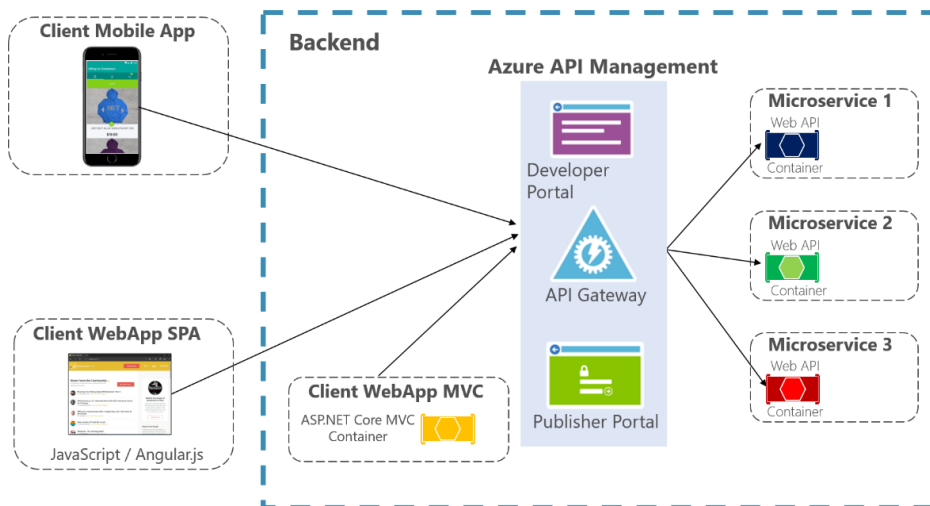
Figure X-XX. Using Azure API Management for your API Gateway

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 you 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

On one hand, in a monolithic deployment application, components invoke one another via language-level method or function calls; strongly coupled if creating objects with code like "new ClassName" or in a decoupled way if using Dependency Injection by referencing to abstractions rather than concrete object instances. Either way, the objects are running within the same process. The biggest issue in changing a monolith into microservices lies in changing the communication pattern. A direct conversion from in-memory method calls to RPC calls to services leads to chatty communications which don't perform well in distributed environments. Instead you need to replace the fine-grained communication with a coarser -grained approach by grouping calls and datasets being returned as much as possible.

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 protocol such as *HTTP, AMQP* or a binary protocol like TCP, depending on the nature of each service.

The microservice community promotes "smart endpoints and dumb pipes", which means to be as decoupled as possible between microservices and as cohesive as possible within a single microservice. As introduced, each microservice owns its own domain logic, but the microservices composing an application are usually choreographed using simple REST approaches rather than complex protocols such as WS-* or a centralized ESB.

The two protocols used most commonly are HTTP request-response with resource API's (when querying, most of all) and lightweight asynchronous messaging when communicating updates across multiple microservices, explained in more detail in the next sections.

Communication Types

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

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

Microsoft Platform and Tools for Docker

- *Synchronous* – The client waits for a response from the service. The wait time typically 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 logic, you can expect immediate responses, or responses returning much later. It doesn't impact client execution as it isn't blocked. When using asynchronous mechanisms, the overall performance can be better balanced as you don't have the bottlenecks associated with synchronous communication, however, development and debugging can be 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.
  - An example of this type of communication is the 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 the dimensions are applied in a complementary way.

	Synchronous	Asynchronous
One-to-One	Request/response	Request/async response
		Fire and forget (Notification)
One-to-Many	--	Publish/Subscription
		- Registration action
		- Publish action
		- Message Handlers

A microservice-based application will often 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 as implemented in an event-driven architecture is very flexible and convenient.

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 between services internally (within your Docker host or microservices cluster) you might also want to use binary format communication mechanisms (i.e. Service Fabric Remoting or WCF using TCP and binary format) . 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). If your chosen binary format is not a standard, it is probably not a good idea to

publicly publish your services using that format. You could use a non-standard format for internal communication between your microservices, like when communicating between microservices within your Docker host or microservice cluster (Docker orchestrators or Azure Service Fabric) or for very particular or propietary client applications talking to the microservices.

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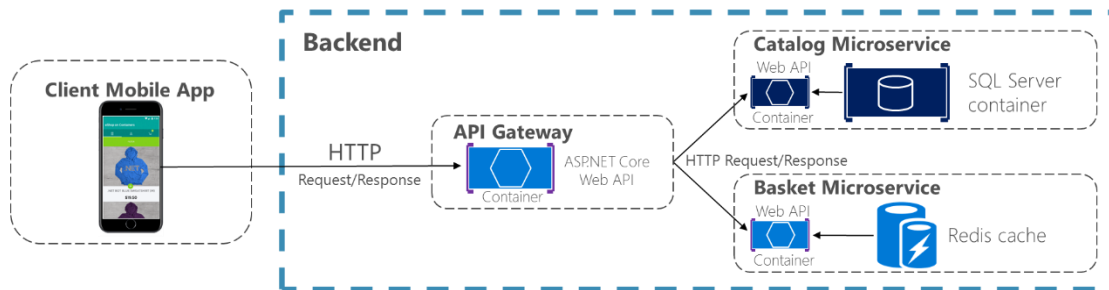
Figure X-XX. Using HTTP Request/Response communication (Sync or Async)

If it is a synchronous request/response communication, the thread that makes the request is blocked while waiting for a response. That's how it behaves in .NET when consuming an ASP.NET Web API synchronously. However, you 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 will usually have a call-back method in the client that will be called by the service when returning the call response. In modern languages that is simplified - using modern async/await keywords in C# you can program async services and client calls in a simplified way, as if if you were invoking synchronous methods. You can therefore communicate asynchronously with ASP.NET Web API services.

When using a request/response communication (either sync or async), the client assumes that the response will arrive in a timely fashion, typically less than a second or a few seconds at most. For 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 commonly used architectural communication approach when creating services. You can implement REST services when developing ASP.NET Core Web API services, as will be explained in the implementation sections of this document.

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

For more information about REST or HTTP, see the following reference.

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

Asynchronous Message-Based Communication

Asynchronous messaging and event-driven communication are critical when propagating changes across multiple microservices and their related Domain Models. As mentioned when discussing microservices, Bounded Contexts and how can you identify each model for each microservice, a User, Customer, Product, Account, etc. may mean different things to different Bounded Contexts or microservices. That means that you'll need some way to reconcile changes across the different models when changes happen. This is where event-driven communication based on asynchronous messaging must be used.

When using messaging, processes communicate by 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 it is a message-based communication, the client is assuming that the reply will not be received immediately or even with no response at all.

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.

In regards the needed infrastructure for this type of communication, the preferred approach in the microservices community is to use a lightweight message broker. The infrastructure chosen is typically dumb, acting just as a message broker, with simple implementations such as RabbitMQ or a scalable service bus in the cloud like Azure Service Bus. In any case, most of the "smart thinking" still lives in the end points that are producing and consuming messages; in the microservices.

There are two kinds of messaging communication, "one-to-one" communication and "publish-subscribe" communication.

One-to-One Asynchronous message communication

A one-to-one communication delivers a message to exactly one of the consumers that is reading from the channel, so the same message should be processed just once. There are special situations that you also need to take into account. For instance, in a cloud system that embraces failure and tries to automatically recover from failures, the same message culd be sent multiple times. Due to network or other failures, the client will have to retry sending messages and the server will have to implement an operation to be idempotent in order to process a particular message just once.

A publish-subscribe channel delivers each message to all 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 example, if one microservice's data is updated but that same data needs to be propagated to a different microservice, that kind of inter-microservice communication should be based on asynchronous communication by using integration events between microservices, as in image X-XX.
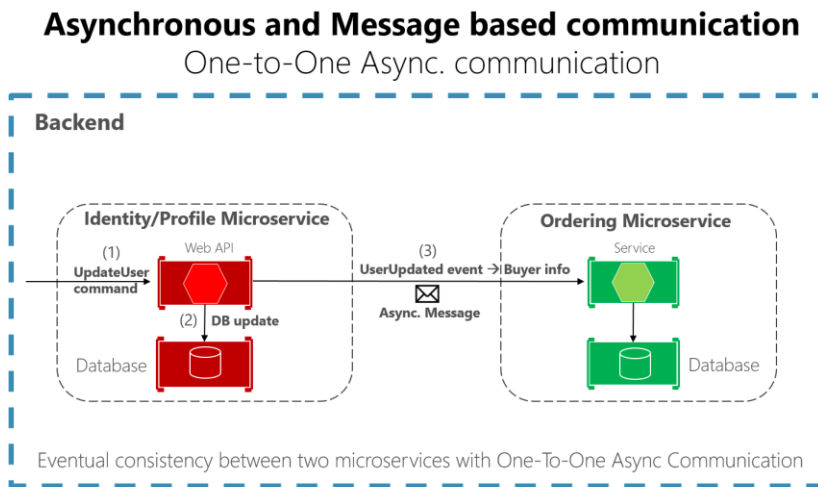
Figure X-XX. One-to-One async message communication

It is also worth to call out that in the One-to-One asynchronous communication approach, typically the sender is aware of and tightly coupled to the channel used by the receiver. Thus, any additional channels will require updates and deployments of the sender microservice

One-to-Many Asynchronous message communication

Additionally, you might want to use a Publish/Subscribe mechanism so your communication from the sender will be available to additional subscriber microservices or even external applications in the future. Thus, it helps you to follow the Open/Close principle for the sending service, since additional subscribers can be added in the future without the need to modify the originating service.

When using a Publish/Subscribe communication you might be using an Event-Bus interface to publish events to any subscriber. Another possibility (usually for different purposes) is a realtime and one-to-many communication with higher level frameworks such as ASP.NET SignalR and protocols such as WebSockets.

Asynchronous Real-Time communication

As shown in image X-XX, real-time asynchronous communication means that you can have server code pushing content to connected clients as it becomes available, rather than having the server wait for a client to request new data.

Asynchronous Real Time communication
One-to-many communication
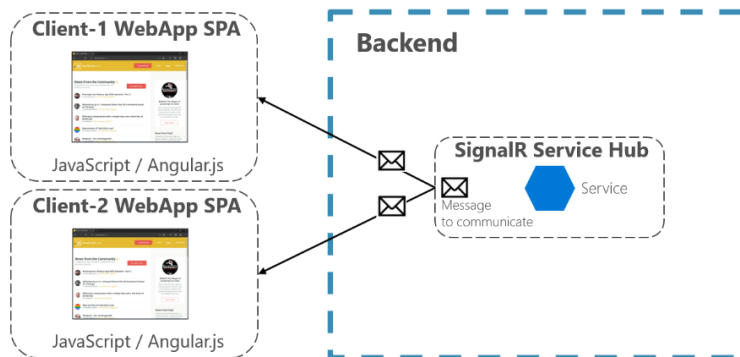
Figure X-XX. One-to-One async message communication

Since it is real-time, client apps will show the changes almost instantly. This is usually handled by a protocol such as WebSockets and underneath it is using many WebSocket connections; 1 per client. A typical example is when a service communicates a change in the score of a sports game to many client web apps, simultaneously.

Asynchronous 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. This subscription/publication system is usually performed by using an implementation of an Event Bus. The Event Bus can be designed as an abstraction/interface with the API needed to subscribe/unsubscribe to events and to publish events plus one or multiple implementations based on any inter-process and messaging communication like a messaging queue or Service Bus supporting asynchronous communication and a subs/pubs model.

As introduced in the previous section "Challenges and solutions for Distributed Data Management", you can use events to implement business tasks that span multiple services, and you 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 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 to the microservice architecture, it is 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
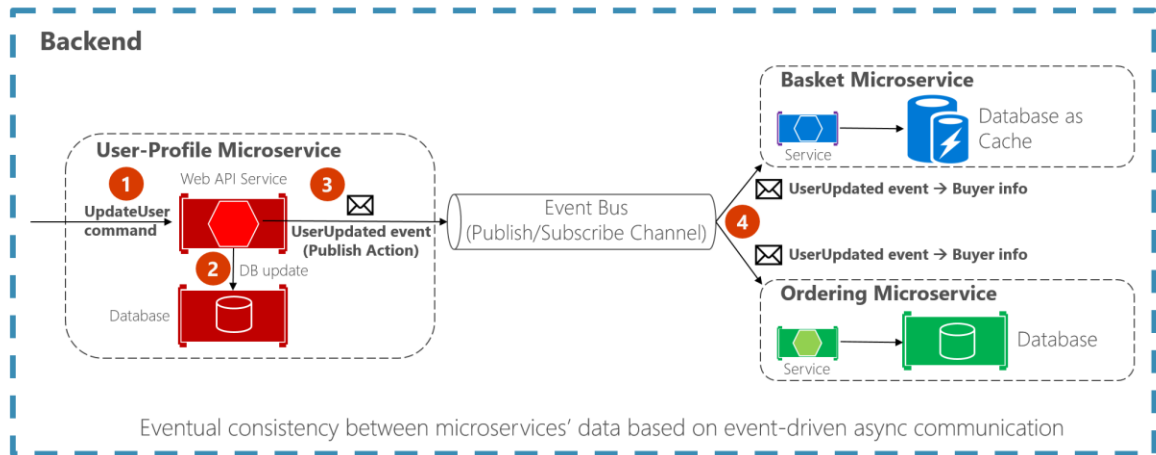
Figure X-XX. Event-Driven and async message communication

In regards the communication protocol to use for event-driven message-based communications, it depends on your implementation. A reliable queued communication could be achieved by using AMQP, but using HTTP asynchronously is also a choice.

What you will probably need is some kind of abstraction level (like an Event-Bus interface) based on a related implementation in classes with code using the API from a service bus like Azure Service Bus with Topics or a queue-based system as RabbitMQ, or even higher level Service Buses like *NServiceBus*, *MassTranssit* or *Brighter*, so you can articulate the mentioned Publish/Subscribe system.

Note on messaging technologies for production systems: Notice that among the multiple messaging technologies you can choose for implementing your abstract Event Bus, some of them can be at a different level than others. For instance, RabbitMQ (messaging broker transport) sits on a lower level than other commercial products like Azure Service Bus, NServiceBus (which can work on top of either RabbitMQ and even on top of Azure Service Bus), or MassTranssit (which can work on top of RabbitMQ). It really depends on how many features and out-of-the-box scalability you need for your application. For implementing just an Event Bus proof of concept for your development environment like in the *eShopOnContainers* sample, a simple implementation on top of "*RabbitMQ running as a container*" might be enough. But for mission critical and production systems needing hyper-scalability you might want to evaluate and use Azure Service Bus. Or for having high level abstractions and features that make distributed applications development easier, other commercial and Open Source service buses like *NServiceBus*, *MassTransit* and *Brighter* are pretty much recommended for you to evaluate. Of course, you could always build more service bus features on top of lower level technologies like RabbitMQ and Docker, but that plumbing work might cost you too much for a custom enterprise application.

Resilient publish to the event bus

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

1. Using a transactional (DTC based) queue as MSMQ (However, this is a legacy approach)
2. Using transaction log mining
3. Using full event sourcing pattern
4. 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.

These approaches are discussed in further details in the implementation sections of this documentation.

References – Publish/subscribe, eventual consistency and other DDD patterns
Event Driven Messaging
http://soapatterns.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
Strategies for Integrating Bounded Contexts
http://culttt.com/2014/11/26/strategies-integrating-bounded-contexts/

Creating and Evolving microservice APIs and Contracts

A microservice API is a contract between the service and its clients. You will only be able to evolve a microservice independently if 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 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 typically can't force all clients to upgrade to your new API contract. You will usually 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. 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 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 can go bad quickly. In the same way that DNS resolves a 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 a later section), service discovery is built-in. For example, within an Azure Container Service environment, Kubernetes and DC/OS with Marathon can handle service instance registration and deregistration. They also run a proxy on each cluster host that plays the role of server-side discovery router. Another example is Azure Service Fabric, which also provides a Service Registry.

References
The Service Registry pattern http://microservices.io/patterns/service-registry.html

Composite UI based on microservices: Including visual UI shape and layout generated by multiple microservices"

Microservices architecture start in the server side handling data and logic. However, a more mature approach is to design your UI based on microservices, as well. That means having a composite UI produced by the microservices themselves instead of having microservices in the server side and just a monolithic client app consuming the microservices. With this approach, the microservices you build can be complete with both logic and visual representation packed together.

On one hand, the image X-XX shows the simpler approach when just consuming microservices from a monolithic client application. You could have, of course a ASP.NET MVC service in between producing the HTML/JS, the diagram below is a simplification highlighting that you have a single client UI (monolithic) which is consuming the microservices which just focus on logic and data, not on the UI shape (HTML/JS).

Monolithic UI *consuming* microservices
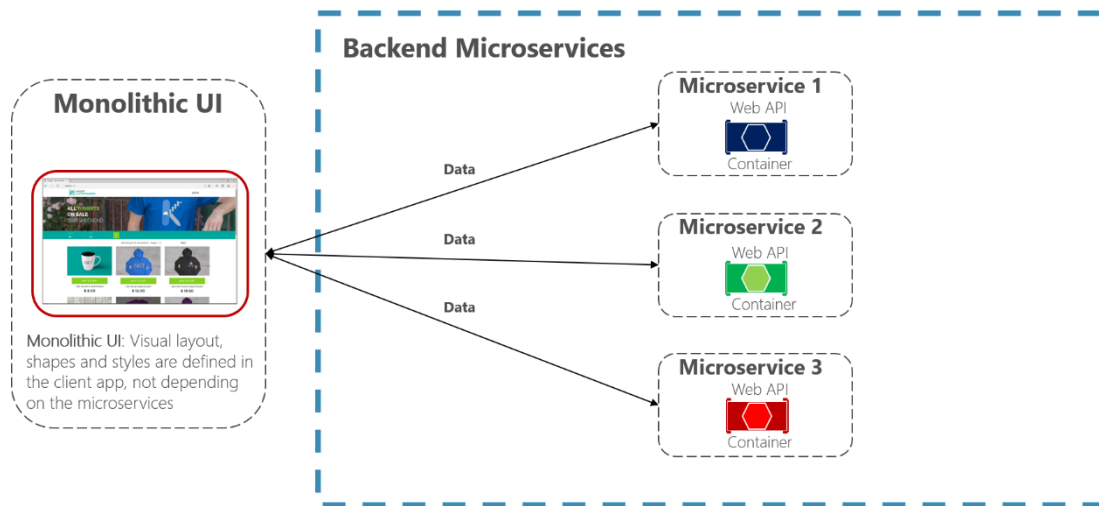
Figure X-X. Monolithic UI application consuming backend microservices

On the other hand, a composite UI is precisely visually-generated and composed by actual microservices, each microservice driving the visual shape of a specific/granular area of the UI.

The key difference is that you will have "client UI components" (TS classes, for example) based on templates, and the "data-shapping-UI" ViewModel for those templates comes from each microservice.

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

The image X-XX shows a simplification as you might have other microservices which might be aggregating the granular parts based on different techniques depending if you are building a traditional web approach (ASP.NET MVC) versus a SPA (Single Page Application).

Composite UI *generated* by microservices
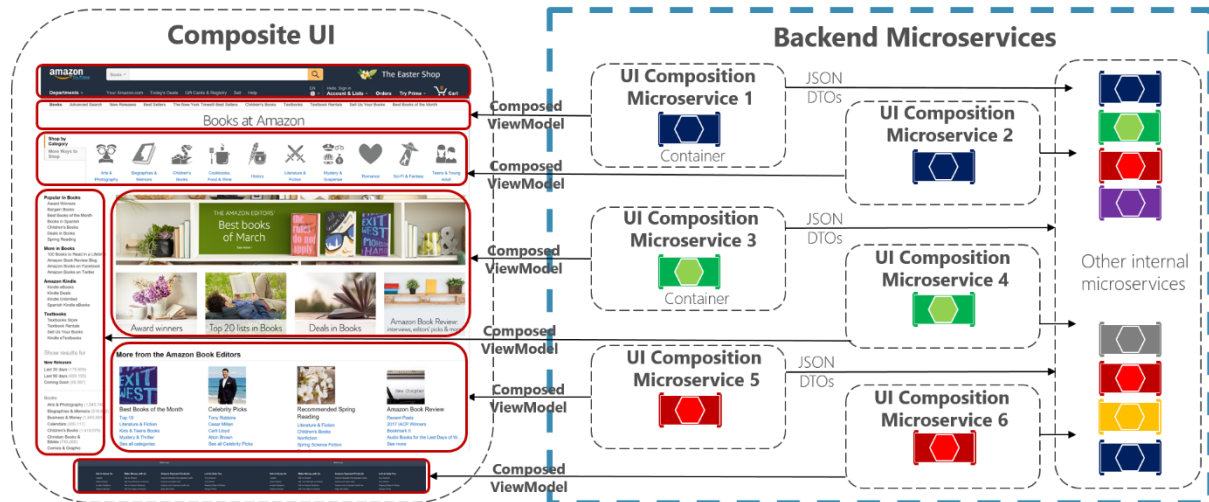
Figure X-X. Composite UI application consuming backend microservices

The composite UI approach driven by the microservices can be more or less challenging depending on what UI technologies you are using. You won't use the same techniques if building a traditional web application versus a SPA (Single Page Application) versus a native mobile apps, like when developing Xamarin apps. The last case is less common.

The *eShopOnContainers* sample microservices-based application is using the monolithic UI approach for multiple reasons. First, it is an introduction to microservices and containers. A composite UI is a further grade of maturity. Second, it is also providing native mobile apps based on Xamarin, which would make it more complex in the client C# side. But, Microsoft encourages you to research further information about composite UI in the following references.

References – Composite UI based on microservices
Composite UI using ASP.NET (Particular's Workshop) http://bit.ly/particular-microservices/ **The Monolithic Frontend in the Microservices Architecture** http://blog.xebia.com/the-monolithic-frontend-in-the-microservices-architecture/ **The secret of better UI composition** https://particular.net/blog/secret-of-better-ui-composition **Including Front-End Web Components Into Microservices** https://technologyconversations.com/2015/08/09/including-front-end-web-components-into-microservices/ **Managing Frontend in the Microservices Architecture** http://allegro.tech/2016/03/Managing-Frontend-in-the-microservices-architecture.html

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 NoSQL 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.
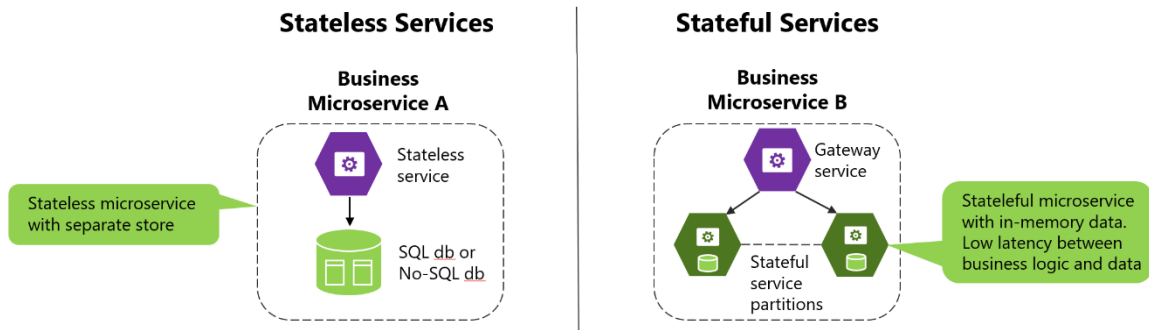
Figure X-XX. Stateless vs. Stateful services

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 trying to improve performance via additional cache and queues. The result is that 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. To get access to those partitions you might need a stateless service acting as a gateway service that knows how to address each partition depending on partition keys.

The drawback in stateful services? - Stateful services impose a level of complexity 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. Currently both frameworks are 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. However, at the time of this writing, Stateful Services in Service Fabric are still not supported as containers but only

on plain microservices. This support (Reliable services support in containers) will come in upcoming versions of Service Fabric.

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 can restart successfully. In other words, there needs to be resilience in the compute (the process restarts) as well as resilience in the state or data (no data loss and the data remains consistent).

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

In addition, resiliency is also related to how cloud-based systems must behave. As previously mentioned, a cloud-based system must embrace failures and need to try to automatically recover from failures. For instance, due to network or container failures, the client apps or client services need to have an strategy in place to retry sending messages or retry requests as in many cases failures in the cloud can be partial. The section named *Implementing Resilient applications* in this book tackles on how to handle partial failure by implementing techniques like retries with exponential backoff or the circuit breaker pattern in .NET Core by using libraries like Polly which offer a large variety of policies to handle this subject.

Health Checks 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.

Health is different from diagnostics. Health is about the microservice reporting its current state to take appropriate actions. A good example is working with upgrade and deployment mechanisms to

maintain availability. Although a service may be currently unhealthy due to a process crash or machine reboot, the service might still be operational. The last thing you need is to make this worse by performing an upgrade. The best approach is to do an investigation first or allow time for the microservice to recover. Health events from a microservice help us make informed decisions and, in effect, help create self-healing services.

In the section named *Implementing Health Checks in ASP.NET Core services* within this book, it is explained how to use a new ASP.NET HealthCheck library within your microservices so they can report their current state to any additional monitoring service to take appropriate actions.

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. It's not just about building your microservice architecture but you will also need, 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

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. Therefore, 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 different in terms of features available from each and their maturity state, sometimes depending on the OS platform.

Microsoft Platform and Tools for Docker

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 that is 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.

A deployment into a cluster of an application composed by multiple microservices/containers would be like the diagram in Figure X-X.
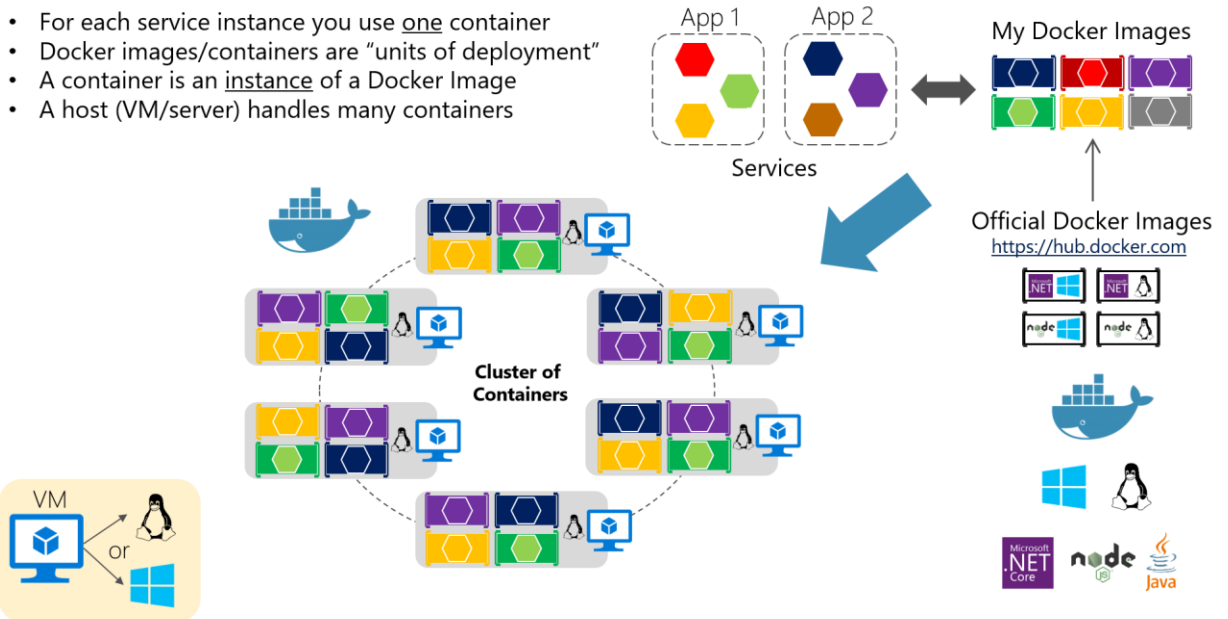
Figure X-X. *Cluster of containers*

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 microservices 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 of 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	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.
	It groups containers that make up an application into logical units for easy management and discovery.
Azure Service Fabric	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.
	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
	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.

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, including Microsoft Azure, Amazon EC2 Container Service, Google Container Engine, and others.

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 an upcoming release. Service Fabric runs on Azure or any other cloud, and also on-premises.

Azure Container Service

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

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.
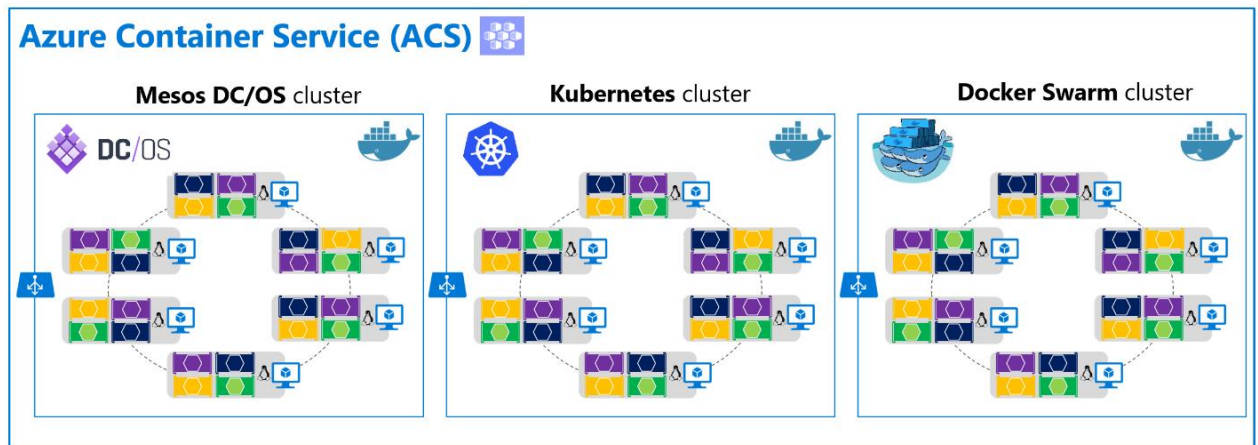
Figure 5-8. Clustering choices in ACS

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 or even tens of thousands of containers.

The Azure Container service enables you to take advantage of the enterprise grade features of Azure while still maintaining application portability, including at the orchestration layers.
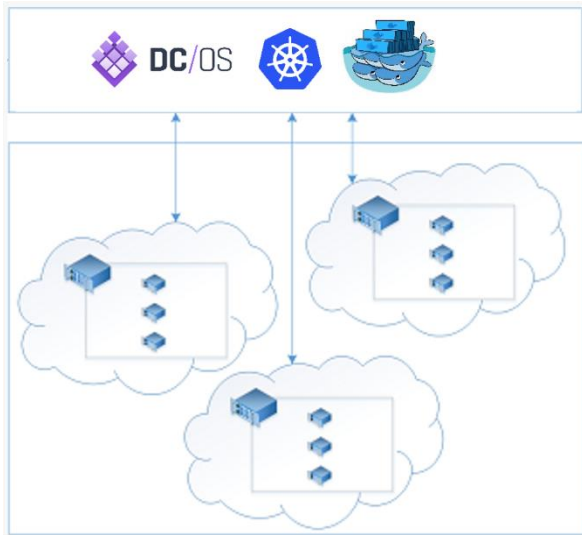
Figure 5-9. Orchestrators in ACS

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

Getting started with Azure Container Service

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

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

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

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

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

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

- A platform that provides system services to deploy, scale-out/scale-in, upgrade, detect, and restart failed services, discover service location, manage state, and monitor health. These system services in effect enable many of the characteristics of microservices described previously.

- Programming APIs, or frameworks, to help you build applications as microservices: reliable actors and reliable services. Of course, you can choose any code to build your microservice, but these APIs make the job more straightforward, and they integrate with the platform at a deeper level. This way, for example, you can get health and diagnostics information, or you can take advantage of reliable state management.

Service Fabric is agnostic to how you build your service, and you can use any technology. However, it does provide 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

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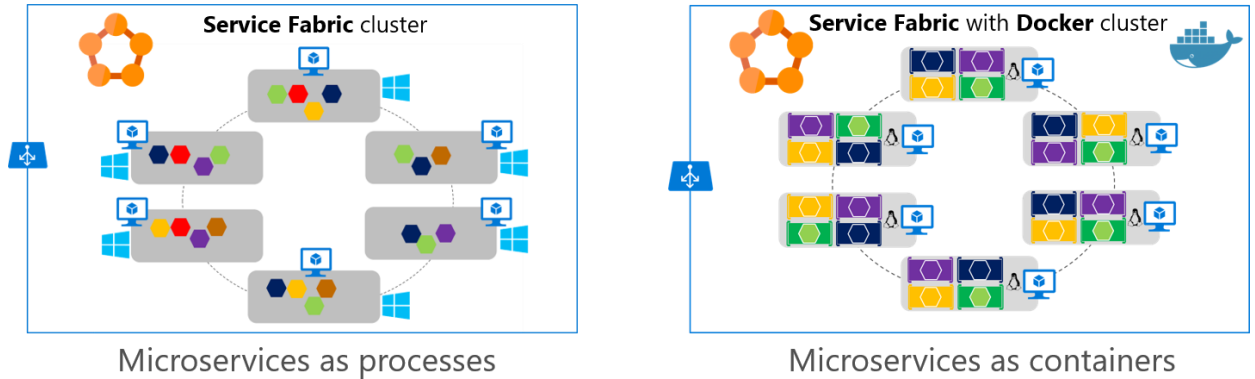
Figure X-X. Deploying microservices as processes or as containers in Azure Service Fabric

Service Fabric clusters based on Linux and Windows hosts can run Docker containers. But, note that as of early 2017, Service Fabric clusters on Windows support just guest containers, meaning that on Windows only regular microservices with no containers can make use of the Reliable Services API. However, in upcoming versions of Azure Service Fabric, you will be able to run either Linux containers or Windows Containers with further access to the Azure Service Fabric infrastructure and APIs.

Development process for Docker based applications

Vision

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

Development environment for Docker apps

Development tools choices: IDE or editor

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

Visual Studio with Docker Tools. If you're using *Visual Studio 2015* you can install the add-in tools "Docker Tools for Visual Studio". If you're using *Visual Studio 2017*, Docker Tools are already installed. In either case 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 most 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 that streamlines the developer workflow. By installing either the "Docker for Mac" or "Docker for Windows" development environment, Docker developers can use a single Docker CLI to build apps for both Windows and Linux. Additionally, Visual Studio Code supports extensions for Docker such as intellisense for Dockerfiles and shortcut-tasks to run Docker commands from the editor.
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**, **.NET Core,** or the OSS project **Mono** when developing Docker containerized .NET applications. You can develop in **C#**, **F#** or **Visual Basic** targeting **Linux** or **Windows** containers, depending on the chosen framework.

Development workflow for Docker apps

The application development lifecycle starts from each developer's machine, coding the app using their preferred language and testing it locally. There is one very important point in common no matter which language, framework, and platform you choose. With this workflow, you are always developing and testing Docker containers, but you are doing so locally.

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

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

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

Workflow for developing Docker container based applications

An app will be composed of your own services plus additional libraries (dependencies).

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

Inner-Loop development workflow for Docker apps
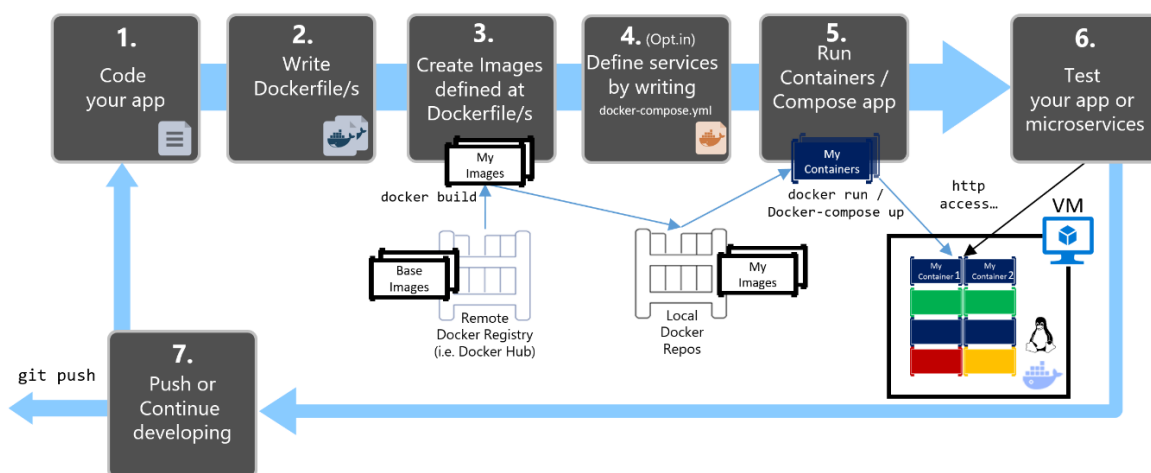
Figure X-XX. Step-by-step workflow developing Docker containerized apps

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 Visual Studio Code and Docker CLI, check the eBook [Containerized Docker Application lifecycle with Microsoft Platforms and Tools](#) for 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* files to your projects with the needed configuration. Visual Studio builds the Docker image and runs the multi-container application directly in Docker after hitting F5, and it even allows you to debug several containers at once. These features will boost your development speed.

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

Visual Studio simplifies that workflow to "the minimum" as explained in the next sections.
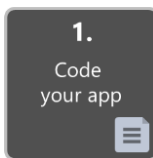
Step 1. Start coding and create your initial app/service baseline

The way you develop your application is similar to the way you would 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 (either 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. The setup is straightforward, as explained in the following reference.

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

In addition, you'll need Visual Studio 2015 with the tools for Docker, or Visual Studio 2017 which includes the tooling for Docker if you selected the ".NET Core and Docker" workload during installation, as shown in Figure x-x.
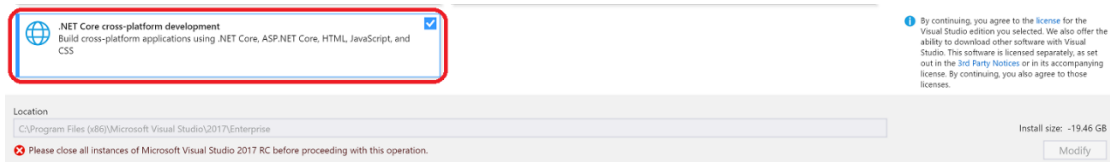
Figure X-X. Selecting the VS workload including Docker Tools

Visual Studio 2017: https://www.visualstudio.com/vs/visual-studio-2017/

Working with .NET and Visual Studio

You can start coding your app in .NET (usually in .NET Core if you are planning to use containers) even before enabling Docker in your app and deploying/testing in Docker. However, it's recommended that you start working on Docker as soon as possible, as that will be the real environment and any issues can be discovered 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 a 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. If your app contains a single custom service, you will need a single dockerfile. If your app contains multiple services (as in a microservices architecture), you'll need one dockerfile per service.

The dockerfile is 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 manually create a dockerfile in code and add it to your project along with your .NET dependencies, however, with Visual Studio and its tools for Docker, it is as simple as a few mouse clicks.

When you create a new project in Visual Studio 2017, there's a new check-box option named "Enable Container (Docker) Support", as highlighted in figure X-X.
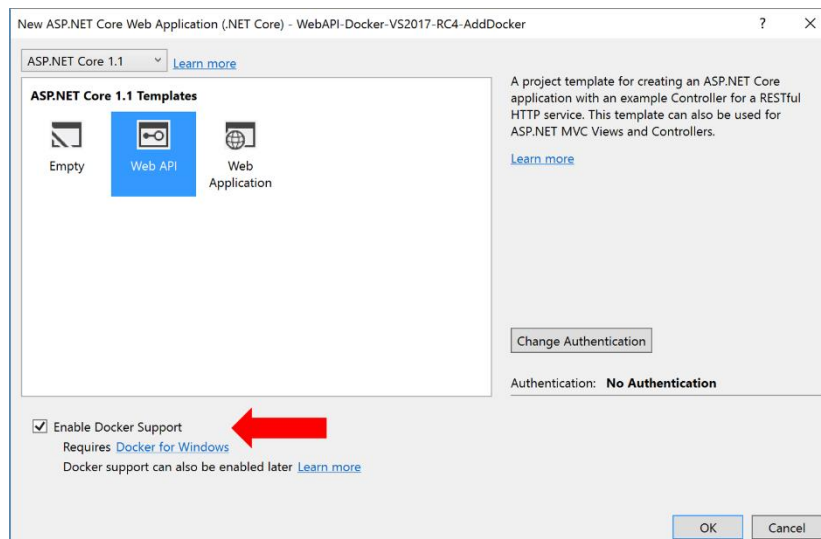
Figure X-X. Enabling Docker Support when creating a new

You can also enable Docker support on a new or existing project by simply right clicking on your project file in Visual Studio and selecting the menu option "Add-Docker Project Support" if your app contains a single project/service or "Add-Docker Solution support" if your app is a multi-container application, as shown in figure X-X.
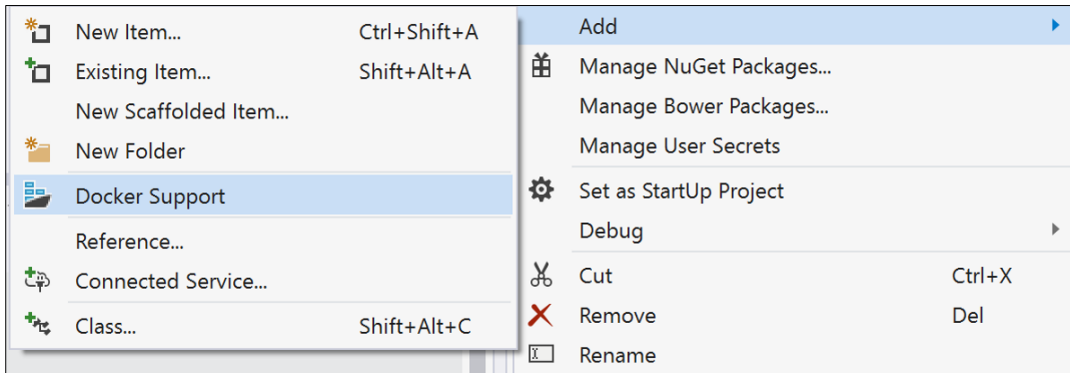
Figure X-XX. Enabling Docker support in a Visual Studio 2017

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 happens under the covers when Visual Studio creates 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.](#) Earlier it was explained which Docker images and repos you can have, depending on the chosen framework and OS. For instance, if you chose to use ASP.NET Core and Linux, the image to 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*, for example, adding "FROM microsoft/aspnetcore:1.1.0" to 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:

```
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, see the [ASP.NET Core Docker Image page](#) and the [.NET Core Docker Image page](#). In the dockerfile, you also need to instruct Docker to listen 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 ENTRYPOINT 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 ENTRYPOINT 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. For example, the microsoft/dotnet repository available in the DockerHub registry provides support for Linux and Windows Nano Server by using the same repo name with different tags, as shown in the following examples.

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

In the future it probably will be possible to use 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 Docker article. This is a scenario that is probably not recommended 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

For each custom service in 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.

Note that the Docker images are built automatically for you in Visual Studio. The following steps are only needed for the Editor/CLI workflow.

Development process for Docker apps

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

To create a custom image in your local environment by using Docker CLI and your *dockerfile*, you can use the `docker build` command, as in the following example.

Optionally, instead of directly running `docker build` from the project's folder, you can first generate a deployable folder with the needed .NET libraries and binaries with `run dotnet publish`, and then use the `docker build` command:

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

Figure X-XX. *Creating a custom Docker*

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

When an application is made by multiple containers (multi-container app) you can also use the `docker-compose up --build` command so it builds all the related images with a single command by using the metadata exposed at the related docker-compose.yml files.

You can find the existing images in your local repository (on your dev machine) by using the `docker images` command.

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

Figure X-XX. *Viewing existing images*

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 when you press F5 and run the *dockerized* application or service. This step is transparent when working in Visual Studio, but it's important that you know what's going on underneath.

4. (opt)
Define services by writing docker-compose.yml

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

The `docker-compose.yml` file lets you define a set of related services to be deployed as a composed application with the deployment commands explained in the following section.

You need to create the file in your main or root solution folder, with content similar to that shown in figure <mark>X-XX</mark>.

```
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 <mark>X-XX</mark>. *Example "docker-compose.yml" file for a multi-container based app*

Note that the docker-compose.yml of the previous example is a simplified version which contains static configuration data for each container that always applies plus configuration that might depend from the deployment environment, like the connection string. This is a simplification for the sake of simplicity, but in later sections you will see how you can split the docker-compose.yml configuration in multiple files and override values depending on de environment and execution type (debug/release).

In the docker-compose.yml file example, it defines five services. The webmvc service (a web app), two microservices (ordering.api and basket.api) and two data source containers, ordering.data based on SQL Server for Linux running as a container, and 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 the webmvc service:

Development process for Docker apps

- Uses the pre-built eshop/web:latest image.
- Uses two environment variables initialized in this file.
- Forwards the exposed port 80 on the container to port 800 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 the docker-compose.yml file in a later section covering microservices and multi-container apps.

Working with docker-compose.yml in Visual Studio 2017

When you add **Docker Solution Support** to a service project in your solution, Visual Studio adds a dockerfile file to your project plus it also addas a service section/project in your solution with the docker-compose.yml files. It is an easy way to start composing your multiple-container solution, and you can then open the docker-compose.yml files and update them with additional features.
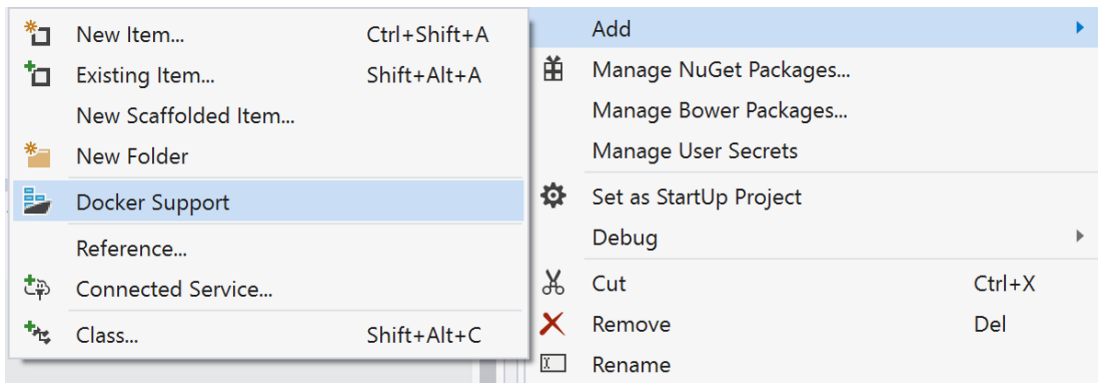
Figure X-XX. Enabling Docker Solution support in a Visual Studio

This action will not only add the dockerfile to your project, but it will also add the required configuration lines of code to a global docker-compose.yml set at the solution level.

After adding Docker support to your solution in Visual Studio, you will also see a new node tree at the solution Explorer with the added docker-compose.yml files.
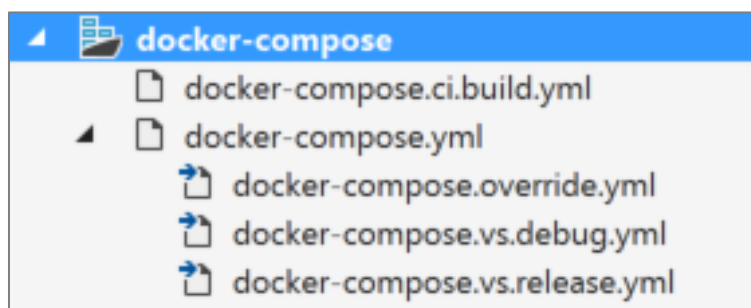
Figure X-XX. Docker-compose tree node in Visual Studio 2017 Solution

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

Development process for Docker apps

Step 5. Build and run your Docker app

If your app only has a single container, you can run it by deploying it to your Docker Host (VM or physical server). However, if your app contains multiple services, you need to *compose* it, too. Let's look at 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`

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

Note that for this deployment, we're redirecting requests sent to port 80 to the internal port 5000. This means that the application is listening on the external port 80 at the host level.

Option B. Running a multi-container application

In most enterprise scenarios, a Docker application will be composed of multiple services, which means you need to run a multi-container application as shown in figure 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. The following example shows the results when running the command from your main project directory containing the docker-compose.yml file.

Figure X-XX. VM with Docker

Figure X-XX. Example results when running the "docker-compose up" command

After running `docker-compose up`, the application and its related containers deployed into your Docker Host, as illustrated in the VM representation in Figure X-XX.

Running and debugging a multi-container application with Visual Studio

Again, when using Visual Studio 2017 it cannot get simpler. You are not only running the multi-container application, but you're able to debug all 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. Visual Studio will spin up a container per project that has Docker solution Support enabled, 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 X-XX, in Visual Studio 2017 there is an additional **Docker** command under the F5 command button. You can run or debug a multiple container application by running all the containers that are defined in the docker-compose.yml file at the solution level. The file was modified by Visual Studio while adding Docker Solution Support to each of your projects. This means that you could set several breakpoints, each breakpoint in a different project/container, and while debugging from Visual Studio you will be stopping at breakpoints defined in different projects and running on different containers.
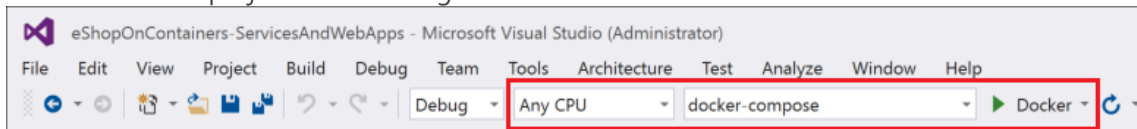
Figure X-XX. Running multi-container apps in Visual Studio 2017

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

> **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: The `docker-compose up` and `docker run` commands (or running/debugging the containers in Visual Studio) might be adequate 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 of 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.

For DC/OS and Kubernetes you would use different deployment commands and scripts as well.
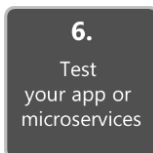
Step 6. Test your Docker application (locally, in your local Docker Host)

This step will vary depending on what your app is 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.
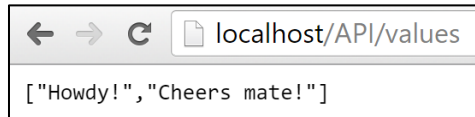
Figure 5-22. Example of testing your Docker application locally using localhost

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 the `docker run` command, as explained in a previous step.

It can also be tested with CURL from the terminal, as shown in figure 5-23. In a Docker installation on Windows, the default IP is 10.0.75.1.

Figure 5-23. Example of testing your Docker application locally using CURL

Testing and Debugging containers with Visual Studio

When running and debugging the containers with Visual Studio, you'll be able to debug the .NET application running on containers in much the same way as you would when running on the plain OS.

For further details on how to debug containers, read the following article:

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/

Testing and Debugging without Visual Studio

When developing without Visual Studio, just using the CLI and maybe a code editor like Visual Studio code, debugging is more difficult.

Container debugging for VS Code is out-of-scope for this doc and when using VS Code, users will want to debug outside of a container whenever possible.

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 Visual Studio, as shown in the image X-XX.
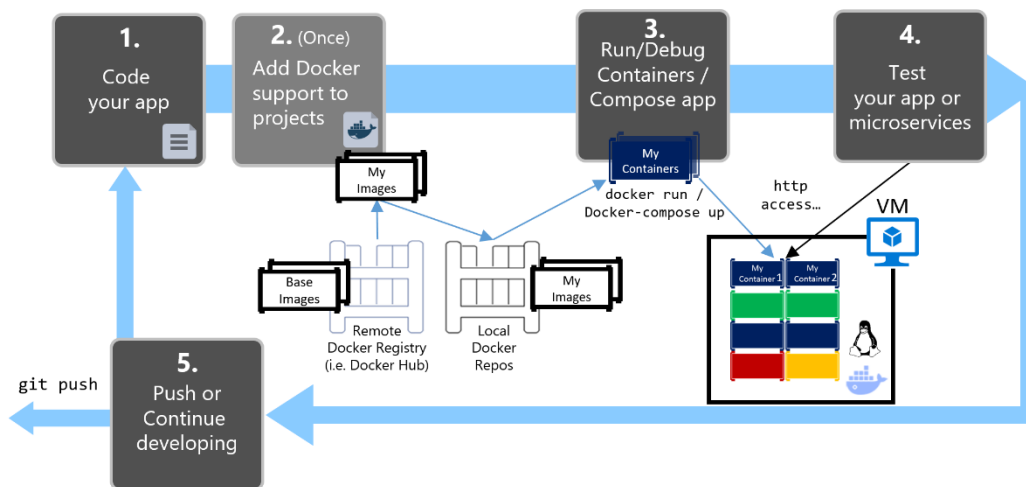
Figure X-XX. Simplified workflow when developing with Visual

Even further, step number 2, "Add Docker support to your projects" needs to be done just once. So usually that process or workflow remains 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 most of the work, it'll be greatly simplified by using Visual Studio, making you a lot more productive.

Using PowerShell commands in a dockerfile to set up Windows Containers

Windows Containers allow you to convert your existing Windows applications into Docker images and deploy them with the same tools as the rest of the Docker ecosystem.

To use Windows Containers, you just need to run PowerShell commands in the *dockerfile*, as in 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, we are using a Windows Server Core base image, also installing IIS with a PowerShell command. In a similar way, you could also use PowerShell commands to set up additional components like ASP.NET 4.x, .NET 4.6, or any other Windows software. For example: `RUN powershell add-windowsfeature web-asp-net45`

Developing and deploying new single-container based .NET Core web applications for Linux or Windows Nano containers

Vision

You can use Docker containers for simpler web applications with monolithic deployment with the objective of improve your CI/CD pipelines and deployment to production success. No more "it works in my machine, why does it not work in production?"

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

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

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

Also, early in the development of an application you may not have a clear idea where the natural functional boundaries are. As you develop a Minimal Viable Product, the natural separation may not yet have emerged.

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

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

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

The first version of *eShopOnContainers* (named *eShopWeb*), running as a monolithic MVC application, gains benefits from the simplicity offered by that design choice. But, even this monolithic application gains benefits from being deployed in a container environment.

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

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

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

Application Tour

The *eShopWeb* application represents some of the eShopOnContainers application running as a monolithic application. It provides the catalog browsing capabilities you've seen earlier. It's an ASP.NET Core MVC based application running on .NET Core.

The application uses an SQL database for the catalog storage. In container based deployments, this monolithic application can access the same data store as the microservices based application. You'll want to configure the SQL server to run in a container alongside of the monolithic application. In a production environment, the SQL server would run on a high availability machine, outside of the Docker host. For convenience in a dev/test environment, we are running the SQL server in its own container.

The initial feature set enables browsing the catalog. Updates would enable the full feature set of the containerized application. The full web application and its architecture will be described in an upcoming [ASP.NET Web Application architecture practices eBook](#) eBook and related [sample app](#), although the simplified version available at [eShopOnContainers](#) is running on a Docker containers as an additional differenciator factor.

Docker Support

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

As before, you could get started by right-clicking on the project, and selection "Add", followed by "Docker Support". The template creates a Docker-compose project that provides a started docker-compose.yml file, along with a Dockerfile for the main project. In *eShopWeb* (monolithic apps) within *eShopOnContainers* you don't have to do that as it is already "Docker enabled".

The Docker-compose project contains six files, including the .dcproj file for the docker project:
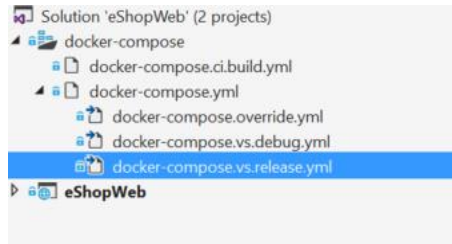
Figure X-XX. docker-compsoe project in a single container web app

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

The docker-compose.yml file contains the information about what images to build and what containers to launch. The templates knew to build the eshopweb image and launch that containers. You need to add the dependency on the sql image, and a service for the sql image for Docker to build and launch that container. See the last four lines on the docker-compose file below:

```
version: '2'

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

The depends_on: directive tells Docker that the eshopweb image depends on the sql.data image. Below that are the instructions to build an image tagged 'sql.data' using the 'microsoft/mssql-server-linux' image.

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

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

The .vs.debug YML file contains settings used for debugging in Visual Studio. Note that the eshopweb image has the 'dev' tag appended to it. That helps separate debug from release images so that you don't accidentally deploy the debug information to a production environment.

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

```
version: '2'

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

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

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

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

Troubleshooting

Stopping Docker containers.

After launching the containerized application, the containers continue to run, even after you've stopped debugging. You can run 'docker ps' from the command line to see which containers are running. The command 'docker stop' will stop a running container (See image)

Figure X-XX. Listing and stopping containers with "docker ps" and "docker stop" CLI

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

Adding Docker to your projects

The wizard that adds docker support communicates with the running Docker process. You'll need to have Docker running to add Docker support to a project. Also, the wizard selects the container platform based on the current Docker configuration: if you want to run in Windows containers, you need to have Docker configured for Windows containers. Similarly, if you want to run in Linux containers, you need to have Docker configured for Linux containers.

Conclusion

In this section, you've seen how to run a monolithic application in a Docker container. More importantly, you've learned to think about when to create a monolithic application instead of creating a microservices based application. Depending on the problem to be solved, and the scope of the application, you can best determine which plan works correctly for you. Remember that choosing to containerize a monolithic application does not preclude moving toward a microservices architecture later. Finally, you can see from the tour of the application how a single monolithic application may later grow and benefit from the greater separation that a microservices based application provides. You will find this path easier if you follow the design recommendations you saw in the tour of the application. Visual Studio tools makes either path easier.

Migrating and deploying legacy monolithic .NET Framework applications to Windows containers

Problem Statement

Earlier sections of this document have championed a microservices architecture where business applications are distributed among different containers each running small, focused services. That goal has many benefits. In new development, it's strongly recommended. Enterprise critical applications will also gain enough benefits to justify the cost of a re-architecture and re-implementation. But not every application will gain enough benefits to justify the cost. That doesn't mean they can't be used in container scenarios, or gain some important benefits.

In this section, we'll explore an internal application for eShopOnContainers, shown in figure X-XX.
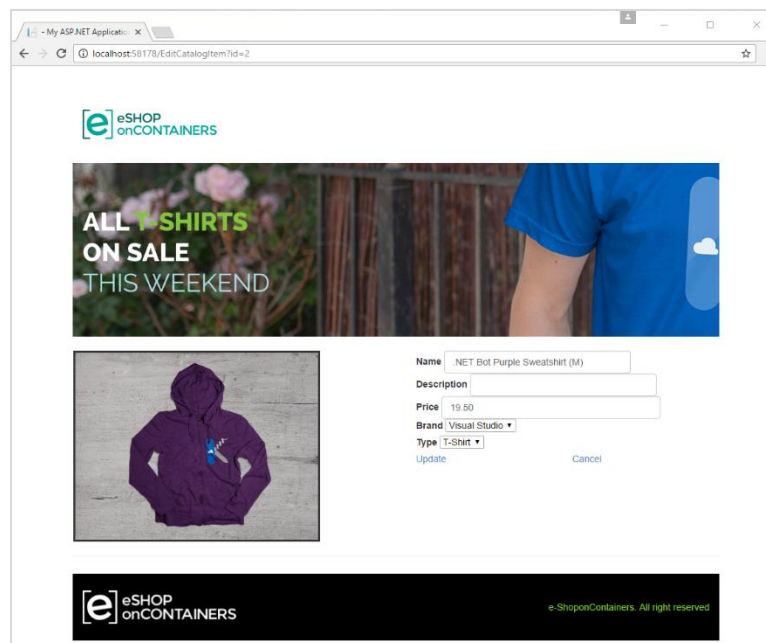
Figure X-XX. Catalog management web app page implemented with ASP.NET Web Forms (Legacy

It's a WebForms application used to browse and modify the catalog entries. The WebForms dependency means this application won't run on .NET Core, without being re-written on ASP.NET Core MVC. You'll see how you can run applications like these in containers without changes. You'll also see how you can make minimal changes to enable them to access some of the services that have been moved to a microservices architecture.

Benefits

The Catalog.WebForms application worked fine as a stand-alone web application accessing a high availability data store. Even so, there are benefits gained by running the application in a container. You create an image for the application. From that point forward, every deployment will be running in the same environment. Every container will use the same OS version, have the same version of dependencies installed, be using the same framework, and have been built using the same process.

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

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

Scenarios like these benefit from a "lift and shift" operation to enable running these applications in a Docker container. The phrase "lift and shift" describes the scope of the task: You *lift* the entire application from a physical or virtual machine, and *shift* it into a container. In ideal situations, you don't need to make any changes to the application code to run it in a container.

Path

The original application was self-contained, including data access libraries. The database ran on a separate high availability machine. That configuration is simulated in the sample code by using a mock catalog service: you can run the catalog.WebForms application against that fake data to simulate a pure 'lift and shift' scenario. This demonstrates the simplest path where you move existing assets to run in a container without the smallest amount of code changes. This path is appropriate for applications that are "done", and have minimal interaction with functionality that you are moving to microservices.

You'll also see a path where applications that you've migrated to a Windows based container can gain some benefits from a small refactoring. The catalog.WebForms application accesses the catalog data. A small set of code changes enables the WebForms application to access the catalog through the microservice. These changes demonstrate the continuum you work with for your own applications. You can do anything from moving an existing application into containers, to making small changes that enable existing applications to access some of the new microservices to completely rewriting an application to fully participate in a new microservice based architecture. The right path depends on both the cost of the migration and the benefits from any migration.

Application Tour

You can load the Catalog.WebForms solution and run the application as a standalone app. In this configuration, the startup code configures the DI container to use the fake catalog service. Run the application and you'll see the WebForms application displaying the catalog data.
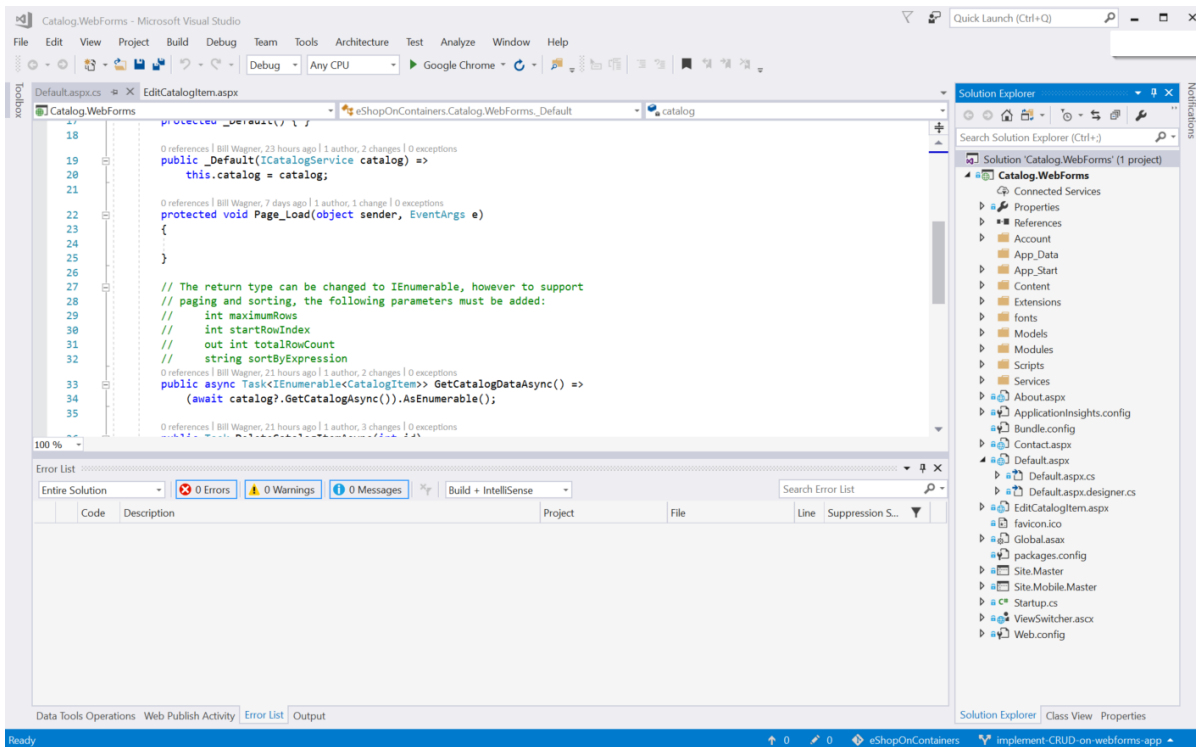
Figure X-XX. Catalog management Web Forms app in Visual Studio 2017

Most of the techniques used in this application should be very familiar to anyone that has used WebForms. The use of the catalog microservice has introduced two techniques that might be unfamiliar: dependency injrection, and working with asynchronous data stores in WebForms.

Dependency Injection inverts the typical object oriented strategy of writing classes that allocate all needed resources. Dependency injection means that classes request their dependencies from a service container. The advantages of Dependency Injection is that you can replace external services with fakes or mocks to support testing or other environments.

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

```csharp
private static IContainer CreateContainer()
{
    // Configure AutoFac:
    // Register Containers:
    var settings = WebConfigurationManager.AppSettings;
    var useFake = settings["usefake"];
    bool fake = useFake == "true";
```

```
    var builder = new ContainerBuilder();
    if (fake)
    {
        builder.RegisterType<CatalogMockService>()
            .As<ICatalogService>();
    }
    else
    {
        builder.RegisterType<CatalogMockService>()
            .As<ICatalogService>();
    }
    var container = builder.Build();
    return container;
}


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

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

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

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

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

    }
}
```

The constructors for the pages (Default.aspx.cs and EditPage.aspx.cs) define constructors that take these dependencies. The module calls those constructors in its InjectDependencies method:

```
protected _Default() { }

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

Note that the default constructor is still present and accessible. The infrastructure needs this.

The catalog APIs are all asynchronous methods. Webforms now supports these for all data controls. The Catalog.WebForms application uses model binding for the list and edit pages, and those controls define SelectMehods, UpdateMethods, InsertMethods and DeleteMethods that are Task-returning asynchronous operations. WebForms controls understand when the methods bound to a control are asynchronous. The only restriction you encounter when using asynchronous Select methods is that you cannot support paging: the paging signature requires an out parameter. Async methods cannot have out parameters. This same technique is used on other pages that require data from the catalog service.

The default configuration for the catalog web forms application uses a mock implementation of the catlog.api service. This mock users a hard coded dataset for its data. This simplifies some tasks by removing the dependency on the catalog.api service in development environments.

Lifting and Shifting

Visual Studio provides great support for containerizing an application. You right-click on the project node, and select "Add", and "Docker Support". This template adds a new project to the solution called "docker-compose". This project contains the Docker assets that compose the images you need, and start the necessary containers. In the simplest lift and shift scenarios, this will be the single service that you use for the web forms application. The template has also changes your startup project to point to the docker-compose project. That means Ctrl+F5 and F5 will now create the Docker image and launch the Docker container.
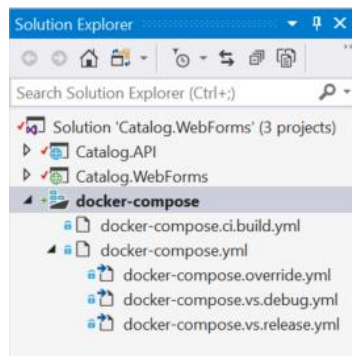
Figure X-XX. The docker-compose project in the Web Forms solution

Before you press Ctrl-F5, make sure you configure Docker to use Windows containers. Right click on the Docker taskbar icon in Windows and select "Switch to Windows containers":
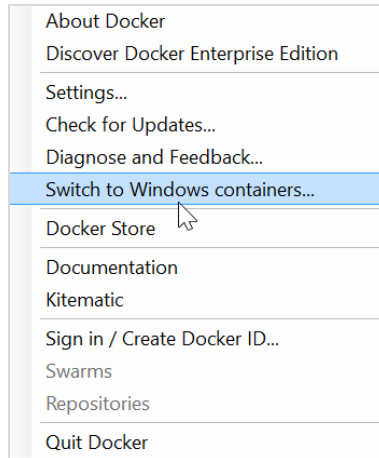
Figure <mark>X-XX</mark>. Switching to Windows Containers from Docker taskbar icon in

If the text says "Switch to Linux containers", you're already running Docker with Windows containers.

This will restart Docker. When you build, you'll build the application, and the Docker image for the WebForms project. The first time you do this, it takes considerable time. You'll pull down the base Windows Server image and the additional image for ASP.NET. Subsequent build and run cycles will be much faster.

The wizard creates several files for you. Visual studio uses these files to create the Docker image and launch a container. You can also use those same files from the CLI to run Docker commands manually.

This Dockerfile shows the basics for building a Docker image based on the Windows ASP.NET image and built to run an ASP.NET site:

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

The most important difference here is that the base image is "microsoft/aspnet", which is the current Windows server image that includes the .NET framework. Other differences are that the directories copied from your source directory are different.

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

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

Using the existing "Catalog" .NET Core Microservice

You can reconfigure the webforms application to use the *eShopOnContainers* catalog microservice instead of the fake data. Edit the web.config file and set the value of the 'useFake' key to false. The DI container will use the class that access the live catalog microservice instead of the class that returns the hard-coded data. No other code changes are needed.

Accessing the live catalog service does mean you need to update the docker-compose project to build the catalog service image and launch the catalog service container. Docker for Windows supports both linux containers and Windows containers, but not at the same time. So, in order to run the catalog microservice, you need to build an image that runs it on top of a windows based container. That requires a different Dockerfile for the microservices project than you've seen in earlier sections. The Dockerfile.windows contains the commands (like to use a Windows Nano Docker image) to build the catalog API container image so that it runs on a Windows container.

The catalog microservice relies on the SQL server database. You'll also need to use a Windows based SQL server Docker image, as well.

The docker-compose file now builds a new windows-based image to run the SQL database and the catalog microservice using the new Dockerfiles that build Windows based images. Then, it launches an instance of all three containers in the same Docker host. Using this configuration, you are running a Docker host with all three services: the SQL data store, the catalog microservice, and the Web Forms application.

Development and Production Environments

There are a couple differences between this configuration and a production configuration. In the development environment, you'll be running the WebForms application, the catalog microservice and the SQL database in Windows containers, as part of the same Docker Host. In earlier sections, you've seen them deployed in the same Docker host as the other .NET core based services on a Linux based Docker host. The advantage of running the multiple microservices in the same Docker host (or cluster) is that you get lower latency from the network communication.

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

Conclusion

The Lift and Shift scenario provides you with the benefits of moving to containerized deployments for existing .NET applications. You can run applications that have taken dependencies on Windows OS

features. You can run applications that rely on features in the .NET Framework that are not available in .NET Core.

This scenario can be the correct long term solution for some applications. In others, it may be a short-term solution on the path to a more complete migration to microservices. The possible benefits can continue to be measured, and the cost of a further migration can be estimated to determine when or if further investment is justified.

Designing and developing multi-container and microservice based .NET applications

Vision

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

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

However, if you know how to design and develop a microservice architecture based application 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. For example, you might design 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 world, this section focuses on a microservice architecture implementation using Docker containers.

Designing a microservice oriented application

Application context

This section focuses on developing a hypothetical server-side enterprise application. It must support a variety of different clients including desktop browsers running SPA (Single Page Applications), traditional web apps, mobile web apps and native mobile apps. The application might also expose an API for 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, or XML responses.

The application will consist of multiple types of components:

- Presentation components - responsible for handling the UI and consuming remote services.
- Domain/business logic - the application's domain logic.
- Database access logic - data access components responsible for accessing databases (SQL or NoSQL).
- Application integration logic - messaging layer, possible service buses, etc.

The application will have requisites of high scalability, but probably, certain sub-systems will require 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. 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 cohesive and narrowly related functions. For example, an application might consist of services such as the catalog service, ordering service, basket service, user profile service, etc.

Microservices communicate using protocols such as HTTP/REST, asynchronously whenever possible, especially when propagating changes/updates.

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

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

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

So you can focus on the architecture and technologies instead of thinking about the business domain, we have selected a simplified ecommerce or e-shop application that presents a catalog of products, takes orders from customers, verifies inventory, and other business features. This container-based application's source code is available on 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 –Direct Client-to-Microservice Communication and Event

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 the `docker-compose` up command. However, if using an orchestrator or container-cluster, each container

could be running in a different host/node and any node could be running any number of containers, as 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 – eShopOncontainers uses two main communication types depending on the kind of the functional action (queries vs. updates/transactions):

1. Direct Client-to-Microservice communication between client apps and microservices. This is mostly used for queries and when accepting update/transctional commands from the client apps.

2. Asynchronous event based communication through an Event Bus in order to propagate updates/states across microservices or external application integration. The Event Bus could be implemented with any messaging broker infrastructure technology, like RabbitMQ or higher level Service Buses like Azure Service Bus, NServiceBus, MassTranssit, Brighter, etc.

The application is 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. Note 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.companyname, 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 in the eShopOnContainers application where the goal is to focus on the microservices deployed as Docker containers.

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

Data Sovereignty Per Microservice

In terms of 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 was made only because this application is a sample reference application, and any developer should be able to just grab the code from GitHub, clone it, open it in Visual Studio or Visual Studio Code. You can also compile the custom Docker images using .NET Core CLI and Docker CLI, and then deploy and run it in a Docker development environment. This can be accomplished in a matter of minutes without having to provision an external database or any other data source with hard dependencies on infrastructure (cloud or on-premises). However, consider that in a real production environment, for high availability and scalability reasons, the databases should be based on database servers in the cloud or on-premises, but not in containers.

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

Benefits

A microservice based solution like this has many 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.
  - The IDE is faster for 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 might need to scale-out more than the ordering process. The resulting infrastructure will be much more efficient in regards to the resources used when scaling out.

- **It enables you to organize the development effort around multiple teams**. Each service can be owned by a single dev team. Each team can develop, deploy and scale their service independently of all the other teams.

- **Improved issues isolation**. For example, if there is a bug or issue in one service then only that service will initially be 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, for example with memory leaks. Additionally, when the bug or issue is resolved, you can deploy just the affected microservice without impacting the rest of the already running microservices.

- **You can use the latest technologies**. Because you can start developing autonomous services independently and run them side-by-side, you can start using the latest technologies and frameworks instead of being stuck on an older stack or framework for the whole application.

Drawbacks

A microservice based solution like this also has many possible drawbacks:

- **Distributed system**. This adds complexity that must be handled by developers when designing and building the applications.

  - Developers must implement inter-service communications, which adds complexity in regards to testing and exception handling. It also adds 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 you are not using a

microservice oriented infrastructure (like an orchestrator or scheduler) that additional complexity can require more development efforts than the business application itself.

- **Atomic transactions.** Atomic transactions between multiple microservices usually aren't 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 N times M services instances (i.e. 8 microservices per application instance). If each service runs in its own .NET Core framework, which is preferred 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 general and the benefit of being able to scale-out just certain areas of the application compared to long-term costs when evolving monolithic applications, 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 or screen. While a client could make that many requests, it would probably be too inefficient over the public Internet and would be impractical over a mobile network, so requests from the client app to the backend system should be minimized.

    o 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 are best used internally. An application should use protocols such as HTTP and WebSocket for communication outside of the firewall.

    o 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 want to consider the API Gateway pattern instead of the simpler Direct Client-to-Microservice communication approach.

Finally, another challenge no matter which approach you take for your microservice architecture is deciding how to partition the system into microservices. This is very much an art, but there are several 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 partitioning services by use case. For example, in our e-Shop application we have the ordering service that is responsible for all of the business logic related to the order process. 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. This is similar to the Single Responsibility Principle (SRP) applied to classes, which states that a class should only have one reason to change. In this case it is about microservices, so the scope might be a bit larger than a single class, and most of all it has to be completely autonomous, end to end, including responsibility for 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, following the 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 these could even use different technologies and programming languages as illustrated in figure ==X-XX==.
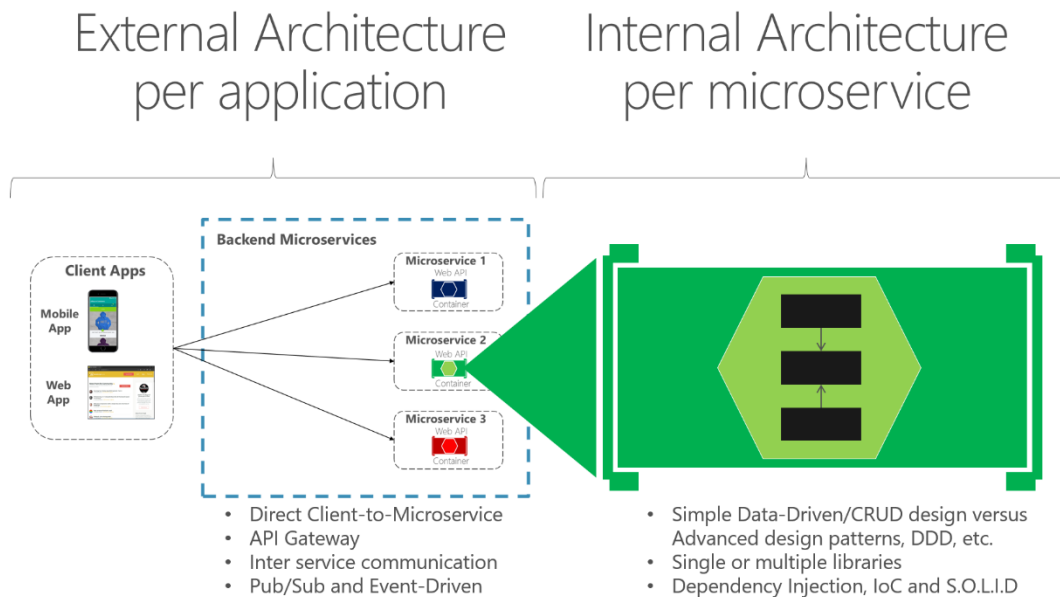
Figure ==X-XX==. External vs. Internal Architecture and Design

For instance, in our initial eShop sample system, the catalog, basket and user profile microservices are simple and basically CRUD sub-systems, therefore, their internal architecture and design is straightforward. However, you might have other microservices, in this case the Ordering microservice, which has further complexity and ever-changing business rules with a high degree of domain/business complexity. In such cases, you might want to implement more advanced patterns within a particular microservice, like the ones defined with Domain-Driven Design approaches, as we are 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 might be related to the nature of the microservice. For certain domain logic, it might be a better implementation if you use a functional programming language such as F#, or even a language like R when targeting AI and machine learning domains, instead of a more object-oriented programming language like C#.

The bottom line is that each microservice can have a different internal architecture and different design patterns. Not all microservices should be implemented using advanced DDD patterns as that would be over engineered, 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 low quality spaghetti code.

The new world: multi Architectural Patterns and polyglot microservices

There are many architectural patterns used by software architects and developers. The following are a few typical architectural patterns that can be implemented:

- Simple CRUD, single Tier, single Layer
- Traditional NLayered
- Domain-Driven Design (DDD) patterns
- Command and Query Responsibility Segregation (CQRS) architectural patterns
- Event-Driven patterns
- Etc.

You can also build microservices with many technologies and languages, like ASP.NET Core Web APIs, NancyFx, ASP.NET Core SignalR (.NET Core 2 timeframe), F#, Node.js, Python, Java. C++, GoLang, Etc.

The important point is that no particular architecture pattern or style, nor any particular technology is right for all situations.

Figure X-XX. Multi-architectural-patterns and polyglot microservices

As shown in figure X-XX, (almost a random order, approaches and technologies could vary) when building large composite applications composed by many microservices (bounded contexts in Domain-Driven Design lingo, or simply call it subsystems in the form of autonomous microservices) you should implement each microservice in a different way with a different architecture patterns and even languages and databases depending on its "nature", business requirements and priorities.

Sometimes all of them could be pretty similar, but usually that shouldn't be the case as each subsystem's context boundary and their requirements are usually different.

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

Even more, when dealing with large composited applications, you shouldn't apply a single top-level architecture based on a single architecture pattern approach. For instance, CQRS shouldn't be applied as a top level architecture for a whole application, but might be useful for a single or a specific set of microservices.

There is no "silver bullet" or a unique "right architecture pattern" for every given case. You cannot have "a single architecture pattern to rule all the microservices or bounded contexts in your system". Depending on the priorities of each microservice, you must choose a different approach.

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 could be as simple as possible, for whatever reason. Might be because the problem to solve is extremely simple or because it is a proof of concept.
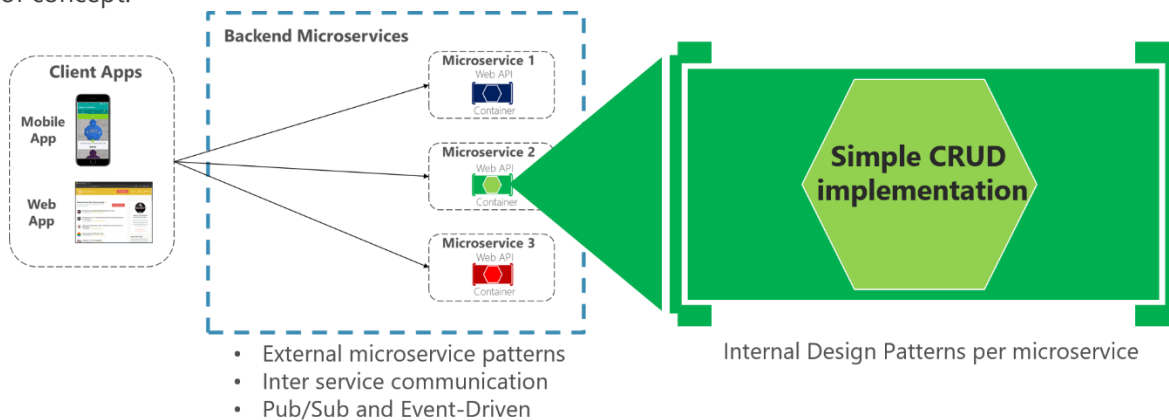
Figure X-XX. Internal Design for simpler CRUD

An example of this kind of service is the Catalog microservice from the eShopOnContainers sample application which is a very simple data-driven Catalog. 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 could have its related data and database running in a SQL Server container as shown in the design diagram in the next figure X-X.

Data-Driven/CRUD microservice container
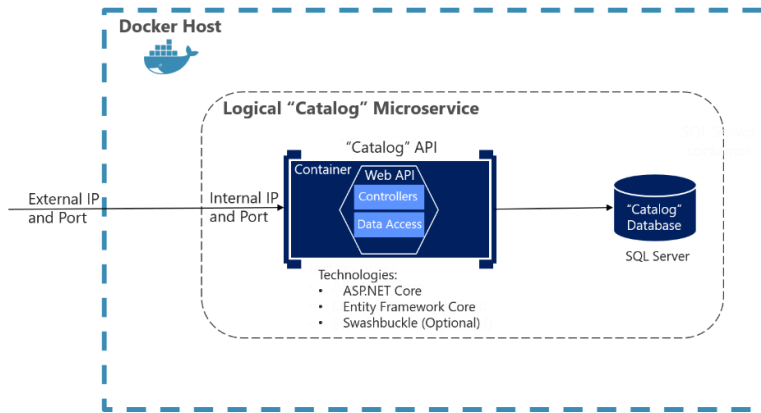
Figure X-XX. Simple data-driven/CRUD microservice design

When developing this API you only 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 to provide a description of what your service offers, as explained in the next section.

Note that running a database server like SQL Server within a Docker container is great for development environments as you can have all your dependencies up and running without needing to provision a database in the cloud or on-premises. This is very convenient when running integration tests. However, for production environments running a database server in a container is not 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 High Availability and High Scalability. For example, you might choose DocumentDB when using a NoSQL approach.

Finally, by editing the *dockerfile* and *docker-compose.yml* metadata files you can configure how the image of this container will be created and what base image it will use, plus design settings such as 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 start by creating a simple ASP.NET Core Web API project (running on .NET Core so it can run on a Linux Docker host), as shown in figure X-X.
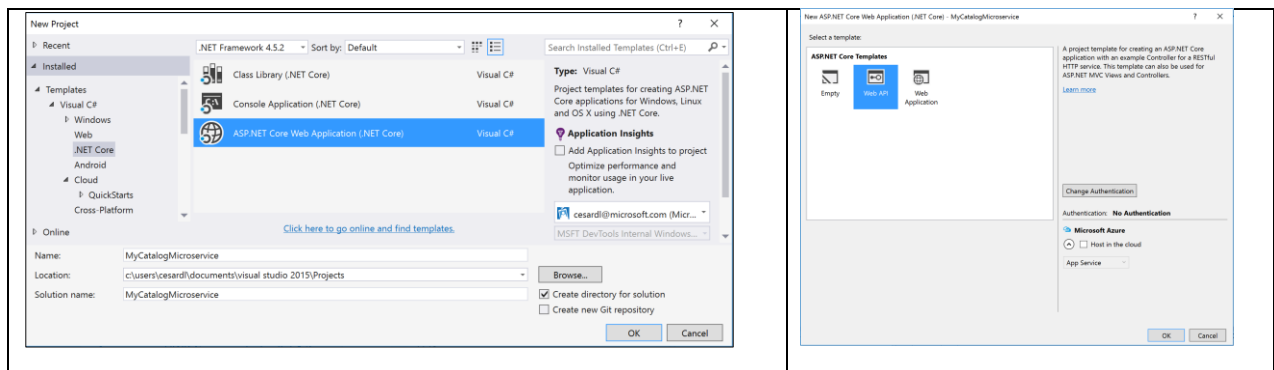
Figure X-XX. Creating an ASP.NET Core Web API project in VS

After creating the project, you can implement your MVC controllers like you would in any other Web API project, using the Entity Framework API or any other API. In the eShopOnContainers.Catalog.API project, you can see that the main dependencies for that microservice are just ASP.NET Core itself, Entity Framework and Swashbuckle:
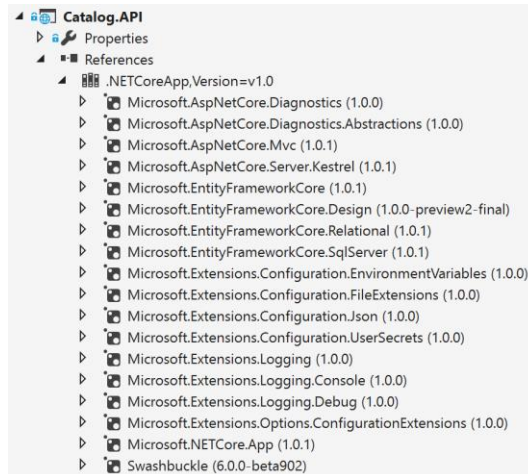
Figure X-XX. Dependencies in a simple CRUD Web API

Implementing CRUD Web API services with Entity Framework Core

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

The Catalog microservice 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 in the ASP.NET Web API microservice.

Add Entity Framework Core to your dependencies

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

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

The model

With EF Core, data access is performed by using a model. A model is made up of entity classes and a derived context that represents a session with the database, allowing you to query and save data. You can generate a model from an existing database, manually code a model to match your database, or use EF Migrations to create a database from your model (and evolve it as your model changes over time). In the case of the Catalog microservice we are using the latter approach. You can see an example of the Product entity class in figure X-X which is a simple POCO (Plain Old CLR Object) entity class.

```
        14 references | Unai, 25 days ago | 1 author, 1 change
 5  ⊟   public class CatalogItem
 6      {
            1 reference | Unai, 25 days ago | 1 author, 1 change
 7          public int Id { get; set; }
 8
            8 references | Unai, 25 days ago | 1 author, 1 change
 9          public string Name { get; set; }
10
            4 references | Unai, 25 days ago | 1 author, 1 change
11          public string Description { get; set; }
12
            5 references | Unai, 25 days ago | 1 author, 1 change
13          public decimal Price { get; set; }
14
            5 references | Unai, 25 days ago | 1 author, 1 change
15          public string PictureUri { get; set; }
16
            6 references | Unai, 25 days ago | 1 author, 1 change
17          public int CatalogTypeId { get; set; }
18
            1 reference | Unai, 25 days ago | 1 author, 1 change
19          public CatalogType CatalogType { get; set; }
20
            6 references | Unai, 25 days ago | 1 author, 1 change
21          public int CatalogBrandId { get; set; }
22
            1 reference | Unai, 25 days ago | 1 author, 1 change
23          public CatalogBrand CatalogBrand { get; set; }
24
            4 references | Unai, 25 days ago | 1 author, 1 change
25          public CatalogItem() { }
26      }
```

Figure X-XX. Sample POCO Entity class:

You also need the previously mentioned DbContext that represents a session with the database. 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 `OnModelCreating()` method being used in the `CatalogContext` class that automatically populates the sample data the first time it tries to access the database. This method is useful for demo data.

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

You can also use the OnModelCreating method ir oder to to customize object/database entity mappings as many other EF extensibility points.

Querying data from Web API controllers

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

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

```
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _context;
    private readonly IOptionsSnapshot<Settings> _settings;

    public CatalogController(CatalogContext context, IOptionsSnapshot<Settings> settings)
    {
        _context = context;
        _settings = settings;
        ((DbContext)context).ChangeTracker.QueryTrackingBehavior =
                                                QueryTrackingBehavior.NoTracking;
    }

    // GET api/v1/[controller]/items[?pageSize=3&pageIndex=10]
    [HttpGet]
    [Route("[action]")]
    public async Task<IActionResult> Items([FromQuery]int pageSize = 10,[FromQuery]int pageIndex=0)
    {
        var totalItems = await _context.CatalogItems
            .LongCountAsync();

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

        itemsOnPage = ComposePicUri(itemsOnPage);

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

        return Ok(model);
    }

  //…

}
```

Saving data

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

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

Figure X-XX. Saving

Dependency Injection in ASP.NET Core and Web API controllers

In ASP.NET Core you can use Dependency Injection (DI) out-of-the-box. There's no need to set up a third party IoC (Inversion of Control) container, 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.
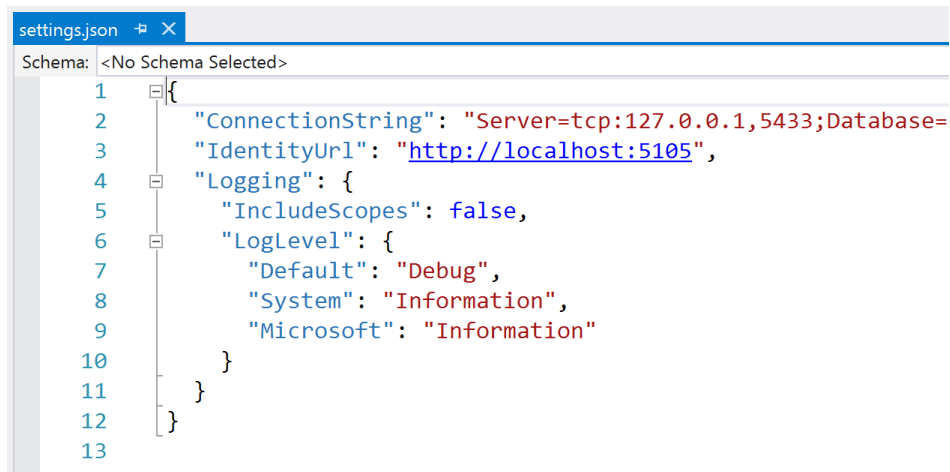
An important configuration to set up in the Web API project is the DbContext class registration into the services IoC container. You typically do so in the Startup.cs class and the `ConfigureServices()` method, with the `services.AddDbContext()` method, as shown in figure X-XX.

```csharp
public void ConfigureServices(IServiceCollection services)
{
    services.AddDbContext<CatalogContext>(c =>
    {
        c.UseSqlServer(Configuration["ConnectionString"]);
        // Changing default behavior when client evaluation occurs to throw.
        // Default in EF Core would be to log a warning when client evaluation is performed.
        c.ConfigureWarnings(warnings =>
                            warnings.Throw(RelationalEventId.QueryClientEvaluationWarning));
    });

    //…
}
```

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

The DB connection string and environment variables used by Docker containers

You can use the ASP.NET Core settings and add a `ConnectionString` property to your settings.json file as shown below.

Figure X-XX. Docker and environment variables for connection

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

From your docker-compose.yml or docker-compose.override.yml files you can initialize those environment variables, so that Docker will set them up as OS environment variables for you, as shown in the docker-compose.override.yml file below.

```yaml
# docker-compose.override.yml
#
  catalog.api:
    environment:
      - ConnectionString=Server=sql.data;Database=Microsoft.eShopOnContainers.Services.CatalogDb;
User Id=sa;Password=Pass@word
      - ExternalCatalogBaseUrl=http://10.0.75.1:5101
      #- ExternalCatalogBaseUrl=http://dockerhoststaging.westus.cloudapp.azure.com:5101
```

Developing Microservices with .NET Core

```
    ports:
      - "5101:5101"
```

The docker-compose.yml files at the solution level are not just more flexible than configuration files at the project/microservice level, but also more secure. Consider that the Docker images that you build per microservice do not contain the docker-compose.yml files, only binary files and configuration files per microservice, including the *dockerfile.* But since the docker-compose.yml file is not deployed along with your application but only used at deployment time, placing environment variables values within those docker-compose.yml files (even without encrypting the values) is still more secure than placing those values in regular .NET configuration files that will actually be deployed with your code.

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

RESTful web API Design and Implementation

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

----

References – API Design and Implementation
REST architectural style
http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm
API Design
https://docs.microsoft.com/en-us/azure/best-practices-api-design/
Best Practices for Designing a Pragmatic RESTful API
http://www.vinaysahni.com/best-practices-for-a-pragmatic-restful-api
API Implementation
https://docs.microsoft.com/en-us/azure/best-practices-api-implementation /

Versioning ASP.NET Web APIs

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

----

References – API Versioning
https://docs.microsoft.com/en-us/azure/best-practices-api-design#versioning-a-restful-web-api/

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

Swagger description metadata should probably be included with any kind of microservice, either Data-Driven microservices or more advanced Domain-Driven microservices (explained in following section). In this case we are using the simpler Data-Driven microservice implementation but you should also implement this feature in more complex microservices.

Swagger is a commonly used open source framework backed by a large ecosystem of tools that help you design, build, document, and consume your RESTful APIs. It is becoming the main standard for the APIs description metadata domain.

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

The main reasons why you would want to generate Swagger metadata about your APIs are 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, such as the following:
    - o **AutoRest** - It automatically generates .NET client classes for calling Swagger. It can be used fron the CLI and also integrates with Visual Studio for easy use through the GUI.
    - 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, such as when building complex microservice based applications, you will need to handle many endpoints with different data models used in the request/response payloads. Proper documentation and having a solid API explorer is key for the success of your API, as well as 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 them.

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

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

Swashbuckle 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 such as Azure API App, Azure Mobile App, Azure Service Fabric microservices based on ASP.NET, or plain Web API in containers, as in this case.

Swashbuckle combines API Explorer and Swagger/swagger-ui to provide a rich discovery and documentation experience 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 result for the API explorer will look like the figure X-XX below:
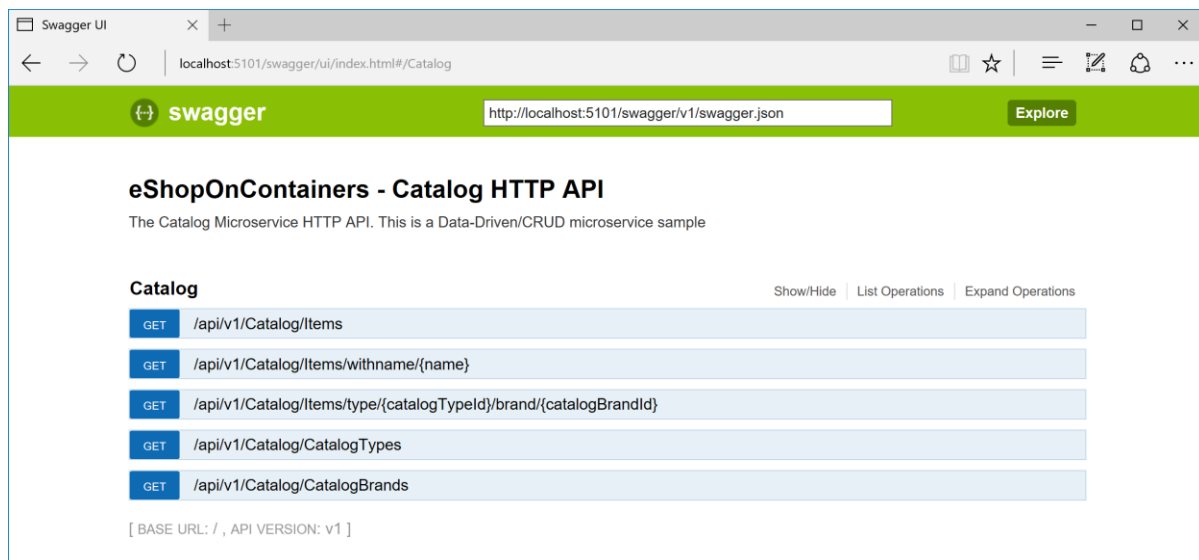
Figure X-XX. Swashbuckle UI based on Swagger metadata – eShop Catalog microservice example

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

Currently, Swashbuckle consists of two NuGet packages - *Swashbuckle.SwaggerGen* and *Swashbuckle.SwaggerUi*. The former provides functionality to generate one or more Swagger documents directly from your API implementation and expose them as JSON endpoints. The latter provides an embedded version of the swagger-ui tool that can be served by your application and powered by the generated Swagger documents to describe your API. However, recent versions of Swashbuckle wrap these with the *Swashbuckle.AspNetCore* meta-package

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

```csharp
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 figure X-XX you can also see how you can test any specific API method.
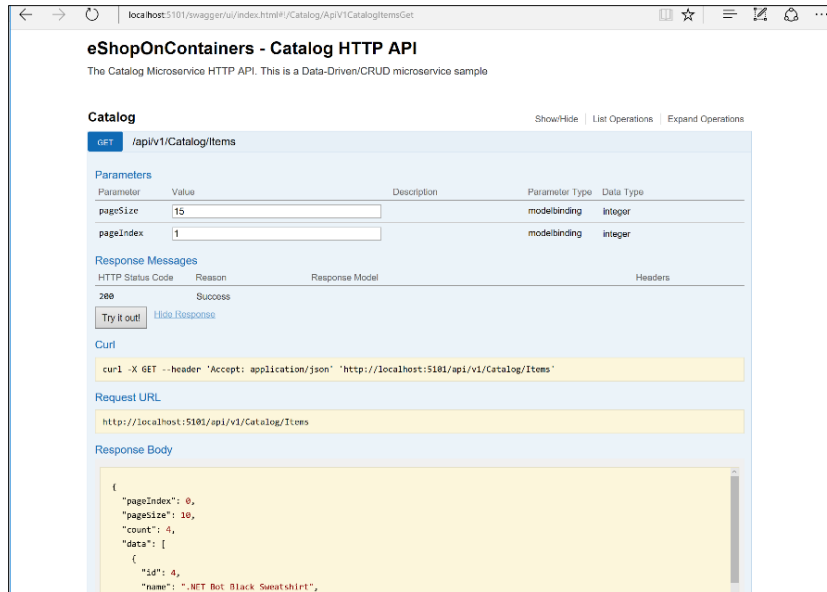
Figure X-XX. Swashbuckle UI testing the Catalog/Items API

In the following figure 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 the `<your-root-url>/swagger/v1/swagger.json` URL using the convenient [Postman](#) tool.
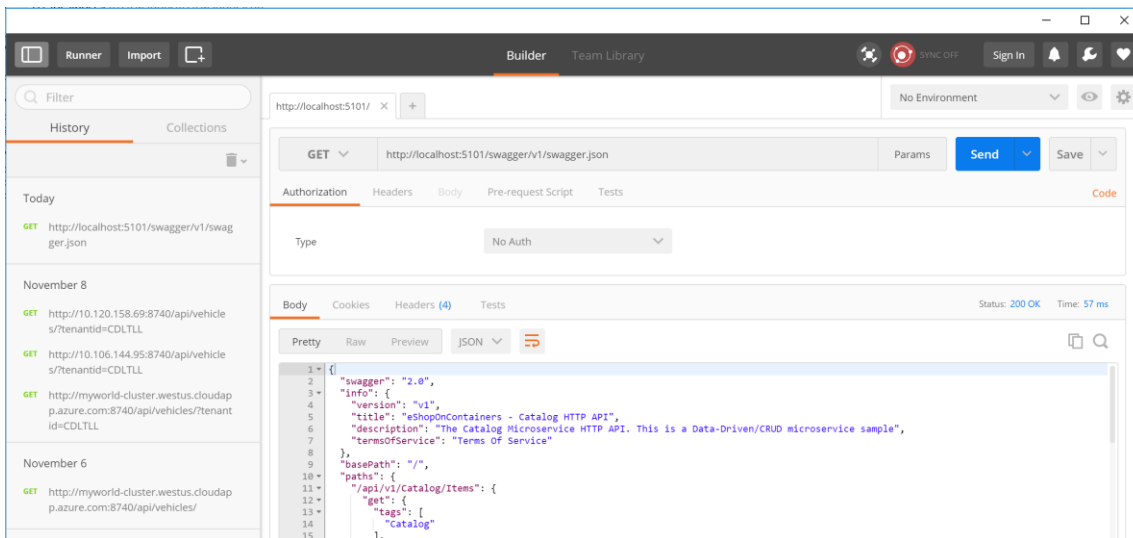
Figure X-XX. Swagger JSON metadata

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

NOTE: Currently, *Swashbuckle.AspNetCore is version 1.0.0-rc1* or later (now that it is refactored into a new package) is what you need to use for .NET Core Web API projects.

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

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

As previously introduced, you can explicitly describe how you would like to deploy your multi-container application in the docker-compose.yml file. Optionally, you can also describe how you are going to build your custom Docker images (custom Docker images can also be built with the Docker CLI). Basically, you define each of the containers you want to deploy plus certain characteristics for each container deployment. Then, once you have a multi-container deployment description file, you can deploy the whole solution in a single action orchestrated by the CLI command `docker-compose up`, or you can deploy it transparently from Visual Studio. Otherwise, you would need to use the Docker CLI to deploy container-by-container in multiple steps by using the command `docker run` from the command line. Therefore, each service defined in docker-compose.yml must specify exactly one of image or build. Other keys are optional, and are analogous to their `docker run` command-line counterparts.

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 additional ways to use the docker-compose files that are worth exploring in further detail.

The following yaml code is the definition of a possible global but single docker-compose.yml for the eShopOnContainers sample. This is not the actual docker-compose file from eShopOnContainers but a simplified and consolidated version in a single file, which is not the best way to work with docker-compose files, as will be explainedlater.

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

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

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 executing the `docker-compose` up command or deploying from Visual Studio by using this docker-compose.yml file. In this case, the docker-compose.yml file has multiple services defined, as described in the following table.

Service name in docker-compose.yml	Description
webmvc	Container with ASP.NET Core MVC app consuming the microservices from server-side C#
catalog.api	Container with the Catalog ASP.NET Core Web API microservice
ordering.api	Container with the Ordering ASP.NET Core Web API microservice
sql.data	Container running SQL Server for Linux, with the microservices' databases
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:
```

```
        - sql.data
```

This containerized service has the following basic configuration in place:

- It is based on the custom `eshop/catalog.api` image. In this case, for simplicity's sake, there is no "`build:` key" in the file, so the image must have been previously built (with `docker build`) or be available locally by downloading it with the `docker pull` command from any Docker registry before running the `docker-compose` up command using this docker-compose.yml file.
- It defines an environment variable named `ConnectionString` with the connection string to be used by Entity Framework to access the SQL Server containing the Catalog data model. In this particular case, the same SQL Server container is holding multiple databases so you'd need less memory in your development machine assigned to Docker, but if you'd want to, you could also deploy one SQL Server container per microservice's database.
- Note that the SQL server name is *sql.data*, which is the same name/id used for the container that is running the SQL Server for Linux. This is very convenient, as being able to use this name resolution (internal to the Docker host) it will resolve the network address so you don't need to know the internal IP for the containers you are accessing from other containers.
  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, for example 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 port 80 for internal access to the catalog.api within the Docker host. The host is currently a Linux VM because it is based on a Docker image for Linux, but you could configure the container to run on a Windows image, too.
- Forwards the exposed port 80 on the container to port 5101 on the Docker host machine (The Linux VM).
- Links the web service to the sql.data service, (the SQL Server for Linux database running in a container). This is useful as by specifying this dependency, the Catalog.API container won't start until the sql.data container has already started, as you need to have the SQL database up and running first. However, this kind of container dependency is not enough in many cases as Docker checks only at the container level, but sometimes the service (in this case SQL Server) might still not be ready, so it is advisable to implement retry logic with exponential backoff in your client microservices so in case a dependency container is not ready for a short time, the app will still be resilient.
- Accessing external servers: Another interesting property is the "`extra_hosts`" which allows you to access external servers or machines out of the Docker host (i.e. out of the default Linux VM which is a development Docker Host). A typical case is when you need to access a database server any other server type placed out of the Docker Host, for instance a local SQL Server on your development PC.

```
extra_hosts:
    - "CESARDLSURFBOOK:10.0.75.1"
```

There are other more advanced and very interesting possible configuration settings at the docker-compose.yml level worth mentioning in the following sections.

Use docker-compose files to target multiple environments

The docker-compose.yml files are definition files and could be used by multiple infrastructures are able to understand that format. The most straightforward tool is the command docker-compose, but other more advanced scenarios like orchestrators (like Docker Swarm) could also understand that file.

Therefore, by using the command docker-compose you can target the following main scenarios (extracted from Docker docs).

Development environments

When you're developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.

The Compose file provides a way to document and configure all of the application's service dependencies (databases, queues, caches, web service APIs, etc). Using the Compose command line tool you can create and start one or more containers for each dependency with a single command (docker-compose up).

Together, these features provide a convenient way for developers to get started on a project. Compose can reduce a multi-page "developer getting started guide" to a single machine readable Compose file and a few commands.

Automated testing environments

An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests. Compose provides a convenient way to create and destroy isolated testing environments for your test suite. By defining the full environment in a Compose file you can create and destroy these environments in just a few commands:

```
$ docker-compose up -d
$ ./run_tests
$ docker-compose down
```

Production deployments

You can also use Compose to deploy to a remote Docker Engine. A typical case could be to deploy to a single Docker host instance (like a production VM or server provisioned with [Docker Machine](#)) but it could also be an entire [Docker Swarm](#) cluster which also I compatible with compose.

If using any other orchestrator is pretty possible that you might need to add extra configuration setup and metadata comparable to docker-compose.yml but in the additional format required by the other orchestrator as Azure Service Fabric, Mesos DC/OS, Kubernetes, etc.

In any case, docker-compose is a very convenient tool for development and testing workflows.

The production workflow will depend on your selected orchestrator.

Using multiple Compose files to handle several environments

When targeting different environments, you should use multiple compose files so you can create multiple configuration variants depending on the environment.

Overriding the base docker-compose file

You could use a single docker-compose.yml file as in the initial simplified examples shown in previous sections, however, that is not usually recommended for most applications.

By default, Compose reads two files, a *docker-compose.yml* and an *optional docker-compose.override.yml* file.

As shown in image X-XX, when using Visual Studio and enabling Docker support, it also creates those files plus some additional files used for debugging.
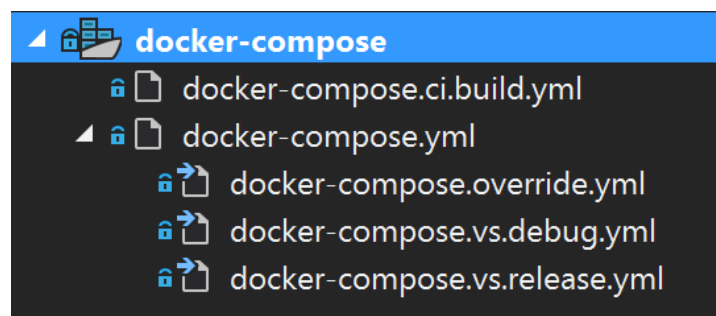
Figure X-XX. docker-compose files in Visual Studio 2017

You can also edit the docker-compose files with any editor like Visual Studio Code or Sublime Text and run it with the docker-compose up command tool. Visual Studio is not required but convenient when using .NET and Docker.

By convention, the *docker-compose.yml* should contain your base configuration. That means service's configuration that should not change depending on the deployment environment you are targeting.

The docker-compose.override.yml file, as its name implies, contains configuration that overrides the base configuration. For instance, configuration that depends on the deployment environment.

The override files usually contain small pieces of configuration, added info needed by the application but specific per environment or specific per deployment.

Multiple environments use case

A typical use case is shaping multiple compose files so you can target multiple environments, like production, staging, CI, development, etc.

To support these differences, you can split your Compose configuration into multiple files, as shown in image X-XX.

Multiple docker-compose files
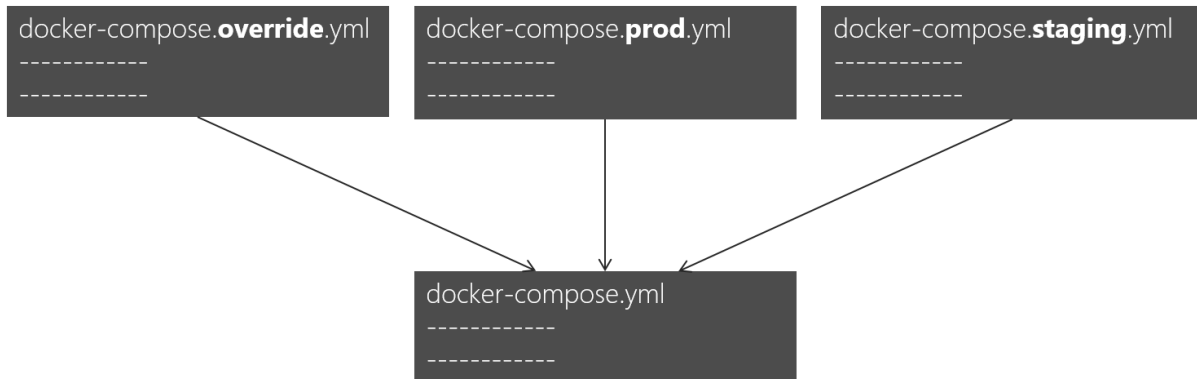
Figure X-XX. Multiple docker-compose files overriding values

In the first place, you have to start with the base *docker-compose.yml* file. This base file has to contain the "static" configuration that won't change depending on the environment. For example, in eShopOnContainers, you have the following *docker-compose.yml* as the base file.

```yaml
#docker-compose.yml (Base)

version: '2'
services:
  basket.api:
    image: eshop/basket.api
    build:
      context: ./src/Services/Basket/Basket.API
      dockerfile: Dockerfile
    depends_on:
      - basket.data
      - identity.api

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

  identity.api:
    image: eshop/identity.api
    build:
      context: ./src/Services/Identity/Identity.API
      dockerfile: Dockerfile
    depends_on:
      - sql.data

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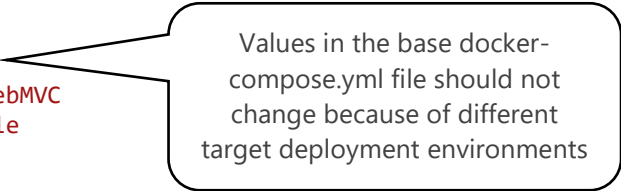
```
        - sql.data

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

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

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

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

Values in the base docker-compose.yml file should not change because of different target deployment environments

If you focus on the service definition "webmvc" for instance, you can see how that information I pretty much the same no matter what environment you might be targeting. You have the following information:

- Service name: "webmvc"
- Container's custom image: eshop/webmvc
- Command to build the custom Docker image saying what dockerfile has to use
- Dependencies on other services, so this container does not start until the other dependency containers have started.

You can have additional configuration but the important point here is that in the base *docker-compose.yml* file you just want to set the information that is common across environments.

Then, in the docker-compose.override.yml or similarfiles for "prod" or "staging" you should place configuration which is specific for each environment.

Usually, the plain "docker-compose.override.yml" will be used for your development environment, as in the following example from eShopOnContainers.

```
#docker-compose.override.yml (Extended config for DEVELOPMENT env.)

version: '2'

services:
# Simplified number of services here:

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

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

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

  sql.data:
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
```

> Values in the base docker-compose.override.ymlare specific per deployment environment, like connection-strings

> External URLs for redirection using "localhost", only for this development/test environment

> Specific SQL credentials and external/internal port mapping for this development/test environment

In this example the development override configuration exposes some ports to the host, defines environment variables with redirect URLs and specifies connection strings for the development environment. All that, specific for that development/test environment.

When you run "docker-compose up" (or launch it from Visual Studio) it reads the overrides automatically like if it were merging both files.

Now you would like to have another Compose app file for the production environment with probably, different configuration data. So, you can create another override file (which might be stored in a different git repo or managed/secured by a different team), like the following.

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

services:
# Simplified number of services here:

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

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

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

  sql.data:
    environment:
      - SA_PASSWORD=Prod@Pass
      - ACCEPT_EULA=Y
    ports:
      - "5433:1433"
```

> Values in the base docker-compose.prod.yml are specific for the production environment, like connection-strings

> External URLs for redirection using env-vars containing the production DNS or IP

> Specific SQL credentials and external/internal port mapping for SQL. Although, in a real Production environment you might get rid of the SQL container and move the databases to Azure SQL DB or a SQL cluster on-premises.

How to deploy with a specific override file

To use multiple override files, or an override file with a different name, you can use the -f option to specify the list of files. Compose merges files in the order they're specified on the command line as in the following command to run from the CLI.

```
docker-compose -f docker-compose.yml -f docker-compose.prod.yml up -d
```

Using Environment Variables in the docker-compose files

Especially in production environments it is vey convenient to be able to get configuration information from the environment variables, like in the previous examples or the following extracted line from a docker-compose.prod.yml file.

```
IdentityUrl=http://${ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP}:5105
```

By specifying an environment variable's name as "${MY_VAR}" you can propagate environment variables to your docker-compose files.

Environment variables can be created or initialized in a different way depending on your host environment (Linux, Windows, Cloud cluster, etc.), however, a very convenient tip is to use the **.env** file.

Compose supports declaring default environment variables in an environment file named **.env** placed in the folder where docker-compose command is executed from (current working directory), as the following .env file.

```
# .env file

ESHOP_EXTERNAL_DNS_NAME_OR_IP=localhost

ESHOP_PROD_EXTERNAL_DNS_NAME_OR_IP=10.121.122.92
```

Compose expects each line in an env file to be in VAR=VAL format. Lines beginning with # (i.e. comments) are ignored, as are blank lines.

It is important to highlight that the values present in the environment at runtime will always override those defined inside the .env file. Similarly, values passed via command-line arguments take precedence as well.

References – Docker Compose   files
Overview of Docker Compose
https://docs.docker.com/compose/overview/
Multiple Compose files
https://docs.docker.com/compose/extends/#multiple-compose-files

Building Optimized ASP.NET Core Docker Images

If you are exploring Docker and .NET Core in the Internet, you often see dockerfiles that demonstrate the simplicity of building a docker image by copying your source into a container and voila, you have a docker image with the environment packaged with your app.

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

While you can do so, there are big optimizations to be had, especially for your production images.

In the container and microservices model, you are constantly starting containers. The common model doesn't restart a sleeping container as they're disposable. The orchestrators (like Docker Swarm, Kubernetes, DCOS or Azure Service Fabric) simply instance new instances of a common image. What this means is that you need to optimize, pre-compile the app when built. When the container is started, it's already to run, you shouldn't restore/compile at run time.

The .NET team has been doing a lot of great work to make .NET Core and ASP.NET Core a container optimized framework. Not only .NET Core is a lightweight framework with a small memory footprint, but we have also focused on startup performance and produced some optimized Docker images, like the microsoft/aspnetcore image available at Docker Hub. That image which compared to the regular microsoft/dotnet or the microsoft/nanoserver images, the microsoft/aspnetcore image provides automatic setting of aspnetcore_urls to port 80 and the "pre-ngend" cache of assemblies which provides faster startup.

References – Optimizing Docker Images
Building Optimized Docker Images with ASP.NET Core
https://blogs.msdn.microsoft.com/stevelasker/2016/09/29/building-optimized-docker-images-with-asp-net-core/

Building the application bits from a Build/CI container

Another great benefit from Docker is that you could build your bits from a pre-configured container, so you wouldn't need to create a build machine or VM to build your application bits. You could use/test that "build container" from your development machine but, what is more interesting, you could use the same "build container" from your CI (Continuous Integration) pipeline.

Building the Application's bits from a container
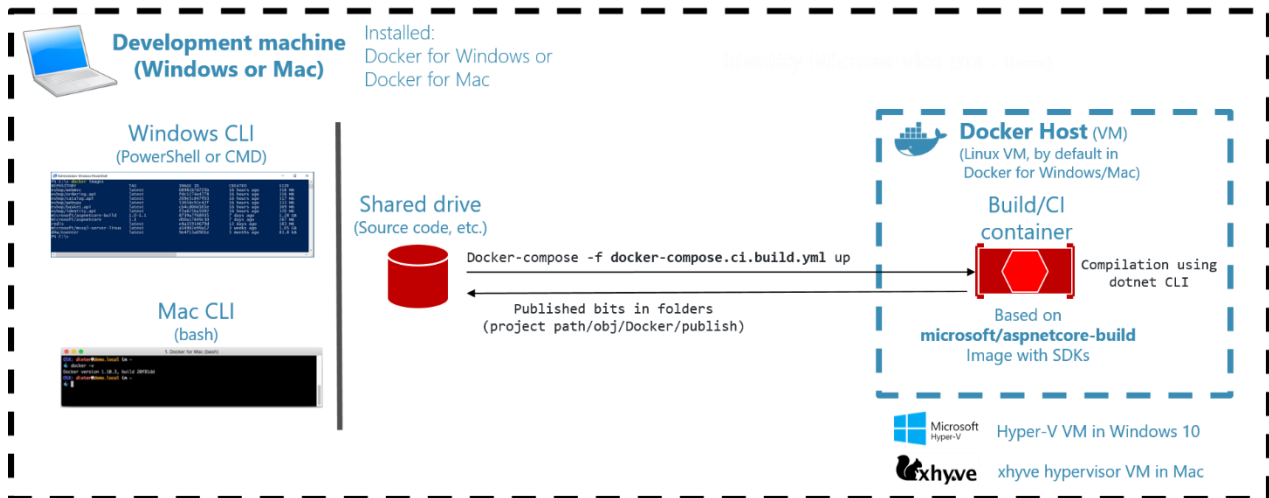
Figure X-XX. Components building .NET bits from a container

Especially made for that scenario, we provide the microsoft/aspnetcore-build image used to compile and build ASP.NET Core apps. The output would be placed in a microsoft/aspnetcore image which is an optimized runtime image, as mentioned.

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

- .NET Core
- ASP.NET SDK
- NPM
- Bower
- Gulp

While we need these dependencies at build time, we don't want to carry these with our app at runtime as it would just make the image unnecessarily bigger.

Precisely, in the eShopOnContainers you can build the application bits from a container by just running the following docker-compose command.

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


Figure X-XX. Building .NET bits from a container

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

That [docker-compose.ci.build.yml](#) contains the following code where you can see it will spin-up a buildcontainer using the mentioned [microsoft/aspnetcore-build](#) image.

```
version: '2'

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

Once the build container is up and running, then it is running the .NET SDK commands "dotnet restore" and "dotnet publish" against all the projects in the solution, in order to compile the .NET bits.

In this particular case, because eShopOnContainers also has a SPA application (Single Page Application) based on TypeScript/JavaScript and Angular 2 for the client code, it also needs to check it with NPM, but that action is not related to the .NET bits.

The command "dotnet publish" will build and publish the compiled bits within each project's folder at the "**../obj/Docker/publish**" folder, like in the image X-XX below.
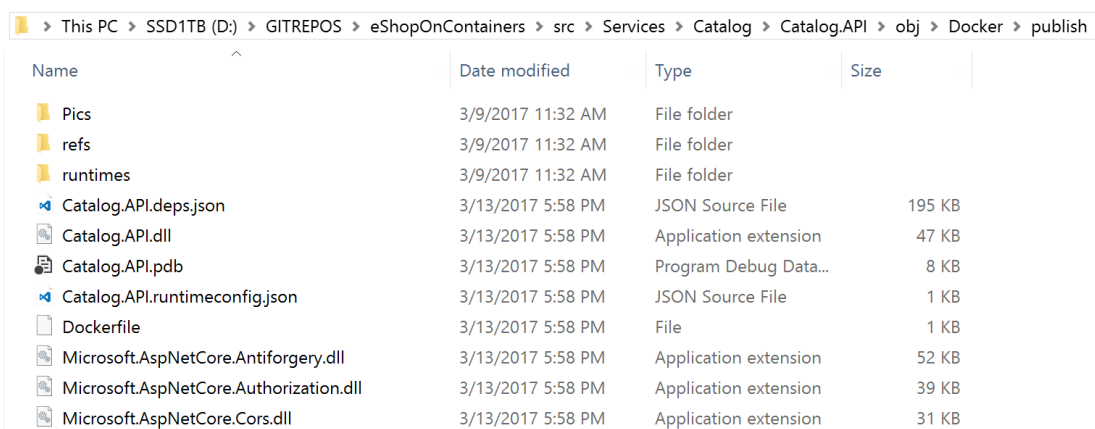
Figure X-XX. Binary files generated by "dotnet publish"

Creating the Docker images from the CLI

Once you have the application bits published at folders, the next step is to actually build the Docker images with "docker-compose build/up", as illustrated in the image X-XX.

Figure X-XX. Building Docker images and running the containers

In the following image X-XX, you can see how the docker-compose build command is run.

Figure X-XX. Building the Docker images with "docker-compose build"

The difference between the command "docker-compose build" and "docker-compose up" is that the second will also spin-up the containers ion addition to build the images.

When using Visual Studio, all those steps are performed under the covers. Visual Studio will compile the .NET application bits, create the Docker images and deploy the containers into the Docker host, plus many important value-added features, like the ability to debug your containers running in Docker, directly from Visual Studio.

The important point for this section is that, as mentioned, you are able to build your application bits the same way your CI/CD pipeline will build it, from a container instead of

the developer's machine. Afterwards, you build and run the Docker images with "docker-compose up".

References – Building the application bits from a Build/CI container
Building bits from a container: Setting the eShopOnContainers solution up in a Windows CLI environment (dotnet CLI, Docker CLI and VS Code) https:// https://github.com/dotnet/eShopOnContainers/wiki/03.-Setting-the-eShopOnContainers-solution-up-in-a-Windows-CLI-environment-(dotnet-CLI,-Docker-CLI-and-VS-Code)

A database server running as a container

SQL Server running as a container with a microservice-related database

You can have your databases (SQL Server, PostgreSQL, MySql, etc.) in regular standalone servers, on-premises clusters or in PaaS services in the cloud, like Azure SQL DB. However, for development and test environments, having your databases running as containers is pretty convenient as you won't have any external dependency and a simple docker-compose would spin-up the whole application. Having those databases as containers is also great for integration tests, because the database is started in the container and always populated with the same sample data, so tests can be more predictable.

In eShopOnContainers, there is a specific container named `sql.data` that runs SQL Server for Linux with all the SQL Server databases needed for the microservices. You could also have one SQL Server container per database, but that would require a lot more of memory assigned to Docker. The important point in microservices is that each microservice "owns" it related database. Then, databases could be anywhere.

The SQL Server container 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.

Note that the following code has consolidated configuration from the generic docker-compose.yml and the docker-compose.override.yml. Usually you'd separate the environment info from the static info related to the SQL Server image.

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

A "docker run" command could run that container.

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

However, if deploying a multi-container application like eShopOnContainers, using "docker-compose up" is a much more convenient method so it deploys all the needed containers for the application to run.

When starting this SQL Server 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 initializes each microservice database with sample data by seeding with 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. As mentioned, it is also great for development and testing environments so you can easily run integration tests starting from a clean SQL Server image and known 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

You should run Redis on a container expecially for dev/test and proof of concept environments. This is very convenient as you can have all your dependencies running on containers not just for your local development machines but also for your testing environments in your CI/CD pipelines.

However, when running Redis in production it is probably better to look for a high available solution like Redis Microsoft Azure which runs as a PaaS service. In terms of your code, you will just need to change your connection strings.

Redis provides a Docker image with Redis set up. That image is available at Docker Hub:

https://hub.docker.com/_/redis/

You can directly run a Docker Redis container by simple executing the following Docker CLI command:

```
$ docker run --name some-redis -d redis
```

The Redis image includes "Expose 6379" (the redis port), so standard container linking will make it automatically available to the linked containers.

In *eShopOncontainers* the Basket.API microservice uses a Redis cache running as a container. That Basket.data container is defined as part of the multi-container docker-compose.yml file.

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

That code at the docker-compose.yml defines a container named "basket.data" based on the "redis" image and publishing the port 6379 internally, meaning that it will be accessible only from other containers running within the Docker Host.

Finally, at the docker-compose.override.yml file, the Basket.API microservice at eShopOnContainers has defined the connection string to use for that Redis container.

```
basket.api:
  environment:
    //Other data…
    - ConnectionString=basket.data
    - EventBusConnection=rabbitmq
```

Implementing event based communication between microservices: Integration Events

As introduced in the initial architecture section in this guide, when using this type of communication, a microservice publishes an event when something notable happens, for example when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event, it can update its own business entities, which might lead to more events being published. This subscription/publication system is usually performed by using an implementation of an Event Bus. The Event Bus can be designed as an abstraction/interface with the API needed to subscribe/unsubscribe to events and to publish events, plus one or more implementations based on any inter-process or messaging communication, such as a messaging queue or a Service Bus supporting asynchronous communication and a subs/pubs model.

You can use events to implement business transactions that span multiple services, and you 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.
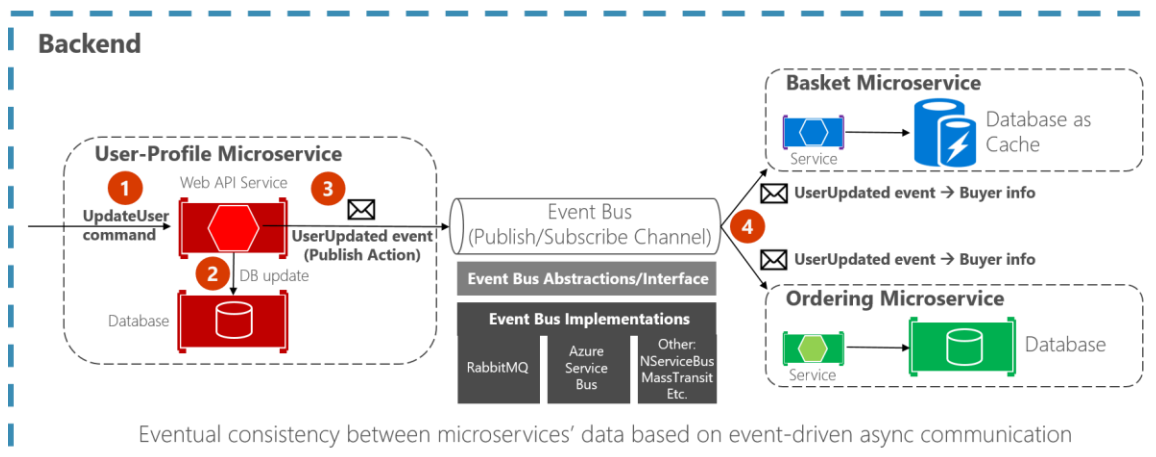
Figure X-XX. Event driven communication based on an Event Bus

As shown in the image X-XX, this section tackles how you can implement this type of communication with .NET by using a generic Event Bus abstraction/interface. There are multiple potential implementations, each using a different technology or infrastructure such as RabbitMQ, Azure Service Bus or any other third party Open Source or Commercial Service Bus.

Important note on messaging brokers and Service Buses for production systems: As introduced in the architecture section, notice that among the multiple messaging technologies you can choose for implementing your abstract Event Bus, some of them can be at a different level than others. For instance, *RabbitMQ* (messaging broker transport) sits on a lower level than other commercial products like Azure Service Bus, NServiceBus, MassTranssit or Brighter. Most of them can work on top of either RabbitMQ or Azure Service Bus. It really depends on how many features and how much out-of-the-box scalability you need for your application. For implementing just an Event Bus proof of concept for your development environment, as in the *eShopOnContainers* sample, a simple implementation on top of RabbitMQ running as a container might be enough. But for mission-critical and production systems that need high scalability, you might want to evaluate and use Azure Service Fabric. If you require high level abstractions and richer features like [Sagas](#) for long running processes that make distributed development easier, other commercial and Open Source service buses like *NServiceBus*, *MassTransit* and *Brighter* are worth evaluating. Of course, you could always build your own service bus features on top of lower level technologies like RabbitMQ and Docker, but the plumbing work needed to "re-invent the wheel" might be too costly for a custom enterprise application.

Reiterating on this important topic, the sample Event Bus abstractions and implementation showcased at *eShopOnContainers* is intended to be used only as a proof of concept. Once you decided that you want to have asynchronous and event-driven communication as explained in the present guidance, you should choose between any of the mentioned Service Buses in the market and substitute everything related to the Event Bus in your code, probably even eliminating your own Event Bus abstractions that you had that could be limiting your production ready Service Bus which usually can already work on top of different "transports" like RabbitMQ or Azure SB so your own abstractions might not make sense unless you just want a very basic Event Bus.

Integration Events

Integration events are usually used for bringing certain domain state in sync across multiple microservices or even external systems. This is done by publishing integration events to outside the microservice. When an event is published to multiple receptor microservices (as many as microservices are subscribed to the integration event) then the appropriate Event Handler (from each microservice subscribed to that event) handles the event.

In terms of code, and integration event is basically a data holding class, like in the following example.

```csharp
public class ProductPriceChangedIntegrationEvent : IntegrationEvent
{
    public int ProductId { get; private set; }
    public decimal NewPrice { get; private set; }
    public decimal OldPrice { get; private set; }

    public ProductPriceChangedIntegrationEvent(int productId, decimal newPrice,
decimal oldPrice)
    {
        ProductId = productId;
```

```
        NewPrice = newPrice;
        OldPrice = oldPrice;
    }
}
```

The integration event class can be really simple, maybe simply containing a GUID as for its Id.

The integration events would be defined at the application level of each microservice, so they are decoupled. Pretty much in a comparable way to ViewModels defined in the server and client side. What is not recommended is to share a common integration events library across multiple microservices because you will be coupling those microservices with a single event definition data library for the same reasons you don't want to share a common domain model across multiple microservices. Microservices must be completely autonomous.

The only libraries you could share across microservices are final building blocks or tools that could also be shared as a Nuget component like you actually do with JSON serializers or any other tool library.

The Event Bus

An Event Bus allows publish and subscribe-style communication between microservices without requiring the components to explicitly be aware of each other, as shown in the image X-XX.
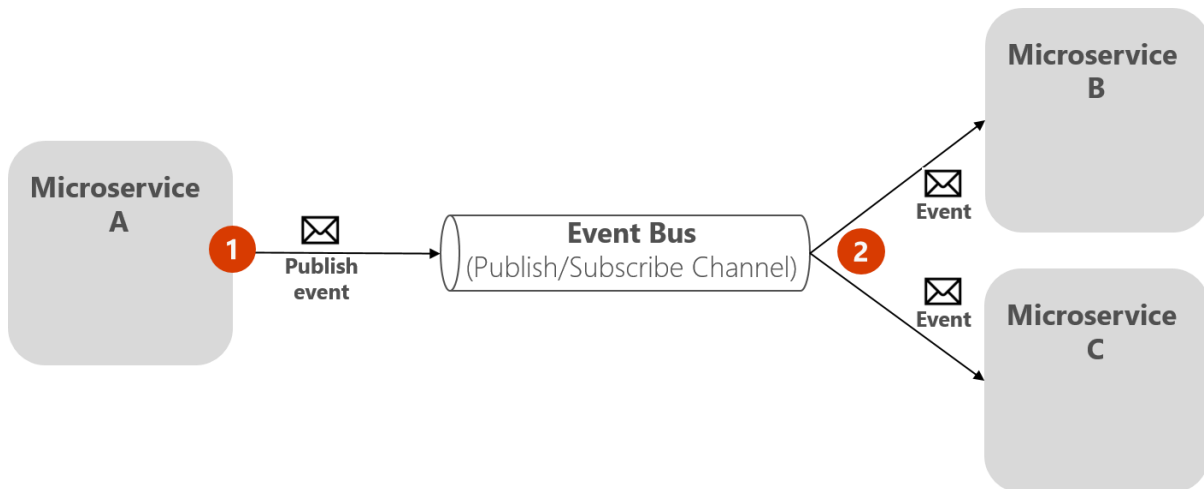
Figure X-XX. Publish/Subscribe basics with an Event

The Event Bus is related to the Observer pattern and the Pub-sub pattern.

Observer pattern: This is a pattern of development in which your primary object (known as the Observable) notifies other interested objects (known as Observers) with relevant information (events).

Pub-sub pattern: The objective of the pub-sub pattern is the same as the Observer pattern. You want to notify other classes when certain events take place. There's an important semantic difference between the Observer and Pub-sub patterns: in the pub-sub pattern the focus is on broadcasting messages. The Observable doesn't want to know who the events are going out to, just that they've gone out. In other words, the Observable (a.k.a. Publisher) doesn't want to know who the Observers (a.k.a. Subscribers) are.

The middle-man or Event Bus: How do you achieve that anonymity between publisher and subscriber? An easy way is let a middleman take care of all the communication. An event bus is one such middleman.

An Event Bus is typically composed of two main parts:

- The abstraction or interface.
- One or more implementations.

In the figure <mark>X-XX</mark> you can see how, from an application point of view, the Event Bus is nothing more than a pub-sub channel. The way you implement that asynchronous communication can vary and could have multiple implementations so you can swap between them depending on the environment requirements (production vs. development environments, for instance).

In figure <mark>X-XX</mark> you can see an abstraction of an Event Bus with multiple implementations based on infrastructure messaging technologies like RabbitMQ, Azure Service Bus or other service buses like NServiceBus, MassTransit, etc.
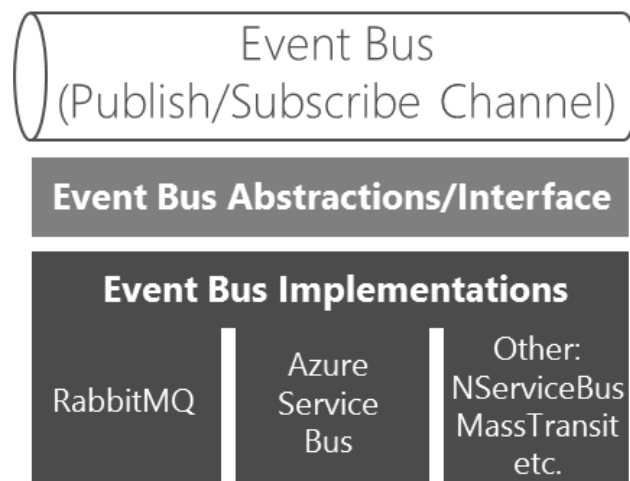
Figure <mark>X-XX</mark>. Multiple implementations of an Event

However, and as highlighted previously, this is only a possibility if you just need very basic Event Bus features supported by your abstractions. If you need richer "Service Bus features", you'd better directly use the API provided by your chosen service bus in the market instead of your own abstractions.

Let's start with some implementation code for the interface and possible specific implementations for experimentation and exploration purposes.

Defining an Event Bus interface

The abstraction or interface should be generic and straightforward, like the following interface.

```csharp
public interface IEventBus
{
    void Publish(IntegrationEvent @event);
    void Subscribe<T>(IIntegrationEventHandler<T> handler)
            where T: IntegrationEvent;
    void Unsubscribe<T>(IIntegrationEventHandler<T> handler)
            where T : IntegrationEvent;
```

```
}
```

The *Publish* method is very straightforward. The event bus will publish or broadcast the provided integration event to any endpoint or microservice subscribed to that event. This method will be used by the microservice who is publishing the event.

The *Subscribe* method will be used by all the microservices who want to receive events and it this method has two important parts to take into account. First, the *integration event* you want to be subscribed to. Second, the integration *event handler* (or callback method) you want the system call when receiving that integration event message.

Implementing an Event Bus with RabbitMQ (Dev/Test environment purposes)

First of all, it is important to highlight that if you create your custom Event Bus implementation based on RabbitMQ running in a container, like eShopOnContainers application does, it should be intended to be used only for your dev/test environments but not for your production environment, unless you are really building a production ready service bus. A simple custom event bus might be missing many production ready critical features that a service bus in the market (like the ones previously mentioned) already has available for resilient systems.

The custom implementation of an Event Bus is basically a library using RabbitMQ API so the microservices can subscribe to events, publish events and receive them, as in the following image X-X.
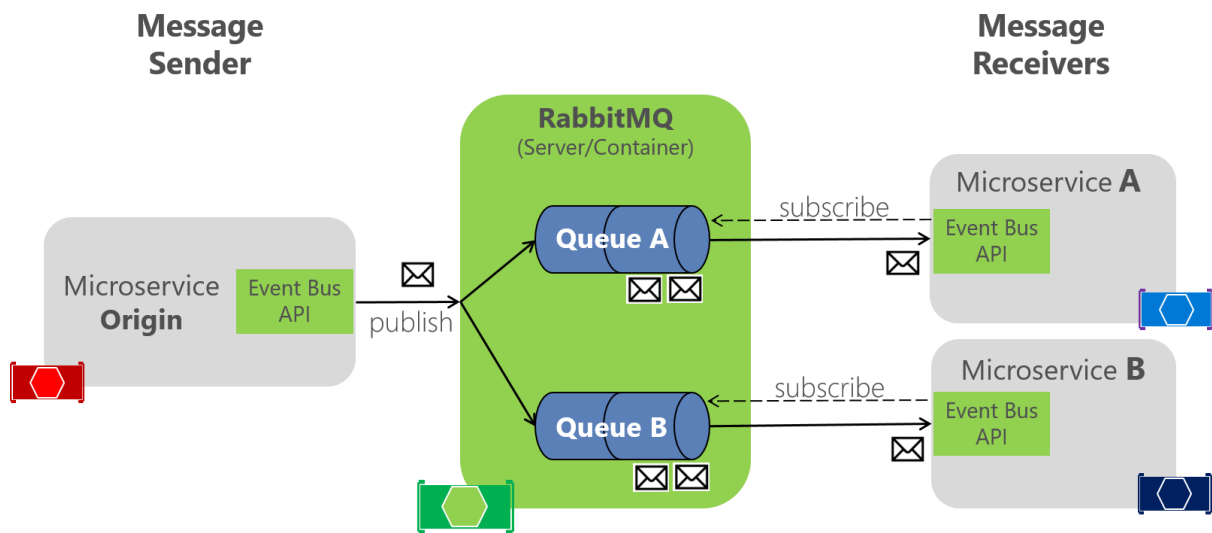
Figure X-XX. *RabbitMQ* implementation of Event Bus

In terms of code, the **EventBusRabbitMQ** class would implement the generic **IEventBus** so based on Dependency Injection you will be able to swap from this dev/test version or a production version implementation (like the Service Buses mentioned previously).

```
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    //Implementation using RabbitMQ API
    //...
```

The RabbitMQ implementation of a sample dev/test Event Bus (not ready for production) is boiler plate code that basically has to handle the connection to the RabbitMQ server, provide basic code on how to publish a message event to the queues and finally, a Dictionary of collections of integration event handlers per event type which would have a different instantiation and different subscriptions per each receiver microservice/container, as shown in the figure X-XX.

Implementing a simple Publish method with RabbitMQ

This code would be part of the Event Bus implementation for RabbitMQ, so you usually won't need to code it, unless when improving the code of the Event Bus.

Basically, you need to get a connection and channel to RabbitMQ, create a message and publish the message into the queue.

```csharp
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods…
    //…
    public void Publish(IntegrationEvent @event)
    {
        var eventName = @event.GetType().Name;
        var factory = new ConnectionFactory() { HostName = _connectionString };
        using (var connection = factory.CreateConnection())
        using (var channel = connection.CreateModel())
        {
            channel.ExchangeDeclare(exchange: _brokerName,
                                    type: "direct");

            string message = JsonConvert.SerializeObject(@event);
            var body = Encoding.UTF8.GetBytes(message);

            channel.BasicPublish(exchange: _brokerName,
                                 routingKey: eventName,
                                 basicProperties: null,
                                 body: body);
        }
    }
}
```

As mentioned, there are many possible configurations in RabbitMQ so this code should be used only for dev/test environments and move to Service Bus products in the market for using a production ready and scalable Event Bus.

Implementing the Subscription code with RabbitMQ API

Again, this code would be part of the Event Bus implementation for RabbitMQ, so you usually won't need to code it, unless when improving the code of the Event Bus.

```csharp
public class EventBusRabbitMQ : IEventBus, IDisposable
{
    // Member objects and other methods…
    //…
    public void Subscribe<T>(IIntegrationEventHandler<T> handler)
                where T : IntegrationEvent
    {
```

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

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

Each event type will have a related channel to get events from RabbitMQ. Then, you can have 'n' event handlers per channel and event type.

That code basically receives an *IntegrationEventHanler* which is like a callback method in the current microservice plus its related IntegrationEvent, then it adds that event handler to the list of event handlers that each integration event type can have per client app or microservice. If not existent, it will also create a channel per event type so it can receive events in a "push style" from RabbitMQ when any event is published.

Subscribing to events

The first step for using the Event Bus is to subscribe the microservices or even external applications to the events they want to receive. That should be done from the receiver microservices.

The following simple code can be what each receiver microservice would need to implement when starting the service (e.i. at the Startup.cs class) so it subscribes to the needed events. For instance, the basket microservice wants to subscribe to the *ProductPriceChangedIntegrationEvent* so it is aware of any product's price change and can warn the user about it if that product exists in the basket.

```
var eventBus = app.ApplicationServices.GetRequiredService<IEventBus>();
        eventBus.Subscribe<ProductPriceChangedIntegrationEvent>(
                                        ProductPriceChangedIntegrationEventHandler);
```

Basically, after this code is executed, the subscriber microservice will be listening through RabbitMQ channels and when any message of type "*ProductPriceChangedIntegrationEvent*" comes, it will execute the provided Event Handler to process that event.

Publishing Events through the Event Bus

Finally, the message sender or origin microservice would publish the integration events with code similar to the following (simplified example not taking into account atomicity) which will be implemented whenever an event has to be propagated across multiple microservices, usually right after committing data/transactions from the original microservice.

```
[Route("update")]
```

```csharp
[HttpPost]
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem product)
{
    var item = await _context.CatalogItems.SingleOrDefaultAsync(
                                                i => i.Id == product.Id);
    //…
    if (item.Price != product.Price)
    {
        var oldPrice = item.Price;
        item.Price = product.Price;
        _context.CatalogItems.Update(item);

        var @event = new ProductPriceChangedIntegrationEvent(item.Id,
                                                item.Price,
                                                oldPrice);

        // Commit changes in original transaction
        await _context.SaveChangesAsync();

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

> Event Bus implementation injected through Dependency Injection coming from the Controller's constructor

In this case, since the origin microservice is a simple CRUD microservice, that code is placed right into a Web API controller. In other more advanced microservice it could be implemented at the CommandHandlers or DomainEventHandlers right after committing the original data.

The specific Event Bus implementation (RabbitMQ or based on a Service Bus) would be injected at the Controller's constructor, like in te following code.

```csharp
[Route("api/v1/[controller]")]
public class CatalogController : ControllerBase
{
    private readonly CatalogContext _context;
    private readonly IOptionsSnapshot<Settings> _settings;
    private readonly IEventBus _eventBus;

    public CatalogController(CatalogContext context,
                             IOptionsSnapshot<Settings> settings,
                             IEventBus eventBus)
    {
        _context = context;
        _settings = settings;
        _eventBus = eventBus;
        //…
    }
}
```

> Event Bus implementation injected through constructor based Dependency Injection

Achieving Atomicity and Resiliency when publishing to the Event Bus

When publishing integration events through a distributed messaging system like your Event Bus based on RabbitMQ or any Service Bus in the market, you have the problem of atomically updating

Developing Microservices with .NET Core

the original database and "publishing" an event. For instance, in the simplified example showed before the code is committing data into the database when the product price is changed, then publishing the *ProductPriceChangedIntegrationEvent*. Initially, it would look essential that those two operations were done atomically. However, if using a distributed transaction involving the database and the Message Broker, like you could do in older systems like [MSMQ (Microsoft Message Queuing)](#), is not recommended for the reasons described at the [CAP theorem](#). Basically, you use microservices to build scalable and high available systems. Simplifying, the CAP theorem says that you cannot build a database (but also related to systems based on microservices as each microservice owns its model) that is continually available, sequentially consistent and tolerant to any partition pattern. You have to choose just two of these three properties. In microservices based architectures you choose availability and tolerant not favoring ACID consistency across multiple distributed resources. Therefore, in most modern microservice based applications, you usually don't want to use distributed transactions, like when implementing [distributed transactions](#) based on the Windows's DTC (Distributed Transaction Coordinator) with our "old friend" [MSMQ](#).

Going back to the initial issue and its example, if the service crashes after updating the database (right after the line of code with `_context.SaveChangesAsync();`) but before publishing the integration event, the overall system could become inconsistent and it will be more or less critical depending on the specific business operation you are dealing with.

As mentioned in the architecture section, you can have several approaches for dealing with this issue. Basically:

1. Using full [event sourcing](#) pattern
2. Using [transaction log mining](#)
3. Using a transactional table to store the integration events (Extending the local transaction)

Using the full event sourcing pattern (ES) is one of the best approaches if not the best, however, in many application scenarios you might not be able to implement a full event sourcing system. ES means that you'd be storing only domain events in your transactional database instead of current state data. Storing only domain events in your transactional database can have great benefits such as having the history of your system available and being able to know state of your system at any moment in the past. However, implementing a full event source system requires you to re-architect most of your system and introduces many other complexities and requirements. For example, you probably would want to use a database specifically made for event-sourcing, such as [Event Store](#), or a document-oriented database such as *Azure Document DB, MongoDB, Cassandra, CouchDB,* or *RavenDB*. ES is a great approach for this problem, but not the most straightforward unless you are already familiar with Event Sourcing.

The transaction log mining choice initially looks very transparent but you'd be coupled to your specific RDBMS transaction log, like the SQL Server transaction log or any other RDBMS. Being coupled to that is probably not desirable. Another drawback is that the low-level updates recorded in the transaction log might not be at the same level than your high-level integration events so the process of reverse engineer those transaction log operations can be difficult.

A simple and balanced approach would be a mix of a transactional database table and a simplified event sourcing pattern. You would use states such as a "*ready to publish the event*" state that you set

in the original event when you commit it into the integration events table, and then you would try to publish the event to the Event Bus (based on queues or any bus implementation). If the publish event action succeeds, then you would start another transaction in the source or original service and move the state from "*ready to publish the event*" to "*event already published*".

If the publish event action in the Event Bus fails, the data still won't be inconsistent within the original microservice because it is still marked as "*ready to publish the event*", and in regards to the rest of the services, it will be eventually consistent. You can always have background tasks/jobs checking the state of the transactions or integration events and if they find a "*ready to publish the event*" state, they can try to re-publish that event into the Event Bus.

Notice that with this approach you are persisting just the integration events per origin microservice, and only the events that you want to communicate to other microservices or external systems. On the contrary, in a full Event Sourcing system, you store all Domain Events as well.

Therefore, this approach can be considered as a very simplified event-sourcing system. You need a list of integration-events with their current state (*ready to publish* vs. *published*), but you only need to implement it for the integration events. In this case, you don't need to store all your domain data as events in the transactional database as you would in a full event sourcing system.

In terms of a specific implementation when you are already using a relational database is to use a transactional table to store your integration events being raised. The technique to achieve atomicity in your application is a two-step process based on local transactions. Basically, you will have an *IntegrationEvent* table within the same database where you have your domain entities. That especial table will work as your "life insurance" asset for achieving atomicity so you include persisted integration events into the same transactions that are committing your domain data.

Explaining the process step by step, the application begins a local database transaction, updates the state of your domain entities, inserts an event into the integration event table, and commits the transaction. You get that desired atomicity.

Then, for the next steps about publishing the events you have two choices.

1. Publish the integration event right after committing the transaction and use another local transaction to mark the events in the table as published. Then, use the table just as an artifact to track the integration events in case of issues in the remote microservices and do compensatory actions with those microservices.
2. Use the table as a "queue". So, a separate application thread or process queries the integration event table, publishes the events to the Event Bus, and then uses a local transaction to mark the events as published.

In image X-XX you can see the choice number 1 architecture diagram.
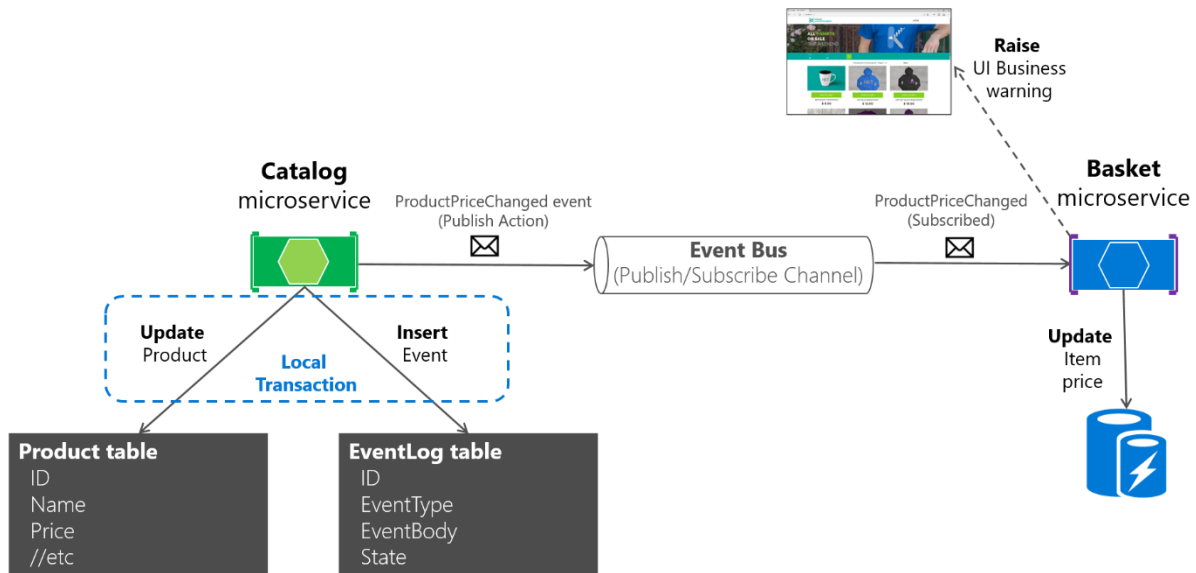
Figure X-XX. *Atomicity when publishing events to the* Event Bus

The approach in the previous diagram would be missing an additional worker microservice in charge of confirming the success of the published integration events and in case of failure it would need to read events from the table and re-publish.

On the other hand, if using a worker microservice, the process would be like in the image X-XX where there is an additional microservice/container and the table is the only single point of source when publishing the events.
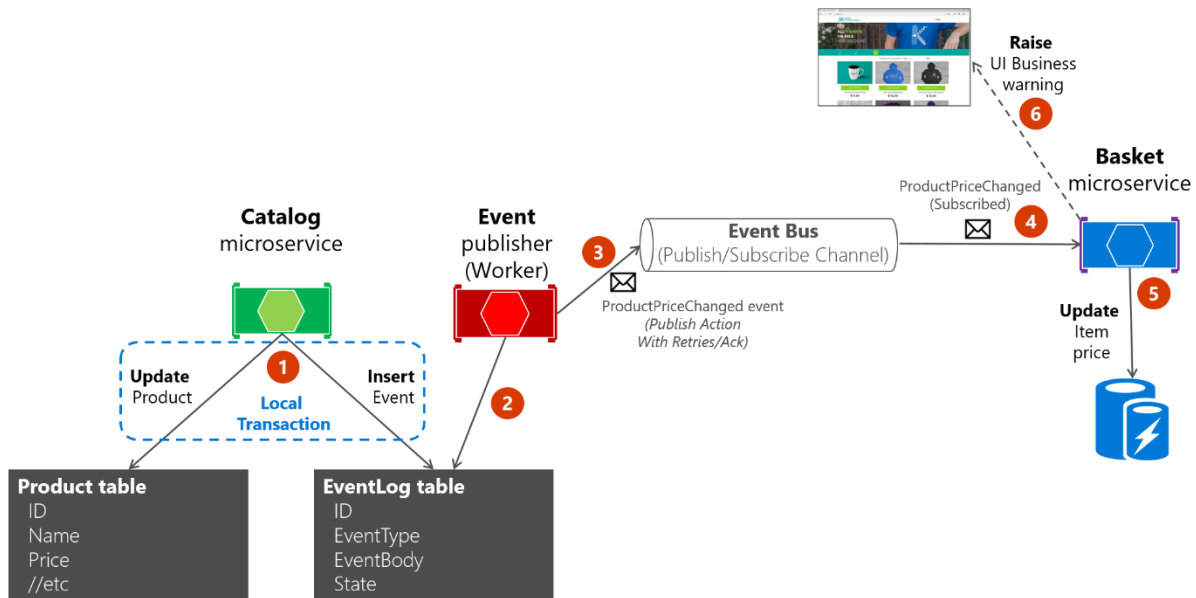
Figure X-XX. *Atomicity publishing events to the* Event Bus with a worker microservice

For simplicity, with no additional processes or workers and better clarity explaining the Event Bus, the *eShopOncontainers* sample uses the first choice, however, it might be incomplete in regards the

possible failure cases you can have. Therefore, as in the cloud you must embrace the fact that issues will happen eventually and retries logic and validation are needed, using the table as a queue could be more effective by having that table as a single point of source of events when publishing them through the event bus.

Implementing atomicity when publishing Integration Events through the Event Bus

The following code shows how you can create a single transaction involving multiple DbContexts, one DbContext related to the original data being updated, the second DbContext related to the IntegrationEventLog table being used.

Note that the transaction in the code snippet below won't be resilient if connections to the database have any issue at this moment which is possible in cloud systems like Azure SQL DB which might move databases across servers. For implementing resilient transactions acros multiple DbContexts, check the section "*Resilient Entity Framework Core Sql Connections*" at the end of this document.

For clarity reasons and being able to read it thorugh in a single step, the code snippet below shows the whole process in a single chunk of code. However, the final *eShopOnContainers* implementation might be slightly refactored and split into multiple classes so it is easier to maintain.

```
//Update Product from the Catalog microservice
//
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem productToUpdate)
{
    var catalogItem = await _catalogContext.CatalogItems.SingleOrDefaultAsync(
                                                    i => i.Id == productToUpdate.Id);
    if (catalogItem == null) return NotFound();

    bool raiseProductPriceChangedEvent = false;
    IntegrationEvent priceChangedEvent = null;
    if (catalogItem.Price != productToUpdate.Price) raiseProductPriceChangedEvent = true;

    if (raiseProductPriceChangedEvent) // Create event if price has changed
    {
        var oldPrice = catalogItem.Price;
        priceChangedEvent = new ProductPriceChangedIntegrationEvent(catalogItem.Id,
                                                    productToUpdate.Price,
                                                    oldPrice);

    }                              Integration Event to publish
    //Update current product
    catalogItem = productToUpdate;

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

        //Save to EventLog only if product price changed
        if(raiseProductPriceChangedEvent)
            await _integrationEventLogService.SaveEventAsync(priceChangedEvent);

        transaction.Commit();                Update to Integration Event Log table
    }
    //Publish to Event Bus only if product price changed
    if (raiseProductPriceChangedEvent)
    {                          Publish to Event Bus
        _eventBus.Publish(priceChangedEvent);
```

```
        await _integrationEventLogService.MarkEventAsPublishedAsync(priceChangedEvent);
    }
    return Ok();
}
```

Mark the event at the event log as "published"

The event log table is updated atomically with the original database operation with a local transaction against the same database, so if any of those operations fails, there would be an exception plus it would roll back any performed operation so it keeps consistent information. Plus, thanks to the EF Core.

Receiving messages from subscriptions: Event Handlers in receptor microservices

Once the receiver microservices are subscribed, you must implement the internal code of the Event Handlers (like a callback method) where you will receive the event messages and process them.

An Event handler first receives an Event instance from the Event Bus. Then, it locates the component to be processed related to that Integration Event, propagating and persisting the event as a change in state in the receptor microservice. For example, if a *ProductPriceUpdated* event was originated in the Catalog microservice then it will be handled in the Basket microservice by an specific Event Handler, as in the following code.

```
namespace Microsoft.eShopOnContainers.Services.Basket.API.IntegrationEvents.EventHandling
{
    public class ProductPriceChangedIntegrationEventHandler :
                              IIntegrationEventHandler<ProductPriceChangedIntegrationEvent>
    {
        private readonly IBasketRepository _repository;
        public ProductPriceChangedIntegrationEventHandler(IBasketRepository repository)
        {
            _repository = repository;
        }

        public async Task Handle(ProductPriceChangedIntegrationEvent @event)
        {
            var userIds = await _repository.GetUsers();
            foreach (var id in userIds)
            {
                var basket = await _repository.GetBasket(id);
                await UpdatePriceInBasketItems(@event.ProductId, @event.NewPrice, basket);
            }
        }

        private async Task UpdatePriceInBasketItems(int productId, decimal newPrice,
                                                    CustomerBasket basket)
        {
            var itemsToUpdate = basket?.Items?.Where(x => int.Parse(x.ProductId) ==
                                                          productId).ToList();
            if (itemsToUpdate != null)
            {
                foreach (var item in itemsToUpdate)
                {
                    if(item.UnitPrice != newPrice)
                    {
                        var originalPrice = item.UnitPrice;
                        item.UnitPrice = newPrice;
                        item.OldUnitPrice = originalPrice;
                    }
                }
            }
            await _repository.UpdateBasket(basket);
```

Integration Event Handle method

```
                }
            }
        }
    }
}
```

> Update the Basket because of the event info

The Event Handler needs to verify whether the product exists in any of the basket instances. It will also update the item price for each related basket line item and create an alert to be displayed as a warning message for the user about that price change, as in the following screenshot in image X-XX.

Figure X-XX. Detecting item price change in basket communicated by integration

Idempotency in update message events

An important aspect of update message events is that a failure at any point in the communication should cause the message to be retried. Otherwise a background task might try to publish an event that was already being published before, creating a race condition You need to make sure that the updates are either *idempotent* or that they provide enough information to ensure that you can detect a duplicate, discard it, and send back only one response.

Idempotency. As introduced in the discussion about Commands, idempotency is a characteristic of an operation which means an operation can be applied multiple times without changing the result. In a messaging environment, as when communicating events, an event is idempotent if it can be delivered multiple times without changing the result for the same recipient microservice. This may be necessary because of the nature of the event itself, or because of the way the system handles the event.

Message idempotency is important in any system that uses messaging, not just in systems that implement the Event Bus pattern.

Developing Microservices with .NET Core

In other words, *Idempotent* means that the operation can be performed multiple times, and beyond the first time it is performed the result is not changed. An example of an idempotent operation is a database script that inserts data into a table only if the data isn't already present. No matter how many times the script is executed beyond the first time the result will be the same. Idempotency such as this can also be necessary when working with message processing if the messages could potentially be processed more than once.

In some scenarios, it would be possible to design idempotent messages. For example, by using an event that says "set the product price to $25" rather than a message that says "add $5 to the product's price". You could safely process the first message multiple times, but not the second. Even in the first case, you might not want to process the first event, as the system could also have sent a newer price change event.

Another example would be an *OrderCompleted* event when propagating it to multiple subscribers, such as other microservices or even external systems like an ERP. It is important that any order created is propagated or updated to other systems just once even if there are several duplicated message events for the same original *OrderCompleted* event.

Even when an event will be broadcasted to many subscribers or recipients, in most cases it's good to have some kind of identity per event, so each event is processed only once per destination recipient.

Some message-processing will be inherently idempotent. For example, if a system generates image thumbnails of a larger file stored in BLOB storage it could be that it doesn't matter how many times the message is processed; the outcome is that the thumbnails are generated and they are the same every time. On the other hand, there are operations such as calling a payment gateway to charge a credit card that may not be idempotent at all. In these cases, you will need to look at your system and ensure that processing a message multiple times has the effect that you want and expect.

Idempotency in Event Bus and messaging
Honoring message idempotency   https://msdn.microsoft.com/en-us/library/jj591565.aspx

De-duplicating integration event messages

Making sure that message events are sent and processed just once per destination microservice or subscriber can be accomplished at different levels. You could make use of de-duplication feature offered by the messaging infrastructure you are using, or you could also implement custom logic in your destination event handlers. Having validations at both the transport level and the application level is probably your best bet.

De-duplicating message events at the EventHandler level

One way to do make sure that an event is processed just once by each destination recipient is by implementing certain logic when processing the message events at the Event Handlers.

De-duplicating messages when using RabbitMQ

According to [RabbitMQ documentation](), "In the event of network failure (or a node crashing), messages can be duplicated, and consumers must be prepared to handle them. If possible, the simplest way to handle this is to ensure that your consumers handle messages in an idempotent way rather than explicitly deal with deduplication.

If a message is delivered to a consumer and then requeued (because it was not acknowledged before the consumer connection dropped, for example) then RabbitMQ will set the `redelivered` flag on it when it is delivered again (whether to the same consumer or a different one). This is a hint that a consumer may have seen this message before (although that's not guaranteed, the message may have made it out of the broker but not into a consumer before the connection dropped). Conversely if the `redelivered` flag is not set then it is guaranteed that the message has not been seen before. Therefore, if a consumer finds it more expensive to de-duplicate messages or process them in an idempotent manner, it can do this only for messages with the `redelivered` flag set."

In any case, it is highly recommended to handle message event in an idempotent way at the event handler's level.

REFERENCES

References – Publish/subscribe, eventual consistency and Event Sourcing
Event Driven Messaging   http://soapatterns.org/design_patterns/event_driven_messaging
Publish/Subscribe channel   http://www.enterpriseintegrationpatterns.com/patterns/messaging/PublishSubscribeChannel.html
Communicating Between Bounded Contexts   https://msdn.microsoft.com/en-us/library/jj591572.aspx
Eventual Consistency   https://en.wikipedia.org/wiki/Eventual_consistency
Strategies for Integrating Bounded Contexts   http://culttt.com/2014/11/26/strategies-integrating-bounded-contexts/
Developing Transactional Microservices Using Aggregates, Event Sourcing and CQRS - Part 2   https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-2-richardson
Event Sourcing pattern   http://microservices.io/patterns/data/event-sourcing.html
Introducing Event Sourcing   https://msdn.microsoft.com/en-us/library/jj591559.aspx
Event Store database   https://geteventstore.com/
Publishing Events Using Local Transactions   https://dzone.com/articles/event-driven-data-management-for-microservices-1
The CAP Theorem   https://en.wikipedia.org/wiki/CAP_theorem   https://www.quora.com/What-Is-CAP-Theorem-1
Data Consistency Primer   https://msdn.microsoft.com/en-us/library/dn589800.aspx
The CAP Theorem: Why "Everything is Different" with the Cloud and Internet

https://blogs.msdn.microsoft.com/rickatmicrosoft/2013/01/03/the-cap-theorem-why-everything-is-different-with-the-cloud-and-internet/

CAP Twelve Years Later: How the "Rules" Have Changed

https://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

MSMQ Participating in External (DTC) Transactions

https://msdn.microsoft.com/en-us/library/ms978430.aspx#bdadotnetasync2_topic3c

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.

Unit tests are implemented based on Tests frameworks, like xUnit.net, MSTest, Moq, NUnit, etc. In the case of eShopOnContainers it is using XUnit.

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.

```csharp
[Fact]
public void Add_new_Order_raises_new_event()
{
    //Arrange
    var street = "fakeStreet";
    var city = "FakeCity";
    var state = "fakeState";
    //.. Other variables omitted for brevity

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

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

```
    }
```

Implementing Integration and Functional Tests 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 *TestServer* 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 instanciate 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
xUnit.net (Test framework) https://xunit.github.io/
MSTest (Test framework) https://msdn.microsoft.com/en-us/library/hh694602.aspx
Moq (Test framework) https://github.com/moq/moq
NUnit (Test framework) https://www.nunit.org/

Implementing Service Tests on a multi-container application

As introduced before, when testing multi-container applications, you 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.

Tackling business complexity in a microservice with Domain-Driven Design (DDD) and Command & Query Responsibility Segregation (CQRS) patterns

Most of the techniques or practices explained for simple data-driven microservices, such as how to implement an ASP.NET Core Web API service or how to expose Swagger metadata with Swashbuckle, are also applicable to the more advanced microservices implemented internally with DDD (Domain-Driven Design) patterns. This section is an extension of the previous sections, as most of the practices explained earlier also apply here.

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 domain experts with ever-changing business rules.

In order to set the context, this whole section focuses on the internal architecture, design and implementation of concrete microservices following more advanced patterns like the once defined in DDD and CQRS, as illustrated in figure X-XX.
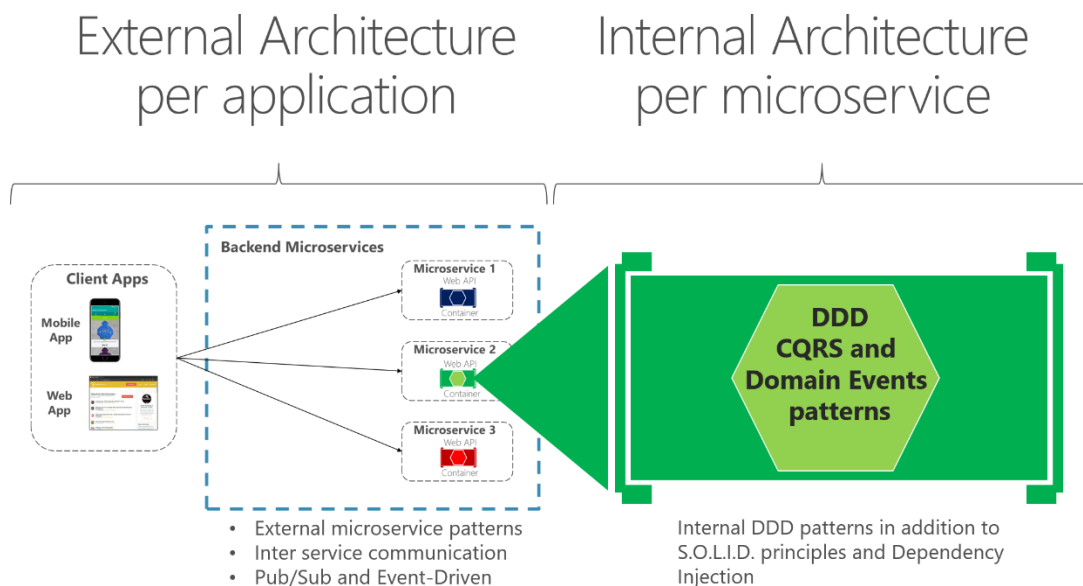
"DDD" vs. "DDD patterns": Not exactly the same

Make no mistake about it, this guidance is not in-depth coverage of DDD and CQRS. It is much less ambitious. This section is only covering how you can design with certain DDD and simplified CQRS 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.

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 with 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 that 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 the 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.

To learn DDD and how to apply it, you can start by reading books like [OBJ], 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., and many other literature from people like Vaughn Vernon, Jimmy Nilsson, Greg Young, Udi Dahan, Jimmy Bogard, and many other DDD/CQRS experts, but most of all, you need to try to learn how to apply DDD techniques from the conversations, whiteboarding, and domain modeling sessions with the experts of your concrete business domain.

References – Domain-Driven Design (DDD)
DDD (Domain-Driven Design)
http://domainlanguage.com/
http://martinfowler.com/tags/domain%20driven%20design.html
https://lostechies.com/jimmybogard/2010/02/04/strengthening-your-domain-a-primer/
DDD Books
Domain-Driven Design: Tackling Complexity in the Heart of Software – Eric Evans
Domain-Driven Design Reference: Definitions and Pattern Summaries - Eric Evans
Implementing Domain-Driven Design - Vaughn Vernon
Domain-Driven Design Distilled - Vaughn Vernon
Applying Domain-Driven Design and Patterns - Jimmy Nilsson
Domain-Driven Design Quickly
DDD Training
Domain-Driven Design Fundamentals – Julie Lerman and Steve Smith
http://bit.ly/PS-DDD

Applying simplified CQRS and DDD patterns within a microservice

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

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

Simplified CQRS and DDD microservice
High level design
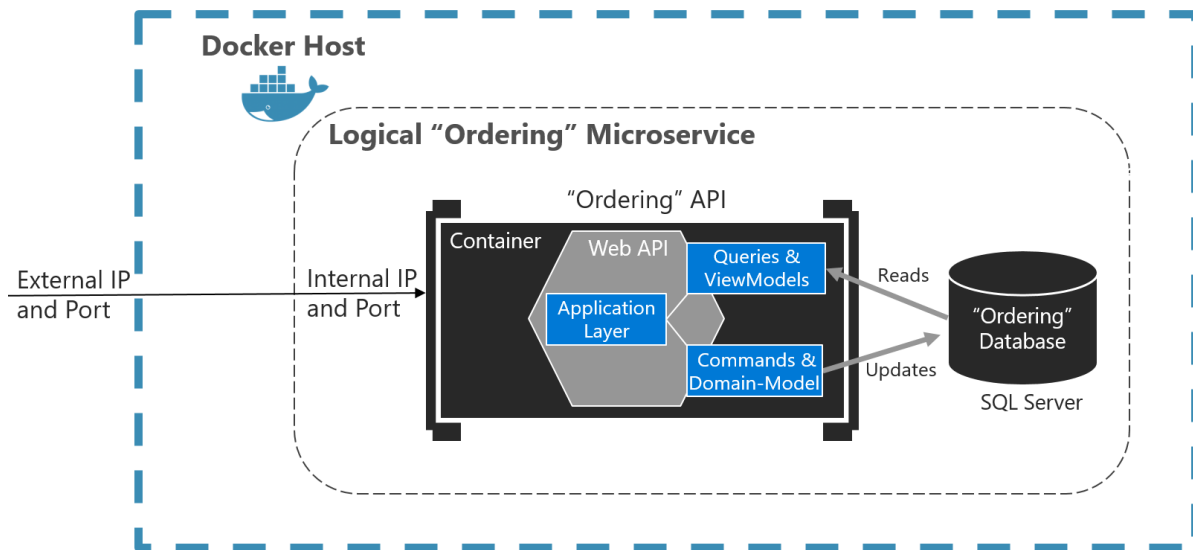
Figure X-XX. Simplified CQRS and DDD based microservice

The Application Layer can be the Web API itself. The important design decision here is that the microservice has split the Queries and ViewModels (Data models especially made for the client applications) from the Commands, Domain Model and transactions following a (CQRS or Command and Query Responsibility Segregation). This approach keeps the queries independent from restrictions and constraints coming from Domain-Driven Design patterns that only make sense to transactions and updates, as explained in later sections.

CQRS and CQS approaches in a DDD microservice

The related term CQS (Command Query Separation) was originally defined by Bertrand Meyer in his book "Object Oriented Software Construction". The basic idea is that you can divide a system's operations into two sharply separated categories:

- **Queries**: Return a result and do not change the state of the system (and are free of side effects).
- **Commands**: Change the state of a system.

CQS is a simple concept, it is about methods within the same object being either queries or commands. Each method either returns state or mutates state but not both. Even a single Repository pattern object can comply with CQS according with that definition. CQS could be It can be considered as a foundational principle for CQRS.

CQRS (Command and Query Responsibility Segregation) was introduced by Greg Young and also strongly promoted by Udi Dahan and other advocates. It is based on the CQS principle, although it is more detailed and can be considered a pattern based on commands and events plus optionally based on asynchronous messages. In many cases, CQRS is related to more advanced scenarios like having a different physical database for the Reads/Queries than for the Writes/Updates. Going even further, a more evolved CQRS system would implement Event-Sourcing (ES) for your Updates/Writes database, so you would only store events in the Domain Model instead of the current state data. However, and as mentioned, this is not the case of this approach used in this guidance where we are using the simplest CQRS approach which is just separating the queries from the commands.

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

The present microservice's design of this guide is based on CQRS principles but using the simplest approach, which is just separating the queries from the commands/updates and initially using the same database for both actions (which is also a possible approach in CQRS).

The essence of those patterns and the important point here is that *queries are idempotent*: no matter how many times you query a system, the state of that system won't change because of the querying. Therefore, you could use a different "reads-data-model" than the transactional logic "writes-domain-model".

On the other hand, commands (which will trigger transactions and data updates) are what impact your system, so the areas related to commands or updates is where you need to be careful when dealing with complexity and ever-changing business rules. Thus, 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 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 also use that approach 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 convoluted 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 for many reasons, compared to plain SQL data access as when using a Micro ORM.

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

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

CQRS and DDD patterns are not top-level architectures

It's important to highlight that CQRS and most DDD patterns (like DDD Layers or a Domain Model with Aggregates) are not architectural styles but only architectural 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 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 probably results in overengineering and poor use of developer resources. 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. It has 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 – CQRS
CQRS https://martinfowler.com/bliki/CQRS.html
CQS vs. CQRS (by Greg Young) http://codebetter.com/gregyoung/2009/08/13/command-query-separation/
CQRS Documents (Greg Young) https://cqrs.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

Implementing the Reads/Queries in a CQRS microservice

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

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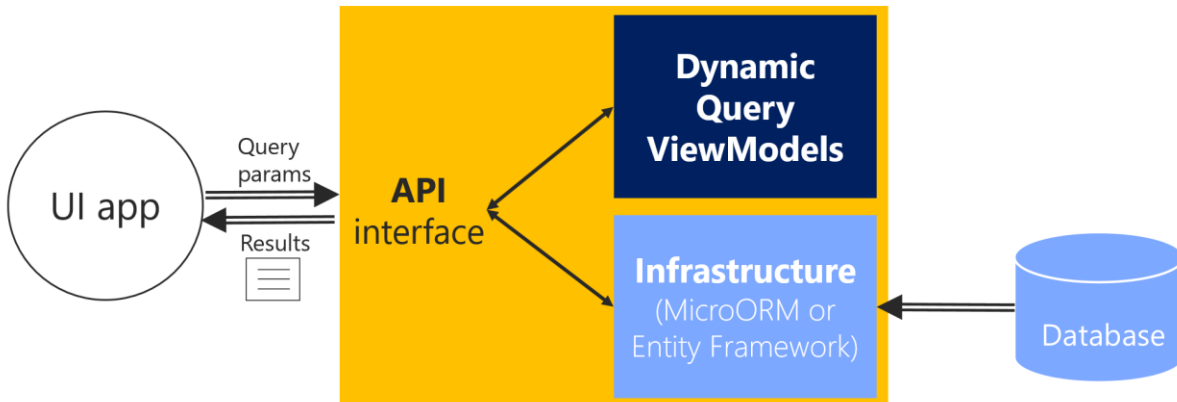
Figure X-XX. Simplest approach for queries in a CQRS

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

Since it is very simple approach, the required code for the "Queries side" like code using a MicroORM as Dapper can be implemented within the same Web API project as shown in figure X-XX where the queries are defined in the Ordering.API microservice project within the eShopOnContainers solution.
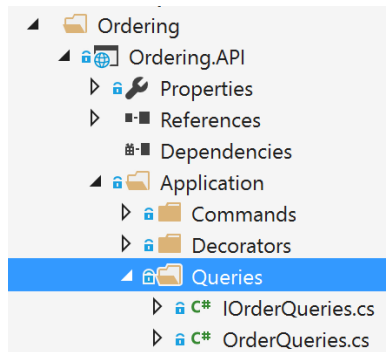
Figure X-XX. Queries in the Ordering microservice from eShopOnContainers

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

Since the queries are performed to obtain the data needed by the client applications, the model to return data to the client apps can be specifically made for them, and can be based on the data returned by the queries. Because of that, these specific models or DTOs (Data Transfer Object) 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 independent of the Domain Model, the Aggregates boundaries and constraints are completely ignored and you are free to query any table and column you might need. This approach provides great flexibility and productivity for the developers creating or updating the queries.

The ViewModels can be pre-defined in classes, or can also be created dynamically based on the queries performed, which is very agile for developers.

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

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

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

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

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

To use Dapper, you just need to install it through NuGet.
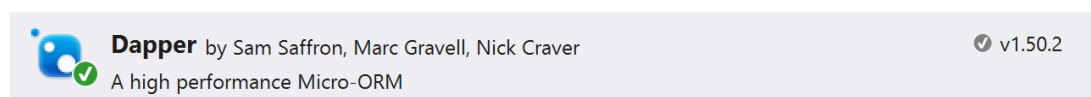
You will also need to add a `using` statement so your code has access to Dapper's extension methods.

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

Dynamic and static ViewModels

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

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

public class OrderQueries : IOrderQueries
{
  public async Task<dynamic> GetOrders()
  {
      using (var connection = new SqlConnection(_connectionString))
      {
          connection.Open();

          return await connection.QueryAsync<dynamic>(@"SELECT o.[Id] as
ordernumber,o.[OrderDate] as [date],os.[Name] as [status],SUM(oi.units*oi.unitprice) as total
              FROM [ordering].[Orders] o
              LEFT JOIN[ordering].[orderitems] oi ON  o.Id = oi.orderid
              LEFT JOIN[ordering].[orderstatus] os on o.StatusId = os.Id
              GROUP BY o.[Id], o.[OrderDate], os.[Name]");
      }
  }
}
```

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

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

References – Dapper
Dapper
https://github.com/StackExchange/dapper-dot-net
Data Points - Dapper, Entity Framework and Hybrid Apps (MSDN Mag. article by Julie Lerman)
https://msdn.microsoft.com/en-us/magazine/mt703432.aspx

Designing a Domain-Driven Design oriented microservice

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 areas of problems as Bounded Contexts (each bounded context correlates to a microservice), and emphasizes a common language to talk about these problems. It also suggests many technical concepts and patterns, like *Domain Entities* with rich-models (no [anemic-domain model](#)), *Value-Objects*, *Aggregate* and *Aggregate-Root* rules to support the internal implementation. 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 in the end, people tend to forget that the important part is to organize code artifacts in alignment with business problems and using the same common, ubiquitous language. Also, DDD approaches should be applied only when implementing complex microservices with ever-changing business rules. 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, for example writing Entity Framework Core code in an ASP.NET Core project.

When designing, and defining a microservice, where do you draw the boundaries? The Domain Driven Design patterns help 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. The components within those boundaries end up being your microservices. Microservices are about boundaries and so is DDD.

Keep the microservice's context boundaries relatively small

In regards to the 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 as small as possible if it makes sense. On the other hand, if you realize that your microservices are having too much 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 the emergent complexity. Distinct layers imply that translation must happen between some of the layers for information to propagate. For example, in a typical enterprise use case, an entity is loaded from the database, operated upon, persisted back to the database and information regarding the operation is returned to the user client app through a service/application layer, perhaps via a REST Web API service. The entity is contained within the domain layer and should not be forced into areas it doesn't belong, like in the presentation layer where a specific MVC view may require a user to enter information in several steps (basket, buying process, etc.). For instance, the user can enter the order's product item first, but the order might still have unspecified info about shipping or billing information.

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

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

The following figure xx-xx shows how that design is implemented in the *eShopOnContainers* app.
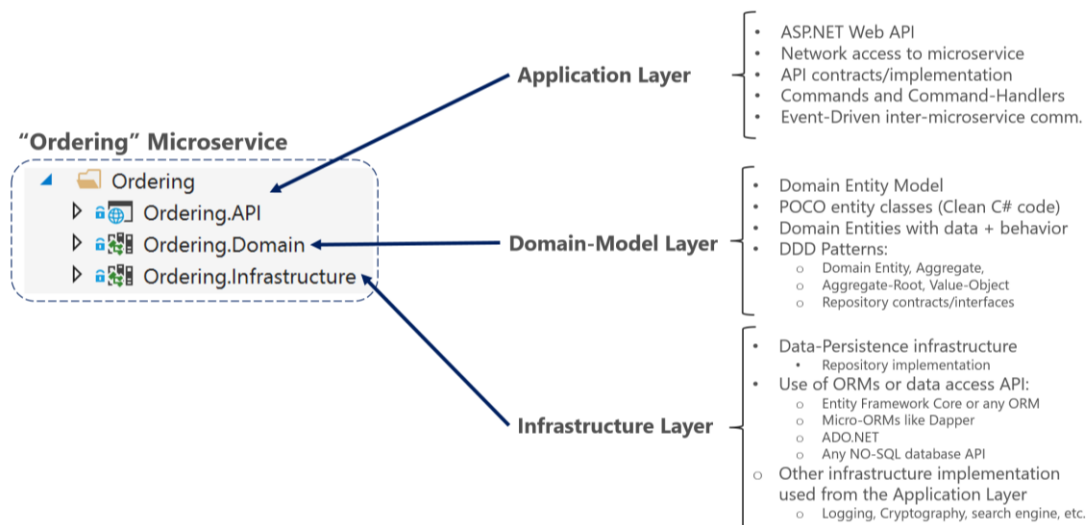
Figure X-XX. DDD Layers in the Ordering microservice from

A layer is simply a set of classes that you can 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 figure x-xx below about the Ordering.Domain layer library which only has dependencies with the .NET Core libraries.
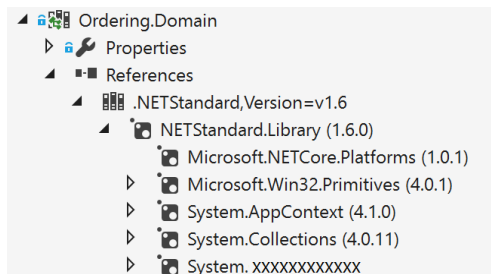
Figure X-XX. Layers implemented as libraries allow a better control of

Eric Evans's excellent book [Domain Driven Design](#) says the following about the Domain Model Layer and Application Layer.

*"**Domain Model Layer**: Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software."*

The Domain Layer is where the business is expressed. When implementing a microservice's Domain Model Layer in .NET, that layer would be coded as a class library with the domain entities that will capture data plus behavior (methods).

Following the [Persistence Ignorance](#) and the [Infrastructure Ignorance](#) principles, this layer must completely ignore the data persistence details. These persistence tasks should be performed by the infrastructure layer. Therefore, this layer should not take direct dependencies on the infrastructure, which means that an important rule should be that your Domain Model entity classes should be [POCO](#) (Plain-Old CLR Objects). Domain Entities should not have any direct dependency with any data-access infrastructure framework like Entity Framework or NHibernate or any other data-access framework. Ideally, your Domain entities should not derive or implement any type defined in any infrastructure framework.

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

You could, of course, also implement data access without an ORM, 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 Core Web API project which implements the microservice's interaction, remote network access and external Web APIs to be used from the UI or client apps. It includes queries if using a CQRS approach, commands accepted by the microservice, and even the event-driven communication between microservices. However, the ASP.NET Core Web API must not contain business rules or domain knowledge (especially domain rules in regards to transactions or updates), which should be owned by the Domain Model class library.

The Application Layer (in this case an ASP.NET Core Web API project) must only coordinate tasks and must not hold or define any domain state (domain model), but it will delegate the business rules execution to be run by the domain model classes themselves (Aggregate Roots and Domain Entities), which will ultimately update the data within those domain entities.

Basically, the application logic is where you implement all use cases that depend on a given front end, implementation for instance related to Web API or specific interfaces/contracts for your services front-end. The domain logic placed in the domain layer, however, is invariant to use cases and entirely reusable across all flavors of presentation and application layers you might have, and it must not depend on any infrastructure framework.

Infrastructure Layer: How the data initially held in domain entities in-memory will be persisted in databases or any other persistent store is a different matter. It will be implemented in the Infrastructure Layer, as when using Entity Framework Core code to implement the Repository pattern classes that use DBContext to persist data in a relational database.

In accordance with the previously mentioned Persistence Ignorance and the Infrastructure Ignorance principles, the Infrastructure Layer must not contaminate the Domain-Model layer. You must keep the Domain-Model entity classes agnostic from the infrastructure that you use to persist data (EF or any other framework) by not taking hard dependencies on frameworks. Your Domain-Model layer class library should have only your domain code, just POCO entity classes implementing the heart of your software completely decoupled from invasive infrastructure technologies.

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

Dependencies between Layers in a Domain-Driven Design service
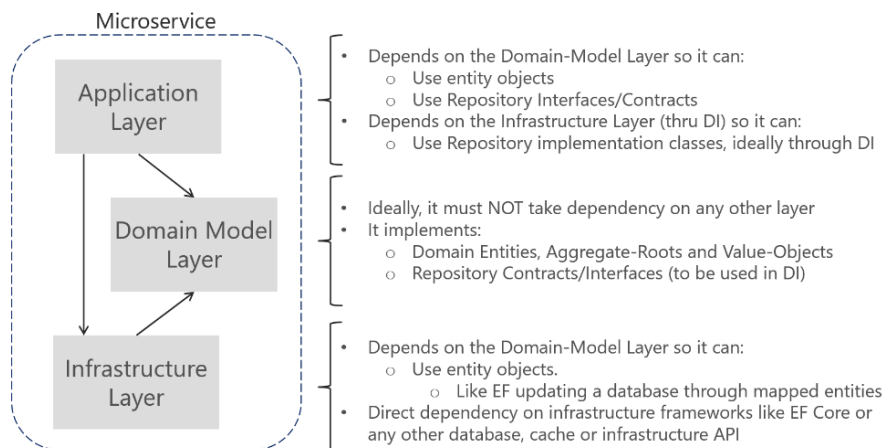
Figure X-XX. Dependencies between Layers in DDD

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

References – Persistence Ignorance principles
Persistence Ignorance principle   http://deviq.com/persistence-ignorance/   **Infrastructure Ignorance principle**   https://ayende.com/blog/3137/infrastructure-ignorance

Designing a microservice Domain-Model

One rich Domain Model per Microservice

Similar to DDD, each Bounded Context has to have its own Domain Model, and each microservice has to have and own its model, as introduced previously in this guide.

However, a Domain Model as defined in DDD is not just a data-model but a model that captures more than data entities. It also captures an entity's rules, behavior, business language and constraints of a specific domain's problem (Bounded Context). That special Rich Domain Model is what you should 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, continuity,* and persistence over time, not only by the attributes that comprise them.

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

Entities across multiple microservices or Bounded Contexts

The same identity 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 in the Ordering microservice (which shares its identity with the Customer entity) might have fewer attributes, because you only care about certain Buyer data related to the order process. The context of each microservice impacts the microservice's domain model.

Domain Entities must implement behavior in addition to data attributes

A Domain Entity in DDD must implement the domain logic related to the entity data (the object accessed in memory). For example, as part of an Order entity class you must have business logic and operations like adding an order item, data validation, or total calculation implemented as methods within the same entity class.

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

Domain Entity pattern
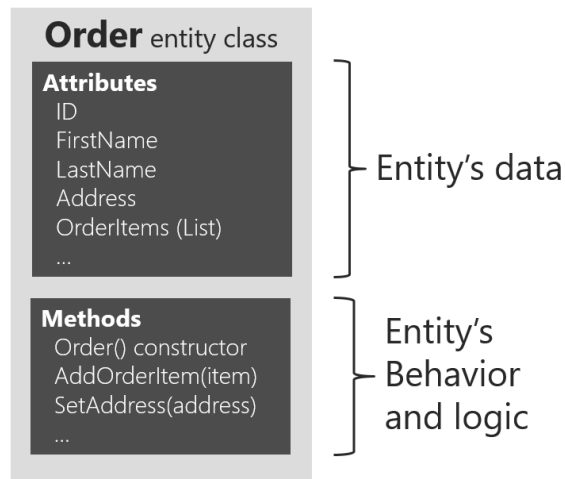
Figure <mark>X-XX</mark>. Example of Domain Entity Design implementing data plus behavior.

Of course, you could also have entities that do not implement any logic as part of the entity class, but this should only happen if that entity 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

As Martin Fowler described in [Anemic Domain Model](#), an Anemic Domain Model is basically a data model implemented as a collection of classes with attributes or properties. There are entity objects, most of them based on the nouns in the domain space, and these objects related to the domain's logic. The catch comes when you look at the behavior of those entity objects, and you realize that there is hardly any behavior in these objects, making them little more than a DTO data class with getters and setters. Of course, these data models will be used from a set of service objects (typically named Business Layer) which capture all the domain or business logic. The Business Layer sits on top of the data-model and use that data-model just for data.

The anemic domain model is just a procedural style design. Anemic entity objects are not real objects because they lack behavior (methods). They only hold data properties and thus completely miss the point of what object-oriented design is all about. By putting all the behavior out into service objects (Business Layer) you essentially end up with spaghetti code or [Transaction Scripts](#), and therefore you lose the advantages that a domain model provides.

Regardless, if your microservice (or Bounded Context) is very simple, data-driven or CRUD, the anemic domain model (entity objects with just data properties) might be good enough and it might not be worth implementing more complex DDD patterns.

Some people might say that the Anemic Domain Model is an anti-pattern. It really depends on what you are implementing. If the microservice you are creating is simple enough and CRUD, probably it is not an anti-pattern. However, if you need to tackle the complexity of a specific microservice's Domain which has a lot of ever-changing business rules, then the Anemic Domain Model might be an anti-pattern for that particular microservice or Bounded Context and designing it as a rich model with

entities containing data plus behavior as well as implementing additional DDD patterns (Aggregates, Value-Objects, etc.) might have huge benefits for the long-term success of such a microservice.

References – Domain Entity pattern , Domain Model and Anemic Domain Model
Domain Entity http://deviq.com/entity/ **The Domain Model** https://martinfowler.com/eaaCatalog/domainModel.html **The Anemic Domain Model** https://martinfowler.com/bliki/AnemicDomainModel.html

The Value-Object pattern

"*Many objects do not have conceptual identity. These objects describe certain characteristics of a thing.*" [Eric Evans]

There are many objects in a system that do not require an identity, whereas an Entity does.

The definition of Value-Object is: An object with no conceptual identity that describes a domain aspect. In short, these are objects that you instantiate to represent design elements which only concern you temporarily. You care about what they are, not who they are. Basic examples are numbers, strings, and such, but they also exist for higher level concepts like groups of attributes.

What may be an Entity in a microservice may not be an Entity in another microservice, because in the second case, Bounded Context might have a different meaning. For example, an address in some systems may not have an identity at all, since it may only represent a set of attributes of a person or company. That would be a Value-Object. That could be the case in an e-commerce application; the address may simply be a group of attributes of the customer's profile. In this case, the address doesn't have an identity per se and should be classified as a Value-Object pattern.

However, in other systems such as an application for an electric power utility company, the customer's address could be important for the business domain. Therefore, the address must have an identity so the billing system can be directly linked to the address. In this case, an address should be classified as a Domain Entity.

References – Value-Object pattern
• https://martinfowler.com/bliki/ValueObject.html • http://deviq.com/value-object/ • https://leanpub.com/tdd-ebook/read#leanpub-auto-value-objects • Value-Object in "Domain Driven Design" Book - Eric Evans.

The Aggregate pattern

A Domain-Model contains clusters of different data entities and processes that can control a significant area of functionality such as order fulfilment or inventory. A more finely grained DDD unit is the Aggregate which describes a cluster or group of entities and behaviors that can be treated as a single cohesive unit.

You usually define an Aggregate based on the transactions that you need. A classic example is an order that also contains a list of order items. An OrderItem will usually be an Entity, but it will be a

child entity within the Order Aggregate which will also contain the Order entity as its root-entity, typically called an Aggregate Root.

Identifying Aggregates can be hard. An aggregate is a group of objects that must be consistent together, but you can't just pick some objects and say "this is an aggregate". You start with modelling a Domain concept and thinking about the entities that need to be used within your most common transactions, and then you can identify the aggregates in your model. Thinking about transaction operations is probably the best way to identify aggregates.

Aggregate-Root or Root-Entity Pattern

An aggregate will be composed of at least one entity: the Aggregate Root (AR), also called root-entity or primary entity. Additionally, it can have multiple child entities and Value-Objects, with all entities and objects working together to implement required behavior and transactions.

The purpose of an Aggregate Root is to ensure the consistency of the aggregate; it should be the only entry point for updates to the aggregate through methods or operations placed in the Aggregate Root class. You should make changes to entities within the aggregate only via the Aggregate-Root. It is the aggregate's consistency guardian, taking into account all the invariants and consistency rules you might need to comply with in your aggregate. If you change a child entity or VO (Value Object) independently, the Aggregate Root cannot ensure the aggregate is in a valid state. It would be like a table with a loose leg. Maintaining consistency is the main purpose of the Aggregate Root.

In figure X-XX, you can see sample aggregates like the Buyer aggregate which contains a single entity (the Aggregate Root "Buyer"); the Order aggregate contains multiple entities and a Value-Object.
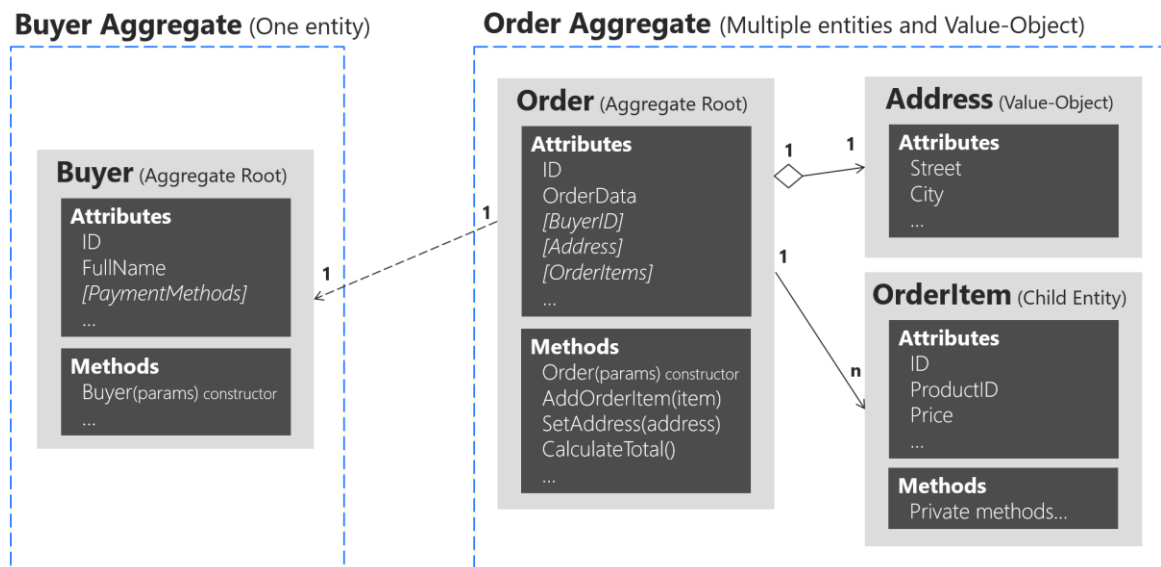
Figure X-XX. Aggregate pattern examples

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

figure 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 are a few articles and blog posts which drill down deeply into the subject and are very much recommended.

References – Aggregate related patterns
The Aggregate pattern http://deviq.com/aggregate-pattern/
Effective Aggregate Design - Part I: Modeling a Single Aggregate https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_1.pdf
Effective Aggregate Design - Part II: Making Aggregates Work Together https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_2.pdf
Effective Aggregate Design - Part III: Gaining Insight Through Discovery https://vaughnvernon.co/wordpress/wp-content/uploads/2014/10/DDD_COMMUNITY_ESSAY_AGGREGATES_PART_3.pdf
DDD Tactical Design Patterns https://www.codeproject.com/Articles/1164363/Domain-Driven-Design-Tactical-Design-Patterns-Part
Developing Transactional Microservices Using Aggregates https://www.infoq.com/articles/microservices-aggregates-events-cqrs-part-1-richardson

Implementing a microservice's Domain Model with .NET Core

In the previous section, the fundamental design principles and patterns to design a domain model were explained. Now it's time to drill down into possible ways to implement the Domain Model by using .NET Core (plain C# code) and EF Core. (EF Core model requirements only. You shouldn't have hard dependencies or references to EF Core in your Domain Model).

Domain Model structure in a .NET Core Standard Library

The 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. Of course, you are free to group your classes (Aggregate-Roots, Entities, Value-Objects and Repository Interfaces) in a different way.

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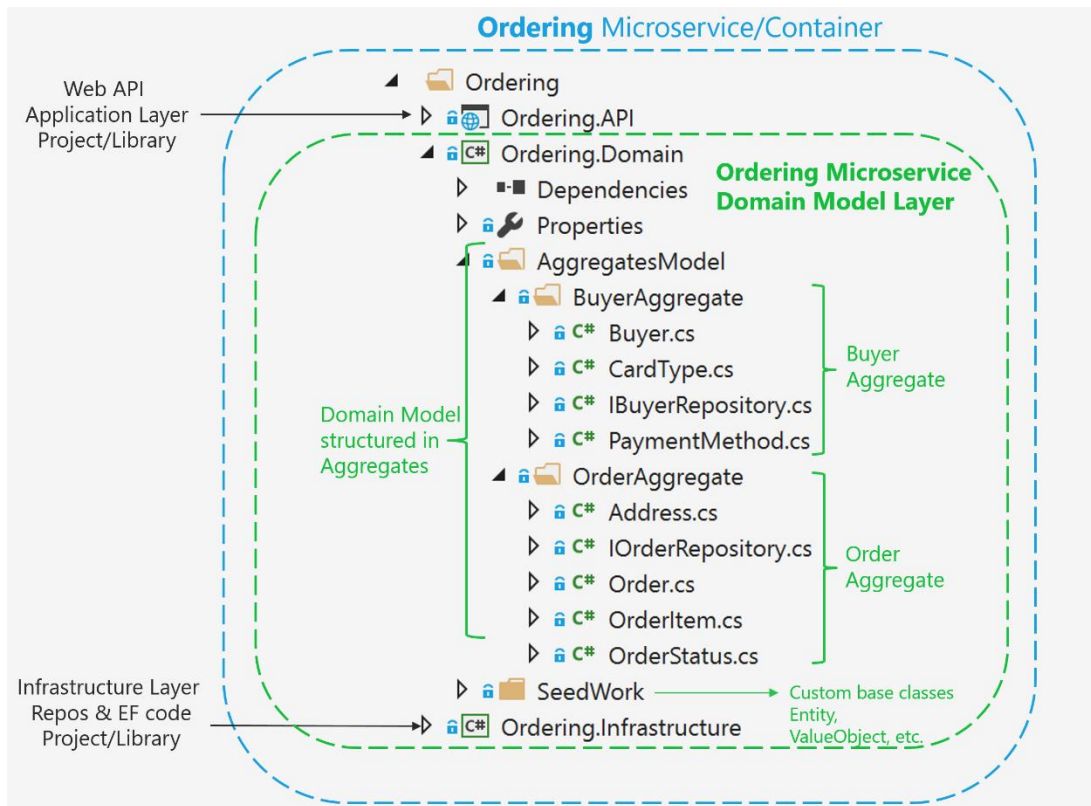
Figure X-XX. Domain Model structure for the Ordering

Additionally, in the Domain-Model layer you typically include the Repository contracts and interfaces that are the infrastructure requirements of your model, but not the infrastructure implementation of those repositories. They should be implemented outside of the domain model layer, in the infrastructure layer library.

You can also see a [SeedWork](#) folder which contains custom base classes that you can use as a base for your domain entities and value-objects, so you don't have to repeat redundant code in each domain's object class.

Structuring Aggregates in a .NET Standard Library

The concept of an aggregate refers to a cluster of domain objects grouped together to match transactional consistency. Those objects could be instances of entities (one of which is the Aggregate-Root or Root-entity) plus 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 of the following elements extracted from the eShopOnContainers Ordering microservice domain model, as shown in the figure X-XX.
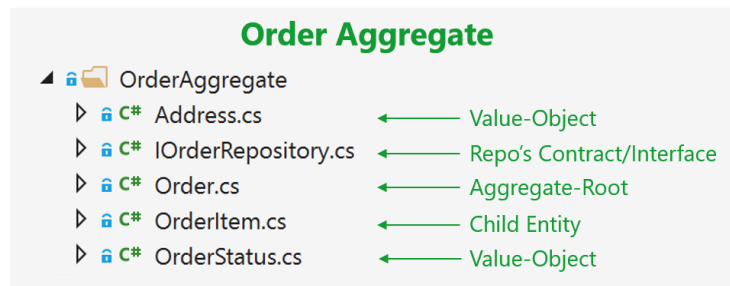
Figure X-XX. The "Order" aggregate in the VS solution

To see what kind of entity or object is contained in each class within an aggregate, you need to open its code and see how it is marked with your custom base classes or Interfaces implemented in the [SeedWork](#) folder.

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. In the following code, you can see that the Order class is defined as an entity and also as an aggregate root. Because the Order class is deriving from the custom base class Entity, it can re-use common code related to entities. Keep in mind that these base classes and interfaces are defined by you here in the domain model project, so it is your code, not infrastructure code from any ORM like EF.

Entity Framework Core 1.0

```csharp
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(productName,
                                 pictureUrl,
                                 unitPrice,
                                 discount,
                                 units)
        {
            //...
            // Domain Rules/Logic related to the OrderItem being added to the order
            //...
            OrderItem item = new OrderItem(this.Id, ProductId, ProductName,
                                           PictureUrl, UnitPrice, Discount, Units);
            OrderItems.Add(item);
        }

        //...
        // Additional methods with Domain Rules/Logic related to the Order Aggregate
        //...
```

The important fact to highlight about the above code snippet is that this is a Domain Entity implemented as a POCO class. It doesn't have any direct dependency to Entity Framework Core or any other infrastructure framework. It is as it should be, just your C# code implementing your Domain Model.

In addition to that, it is also decorated with an interface named `IAggregateRoot`. That interface is an empty interface, sometimes called a *marker interface*, which is used just to say that this entity class is also an Aggregate-Root or the root entity of the aggregate. That means that most of the code related to the consistency and business rules of the aggregate's entities should be implemented as methods in the Order Aggregate-Root class (for example, `AddOrderItem()` when adding an `OrderItem` to the Aggregate). You should not create or update `OrderItems` independently or directly; the `AggregateRoot` class must keep the control and consistency of any update operation against its child entities.

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

Wrong according to DDD patterns – Code at the application layer or Command Handlers

//My code in CommandHandlers or Web API controllers

//... (WRONG) Some code with business logic out of the Domain classes...

OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName, pictureUrl, unitPrice, discount, units);

//... (WRONG) Accessing the OrderItems colletion directly from the application layer or command handlers

myOrder.**OrderItems.Add**(myNewOrderItem);

//...

In this case, the `Add()` operation is purely an operation to add data, with direct access to the `OrderItems` collection. Therefore, most of the domain logic, rules or validations related to that operation with the child entities will be spread across the application layer (Command-Handlers and Web API controllers). Eventually you'll have spaghetti code, or a transactional script code implementation.

Following DDD patterns entities must not have public setters in any entity's property.

Going further, collections within the entity (like the order items) should be read-only properties (check the "`.AsReadOnly()`" pattern explained later) so you should be only able to update it from within the Aggregate root class methods.

As you can see in the code implementing the Order Aggregate-Root, all setters should be private, so any operation against the entity's data or its child entities will need to be performed through methods in the Aggregate-Root class. This will keep consistency in a more controlled and object-oriented way instead of doing a transactional script code implementation.

The following code snippet shows the proper code when adding an `OrderItem` to the Order aggregate.

Right according to DDD – Code at the application layer or Command Handlers

//My code in CommandHandlers or WebAPI controllers, only related to application stuff

// NO code here related to OrderItem's business logic

myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units**);**

// The code related to OrderItem params validations or domain rules should be <u>within</u> AddOrderItem()

//...

The important point here is that most of the validations or logic related to the creation of an `OrderItem` will be under the control of the Order aggregate-root, within the `AddOrderItem()` method, especially validations and logic related to other elements in the Aggregate. For instance, you might get the same product item as multiple `AddOrderItem(params)` invocations. In this method, you could check that out and consolidate the same product items in a single `OrderItem` with several units. Additionally, if there are different discount amounts but the product Id is the same, you would likely apply the higher discount. This principle applies to any other domain logic for the `OrderItem`.

In addition, the operation `new OrderItem(params)` will also be controlled and performed by the `AddOrderItem()` method from the Order aggregate-root, so most of the logic or validations related to that operation (especially if it impacts the consistency between other child entities) will be in a single place within the aggregate root. That is the ultimate purpose of the Aggregate Root pattern.

When using Entity Framework 1.1, a DDD entity can be better expressed because one of the new features of Entity Framework Core 1.1 is that it allows [mapping to fields](#) in addition to properties. This is extremely useful when protecting collections of child entities or value objects.

Now, you can use simple fields instead of properties and implement any update to the field collection through methods and making it read only through the "`.AsReadOnly()`" pattern.

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

Entity Framework Core 1.1 or later

```csharp
public class Order : Entity, IAggregateRoot //Entity is a custom base class with the Id
{
    // DDD Patterns comment
    // Using private fields, allowed since EF Core 1.1, is a much better encapsulation
    // aligned with DDD Aggregates and Domain Entities (Instead of properties and property collections)
    private bool _someOrderInternalState;
    private DateTime _orderDate;

    public Address Address { get; private set; }

    public Buyer Buyer { get; private set; }
    private int _buyerId;

    public OrderStatus OrderStatus { get; private set; }
    private int _orderStatusId;

    // DDD Patterns comment
    // Using a private collection field, better for DDD Aggregate's encapsulation
    // so OrderItems cannot be added from "outside the AggregateRoot" directly to the collection,
    // but only through the method OrderAggrergateRoot.AddOrderItem() which includes behaviour.
    private readonly List<OrderItem> _orderItems;

    public IEnumerable<OrderItem> OrderItems => _orderItems.AsReadOnly();
    // Using List<>.AsReadOnly()
    // This will create a read only wrapper around the private list so is protected against "external updates".
    // It's much cheaper than .ToList() because it will not have to copy all items in a new collection.
    // (Just one heap alloc for the wrapper instance)
    // https://msdn.microsoft.com/en-us/library/e78dcd75(v=vs.110).aspx

    public PaymentMethod PaymentMethod { get; private set; }
    private int _paymentMethodId;

    protected Order() { }

    public Order(int buyerId, int paymentMethodId, Address address)
    {
        _orderItems = new List<OrderItem>();
        _buyerId = buyerId;
        _paymentMethodId = paymentMethodId;
        _orderStatusId = OrderStatus.InProcess.Id;
        _orderDate = DateTime.UtcNow;
        Address = address;
    }

    // DDD Patterns comment
    // This Order AggregateRoot's method "AddOrderitem()" should be the only way to add Items to the Order,
    // so any behavior (discounts, etc.) and validations are controlled by the AggregateRoot
    // in order to maintain consistency between the whole Aggregate.
    public void AddOrderItem(int productId, string productName, decimal unitPrice,
decimal discount, string pictureUrl, int units = 1)
    {
        //...
        // Domain Rules/Logic related to the OrderItem being added to the order
        //...
        OrderItem item = new OrderItem(this.Id, productId, productName,
                                            pictureUrl, unitPrice, discount, units);
```

Developing Microservices with .NET Core

```
        OrderItems.Add(item);
    }

    //...
    // Additional methods with Domain Rules/Logic related to the Order Aggregate
    //...
}
```

Mapping properties with only get accessors to the fields in the database table

When using EF 1.0, within the DbContext, you need to map the properties that you defined with only get accessors to the actual fields in the database table. This is done with the HasField method of the PropertyBuilder.

Mapping Fields without Properties

With this new feature in EF Core 1.1 to map columns to fields, it's also possible to not use properties, and instead just to map columns from a table to fields. A common use for that would be private fields for any internal state that doesn't need to be accessed from outside the entity.

For example, the _someOrderInternalState_ field has no related property for either setter or getter. That field will also be calculated within the order's business logic and used from the order's methods, but it needs to be persisted in the database as well. So, in EF 1.1 there's a way to map a field without a related property to a column in the database. This is also explained in the Infrastructure Layer section of this guide.

References – Implementing Aggregates and Domain Entities
Modeling Aggregates with DDD and Entity Framework (By Vaughn Vernon)
https://vaughnvernon.co/?p=879 (Note that this is NOT Entity Framework Core)
Coding for Domain-Driven Design: Tips for Data-Focused Devs (Julie Lerman)
https://msdn.microsoft.com/en-us/magazine/dn342868.aspx
How to create fully encapsulated Domain Models (Udi Dahan)
http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/

The SeedWork or reusable base classes and interfaces for your Domain Model

As mentioned, in the solution folder you can also see a SeedWork folder which contains custom base classes that you can use as a base for your domain entities and value-objects, so you don't have to repeat redundant code in each domain's object class.

It's called SeedWork instead of framework because it is just a small subset of reusable classes, but it cannot be considered a framework. Seedwork is a term introduced by Martin Fowler, but you could also name that folder "Common" or any other name.

Figure <mark>X-XX</mark> shows the classes that form the SeedWork of the Domain Model in the Ordering microservice. It is just the custom "Entity" base class plus a few interfaces of the requirements asked to the implementation layer to have implemented. Those interfaces are also used through Dependency Injection from the application layer.
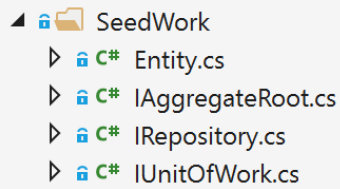
Figure <mark>X-XX</mark>. A sample Domain Model "Seedwork" with base classes and

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

The custom Entity base class

The following code is an example of an Entity base class where you can place code that can be used the same way by any Domain Entity, such as the entity Id, <u>equality operators</u>, etc.:

```
Entity Framework Core 1.1

    public abstract class Entity
    {

        int? _requestedHashCode;
        int _Id;

        public virtual  int Id
        {
            get
            {
                return _Id;
            }
            protected set
            {
                _Id = value;
            }
        }

        public bool IsTransient()
        {
            return this.Id == default(Int32);
        }

        public override bool Equals(object obj)
        {
            if (obj == null || !(obj is Entity))
                return false;

            if (Object.ReferenceEquals(this, obj))
                return true;

            Entity item = (Entity)obj;

            if (item.IsTransient() || this.IsTransient())
                return false;
```

```
            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);
        }

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

Repository contracts/interfaces placed in the Domain Model Layer

The Repository contracts are simply .NET interfaces that express the contract requirements of the Repositories to be used per each Aggregate. The Repositories themselves, with EF Core code or any other infrastructure dependencies and code, must not be implemented within the Domain Model; only the contracts or interfaces you demand to be implemented.

A pattern related to this practice (placing the Repository Interfaces in the Domain Layer) is the Separated Interface pattern defined by Martin Fowler as "*Use Separated Interface to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation*". Doing it that way, from the application layer (in this case, the Web API project for the microservice) when using Dependency Injection you will have a dependency on 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 implement the `OrderRepository` in the infrastructure layer library. In the current implementation of the application it just needs to add the order to the database, since queries are split following the CQS approach and updates to Orders are not implemented in this implementation.

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

```
    }

    public interface IRepository<T> where T : IAggregateRoot
    {
        IUnitOfWork UnitOfWork { get; }
    }
```

References – Repository Contracts
Separated Interface pattern (By Martin Fowler)
http://www.martinfowler.com/eaaCatalog/separatedInterface.html

Value Objects

"*Many objects do not have conceptual identity. These objects describe certain characteristics of a thing.*" [E.E.]

As shown in previos sections when drilling down on entities and aggregates, the identity is fundamental for the entities; however, there are many objects and data in a system that do not require such an identity and identity tracking.

The definition of Value Obects is: Objects that describe things; to be more accurate, an object with no conceptual identity that describes a domain aspect. In short, these are objects that we instantiate to represent design elements which only concern us temporarily. We care about what they are, not who they are. Basic examples are numbers, strings, etc. but they also exist in higher level concepts. For example, an "Address" in some systems/domains could be an entity because in that system an address is important as an identity, like in an electric power utility. But in most domain/systems, the "Address" can be simply a Value Object, a descriptive attribute of a company or person.

A Value Obects can also reference to other entities. For example, in an application that generates a Route about how to get from one point to another, that route would be a Value Obect (it would be a "snapshot" of points on how to go through an specific route, but this suggested route won't have an identity) even though internally it is referring to different entities (City, Roads, etc. if those were entities in that domain).

The following example shows a diagram of the Address Value Object within the Aggregate Order:

Value Object within Aggregate
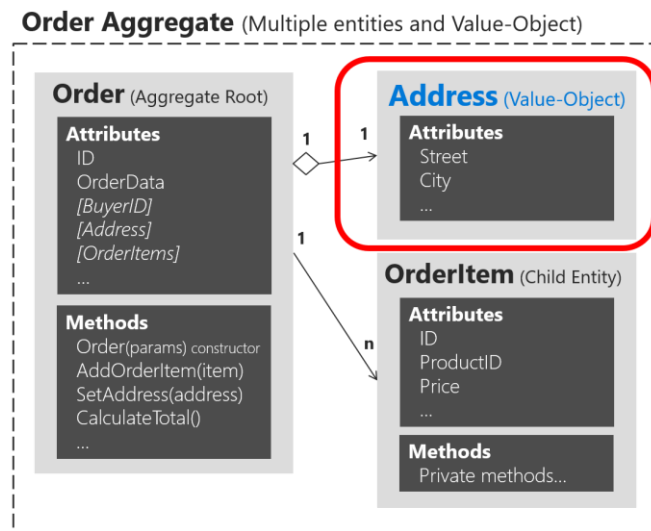
Figure X-XX. Value Object "Address" within the Order Agreggate

As shown in figure X-XX, an Entity is usually composed by multiple attributes. For example, the Order can be modeled as an Entity with an identity and composed internally by a set of attributes such as OrderId, date, Items, etc. But then, the address, which is simply a "complex value" composed by country, street, city, etc. must be modeled and treated as a Value Object.

Important characteristics of the Value Object

There are two main characteristics for the Value Objects:

- No Identity
- Immutable

The first characteristic was already introduced. In regards immutability, it is an important requirement. The values of a Value Object must be immutable once it is created. Therefore, at its construction, youy must provide the required values but you must not allow them to change during the object's lifetime.

Regarding performance, Value Objects allow you to perform certain "tricks", thanks to their immutable nature. This is especially true in systems where there may be thousands of VALUE-OBJECT instances with many coincidences of the same values. Their immutable nature would allow us to reuse them; they would be "interchangeable" objects, since their values are the same and they have no identity. This type of optimization can sometimes make a difference between software that runs slowly and another with good performance. Of course, all these recommendations depend on the application environment and deployment context.

Value Object implementation in C#

In terms of implementation, you can have a Value Object base class that with basic utility methods like equality based on comparison between all the attributes (since it must not be based on identity) and other fundamental characteristics, like the following base class used in the *Ordering* microservice from *eShopOnContainers*.

```csharp
public abstract class ValueObject
{
    protected static bool EqualOperator(ValueObject left, ValueObject right)
    {
        if (ReferenceEquals(left, null) ^ ReferenceEquals(right, null))
        {
            return false;
        }
        return ReferenceEquals(left, null) || left.Equals(right);
    }

    protected static bool NotEqualOperator(ValueObject left, ValueObject right)
    {
        return !(EqualOperator(left, right));
    }

    protected abstract IEnumerable<object> GetAtomicValues();

    public override bool Equals(object obj)
    {
        if (obj == null || obj.GetType() != GetType())
        {
            return false;
        }
        ValueObject other = (ValueObject)obj;
        IEnumerator<object> thisValues = GetAtomicValues().GetEnumerator();
        IEnumerator<object> otherValues = other.GetAtomicValues().GetEnumerator();
        while (thisValues.MoveNext() && otherValues.MoveNext())
        {
            if (ReferenceEquals(thisValues.Current, null) ^ ReferenceEquals(otherValues.Current,
                                                                             null))
            {
                return false;
            }
            if (thisValues.Current != null && !thisValues.Current.Equals(otherValues.Current))
            {
                return false;
            }
        }
        return !thisValues.MoveNext() && !otherValues.MoveNext();
    }

    //Other utilility methods
}
```

Then you can use it when implementing your actual Value Object, like the Address Value Object.

```csharp
public class Address : ValueObject
{
    public String Street { get; private set; }
    public String City { get; private set; }
    public String State { get; private set; }
    public String Country { get; private set; }
    public String ZipCode { get; private set; }
    public Address(string street, string city, string state,
                   string country, string zipcode)
    {
```

```
            Street = street;
            City = city;
            State = state;
            Country = country;
            ZipCode = zipcode;
        }

        protected override IEnumerable<object> GetAtomicValues()
        {
            yield return Street;
            yield return City;
            yield return State;
            yield return Country;
            yield return ZipCode;
        }
}
```

No Identity characterictic when using Entity Framework Core

A limitation when using EF Core is that in its current version (EF Core 1.1) you cannot use complex-types. Therefore, you must store your Value Object as an EF entity. However, you can hide its Id so you make clear that the identity is not important in the model of your value-object. The way you hide the Id is by using the ID as a "shadow property". Since that configuration is set up in the infrastructure level it will be transparent for your domain model and its infrastructure implementation could change in the future.

In *eShopOnContainers* that "hidden Id" needed by EF Core infrastructure is implemented in the following way in the DbContext level, using Fluent API, at the infrastructure project.

```
//Fluent API within the OrderingContext:DbContext at Ordering.Infrastructure project
void ConfigureAddress(EntityTypeBuilder<Address> addressConfiguration)
{
    addressConfiguration.ToTable("address", DEFAULT_SCHEMA);

    addressConfiguration.Property<int>("Id")
        .IsRequired();

    addressConfiguration.HasKey("Id");
}
```

Therefore, it is hidden from the domain model point of view and in the future, the Value Object infrastructure could also be implemented as a complex type or any other way.

References – Implementing Value-Objects in C#
Value Object pattern [Martin Fowler] https://martinfowler.com/bliki/ValueObject.html
Value Object pattern [Eric Evans book]
Value Object discussion [Vaughn Vernon book]
Shadow Properties in EF Core https://docs.microsoft.com/en-us/ef/core/modeling/shadow-properties
Complex types and/or value objects discussion for EF Core https://github.com/aspnet/EntityFramework/issues/246
Base Value Object class at eShopOnContainers https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/SeedWork/ValueObject.cs

Using Enumeration classes instead or Enums

Regular Enums are just fine in many scenarios, but quite dangerous in others. Specifically, using regular enums in your domain model can be a poor choice especially if those Enums are used to control the flow of your domain logic. Basically, poorly handled enums can infect code with fragility and tight couple the code with sentences of control like "if" or "switch" which are implementing knowledge about the semantics of each member of the enum that are spread throughout the code.

Enums are just an easy excuse for not creating the right abstractions. They are handy to use, simple to understand and readily available, but when using enums, pretty soon symptoms become externally visible. The code will arise many more bugs, unit tests will require a lot more of maintenance when you make a change because having hard-coded the flow's control and you will even need too much comments on every member of the enum to explain its ramifications.

Enums can be considered a code smell in many cases. The root cause of the Enum's disease is coupling and semantic diffusion. It forces you to sprinkle switch statements all over your code, thus violating the DRY Principle.

Additional problems derived from the usage of enums are:

- New enumeration values require many code changes across the application. Adding a new enumeration value can sometimes be painful, as there are lots of these switch statements around you need to modify.

- Behavior related to the enumeration gets scattered around the application

- Enumerations don't follow the Open-Closed Principle (SOLID)

The way to avoid that disease is by using encapsulation in the domain model by implementing Enumeration classes, as explained in the next section.

When Enums are okay to be used

When you have a fixed list of integer values which are not used to control your flow of instructions, then an enum could be perfectly valid. Things like gender (Male, Female, Undefined) or any other list of values as long as they are used just to store data and not as a data controlling the flow of your domain logic.

Implementing Enumeration classes

eShopOnContainers, within the Ordering microservice, provides a sample Enum base class implementation like the following.

```csharp
public abstract class Enumeration : IComparable
{
    public string Name { get; private set; }

    public int Id { get; private set; }
```

```csharp
    protected Enumeration()
    {
    }
    protected Enumeration(int id, string name)
    {
        Id = id;
        Name = name;
    }
    public override string ToString()
    {
        return Name;
    }
    public static IEnumerable<T> GetAll<T>() where T : Enumeration, new()
    {
        var type = typeof(T);
        var fields = type.GetTypeInfo().GetFields(BindingFlags.Public | BindingFlags.Static |
                                                    BindingFlags.DeclaredOnly);

        foreach (var info in fields)
        {
            var instance = new T();
            var locatedValue = info.GetValue(instance) as T;

            if (locatedValue != null)
            {
                yield return locatedValue;
            }
        }
    }
    public override bool Equals(object obj)
    {
        var otherValue = obj as Enumeration;

        if (otherValue == null)
        {
            return false;
        }

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

        return typeMatches && valueMatches;
    }

    //Other utility methods
}
```

Then you can use it as a type in any entity or value object like for the "CardType" enum class.

```csharp
public class CardType : Enumeration
{
    public static CardType Amex = new CardType(1, "Amex");
    public static CardType Visa = new CardType(2, "Visa");
    public static CardType MasterCard = new CardType(3, "MasterCard");

    protected CardType() { }
    public CardType(int id, string name)
        : base(id, name)
    {
    }
    public static IEnumerable<CardType> List()
    {
        return new[] { Amex, Visa, MasterCard };
    }
    //Other util methods
}
```

References – Enumeration classes
Why Enums are dangerous for your Domain Model
http://www.planetgeek.ch/2009/07/01/enums-are-evil/
https://codecraft.co/2012/10/29/how-enums-spread-disease-and-how-to-cure-it/
Implementing Enumeration classes in .NET
https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/
Base Enumeration class at eShopOnContainers
https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/SeedWork/Enumeration.cs
Sample "CardType" enumeration class at eShopOnContainers
https://github.com/dotnet/eShopOnContainers/blob/master/src/Services/Ordering/Ordering.Domain/AggregatesModel/BuyerAggregate/CardType.cs
Enum Alternatives in C#
http://ardalis.com/enum-alternatives-in-c

Designing Validations in the Domain Model Layer

From the DDD perspective, validation rules can be viewed as invariants. One of the central responsibilities of an aggregate is enforcement of invariants across state changes for all the entities within that aggregate.

Domain Entities should always be valid entities. There are a certain number of invariants for an object that should always be true. For example, an OrderItem object always has to have a quantity and a name. From that point of view, invariant enforcement is the responsibility of the domain entity itself (especially of the Aggregate-Root) and therefore an entity shouldn't be able to exist without being valid. Invariant rules are simply expressed as contracts, and exceptions or notifications are raised when they are violated.

The reasoning behind this is many bugs occur because objects are in a state they should never have been in. The following is a good and practical explanation from *Greg Young*:

"*Let's propose we now have a SendUserCreationEmailService that takes a UserProfile … how can we rationalize in that service that Name is not null? Do we check it again? Or more likely … you just don't bother to check and "hope for the best" you hope that someone bothered to validate it before sending it to you. Of course, using TDD one of the first tests we should be writing is that if I send a customer with a null name that it should raise an error. But once we start writing these kinds of tests over and over again we realize … 'wait if we never allowed name to become null we wouldn't have all of these tests'…*".

Implementing Validations in the Domain Model Layer

Validations are usually implemented in the Domain entities constructors, or within methods that can update the entity. There are multiple ways to implement validations, such as verifying data and raising exceptions if the validation fails. There are also more advanced patterns such as using the Specification pattern for validations, and the Notification pattern to return a collection of errors instead of returning an exception for each validation as it occurs.

Validating conditions and returning exceptions

The following code example shows the simplest approach to validation in a Domain Entity by raising an exception. In the references table at the end of this section you can see more advanced implementations based on the previously mentioned patterns and others.

```
public void SetAddress(Address address)
{
    _shippingAddress = address?? throw new ArgumentNullException(nameof(address));
}
```

A similar approach can be used in the entity's constructor, raising an exception to make sure that the entity is valid when you create it.

Using Validation attributes in the model based on Data Annotations

Another approach is to use validation attributes based on Data Annotations. Validation attributes provide a way to configure model validation, similar conceptually to validation on fields in database tables. This includes constraints such as assigning data types or required fields. Other types of validation include applying patterns to data to enforce business rules, such as a credit card number, phone number, or email address. Validation attributes make it easy to enforce requirements.

However, this approach might be too intrusive in a Domain-Driven Design Model, as it takes a dependency on ModelState.IsValid() from Microsoft.AspNetCore.Mvc.ModelState, which you must call from your MVC controllers. The model validation occurs prior to each controller action being invoked, and it is the controller method's responsibility to inspect ModelState.IsValid() and react appropriately. The decision to use it depends on how tightly coupled you'd like your model to be with that infrastructure:

```
using System.ComponentModel.DataAnnotations;
//Other usings
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 the use of exceptions in your entity's behavior methods, or by implementing the Specification and Notification patterns to enforce validation rules. Validation frameworks like Data Annotations in ASP.NET Core or any other

validation frameworks like FluentValidation carry a requirement to invoke the application framework. For example, when calling the ModelState.IsValid() method in Data Annotations, you need to invoke ASP.NET controllers.

It can make sense to use DataAnnotations at the application layer on ViewModel classes (instead of Domain Entities) that will accept input, to allow for model validation within the UI layer. However, this should not be done at the exclusion of validation within the domain model.

Validating Entities by implementing the Specification pattern and the Notification pattern

Finally, a more elaborate approach to implementing validations in the domain model is by implementing the Specification pattern in conjunction with the Notification pattern, as explained in some of the referenced articles below.

It is worth mentioning that you can also use just one of those patterns, for example validating manually with sentences of control but using the Notification pattern to be able to stack and return a list of validation errors.

Dealing with deferred validation in the domain

There are various approaches to deal with deferred validations in the domain, such as the Implementing Domain-Driven Design book by Vaughn Vernon, from pages 208-215.

References – Validations in the Domain Model
Model Validation in ASP.NET Core
https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation
Adding Validation in ASP.NET Core
https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation
Using the Notification Pattern to replace throwing exceptions with notification in validations
https://martinfowler.com/articles/replaceThrowWithNotification.html
Specification and Notification Patterns
https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns
Validation in Domain-Driven Design (DDD)
http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/
Domain Model Validation
http://colinjack.blogspot.com/2008/03/domain-model-validation.html
Validation in a DDD world
https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/

Client side validation (Validation in the Presentation Layers)

Even when the source of truth is the Domain Model and ultimately you must have validation at the Domain Model level, validation can still be handled at both the domain model level (server side) and the client side.

Client side validation is a great convenience for users. It saves time they would otherwise spend waiting for a round-trip to the server that might return validation errors. In business terms, even a few fractions of seconds multiplied hundreds of times each day adds up to a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

Just as the view model and the domain model are different, view model validation and domain validation might be similar but serve a different purpose. If you're concerned about being DRY (the "Don't Repeat Yourself" principle), consider that in this case code reuse might also mean coupling, and in enterprise applications it is more important not to couple the server side to the client side than to follow the DRY principle.

You could also validate your commands or input DTOs in the server side code, especially if your system doesn't have a client UI application, for example, if you are only creating a public API. If you have a client application, from a UX perspective, it is best to be proactive and not allow the user to type in stuff that makes no sense.

Therefore, in the client side code you will typically be validating the ViewModels in the client app. You could also validate the client output DTOs or commands to be sent to the server before you send them to the services.

The implementation of client side validation depends on what kind of client application you are building. It will be different if you are validating data in a web MVC web application with most of the code in .NET, or a SPA web app with that validation being coded in JavaScript or TypeScript, or a mobile app coded with Xamarin and C#.

Below are a few references for various types of client apps and technologies.

References – Validation in the Client side (Presentation Layer apps)
Validation in Xamarin mobile apps
https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/
https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/
Validation in ASP.NET Core apps
https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation
Validation in SPA web apps (Angular 2 / TypeScript / Javascript)
https://scotch.io/tutorials/angular-2-form-validation
https://angular.io/docs/ts/latest/cookbook/form-validation.html
http://breeze.github.io/doc-js/validation.html

In summary, the following are the most important concepts in regards to validation:

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

If you think that an entity needs to enter an invalid state, consider using a different object model, for example, using a temporary DTO until you create the final domain entity.

If you need to create several related objects, such as an aggregate, and they are only valid once all of them have been created, consider using the Factory pattern for this purpose.

Validation frameworks are best used in specific layers such as the presentation layer or the application/service layer, but usually not in the Domain Layer, as you would need to take a strong dependency on an infrastructure framework.

It is easier to duplicate validation logic than to keep it consistent across application layers, and in many cases having redundant validation in the client side is good, as you can be proactive.

Domain Events

Use domain events to explicitly implement side effects of changes within your domain.

In other words, and using DDD lingo, use domain events to explicitly implement side effects across multiple *aggregates*. Optionally, for better scalability and less impact in database locks, use eventual consistency between aggregates within the same domain.

What is a Domain Event?

An event is "*something that has happened in the past*". A domain event is, logically, something that happened in a particular domain and you wish other parts of the same domain could be aware and react based of that.

An important benefit from domain events is that side effects, after something happened in a domain, can be expressed explicitly instead of implicitly. For example, if you were using just Entity Framework and entities or even aggregates, if there is a change to the side effects of a use case, it will be implicit concept implemented by code after something happened. Sometimes you don't know if that side effect is part of the main operation or if it is really a side effect. When using domain events, it makes the concept explicit and part of the Ubiquitous Language; in the *eShopOnContainers* application, for example, creating an order is not just about that order, it updates or even creates a Buyer aggregate originated from the original user, because the user is not a buyer until and after he has bought. If using domain events, we can explicitly express that domain rule based on the ubiquitous language provided by the domain/business experts.

Domain events are partially similar to messaging-style events, with one important difference. With true messaging, queuing and a service bus, a message is fired and always handled asynchronously and communicated across processes and machines. This is useful for integrating multiple Bounded Contexts, microservices or even different applications. However, with domain events, you want to raise an event from the domain operation you are currently running but you want any side effects of the domain event to occur within the same domain.

Independently of the chosen implementation, the domain events and their side effects (the actions triggered afterwards that are managed by event-handlers) should occur almost immediately, usually in-process, and within the same domain.

Thus, domain events could be synchronous or asynchronous. Integration events, however, should always be asynchronous.

Domain Events vs. Integration Events

Semantically, domain and integration events are the same thing: notifications about something that just happened. However, their implementation can be different. Domain Events are just messages pushed to a Domain Event Dispatcher, which could be implemented as an in-memory mediator based on an IoC container or any other method.

On the other hand, the purpose of Integration events is to propagate committed transactions and updates to additional sub-systems, whether they are other microservices, Bounded Contexts or even external applications. Hence, they should occur only if the entity is successfully persisted, since in many scenarios if this fails, the entire operation effectively never happened.

In addition, and as mentioned, integration events must be based on asynchronous communication between multiple microservices (other Bounded Contexts) or even external systems/applications. Thus, under the Event Bus interface needs some infrastructure that allows inter-process and distributed communication between potentially remote services. It can be based on a commercial service bus, queues, a shared database used as a mailbox, or any other distributed and ideally push based messaging system.

Domain Events as a preferred way to trigger side effects across multiple aggregates within the same domain

If executing a command related to one aggregate instance requires additional domain rules to be run on one or more additional aggregates, you should design and implement those side effects to be triggered by domain events.

As shown in the image X-XX, and as one of the most important use cases, a domain event should be used to propagate state changes across multiple aggregates within the same domain model.
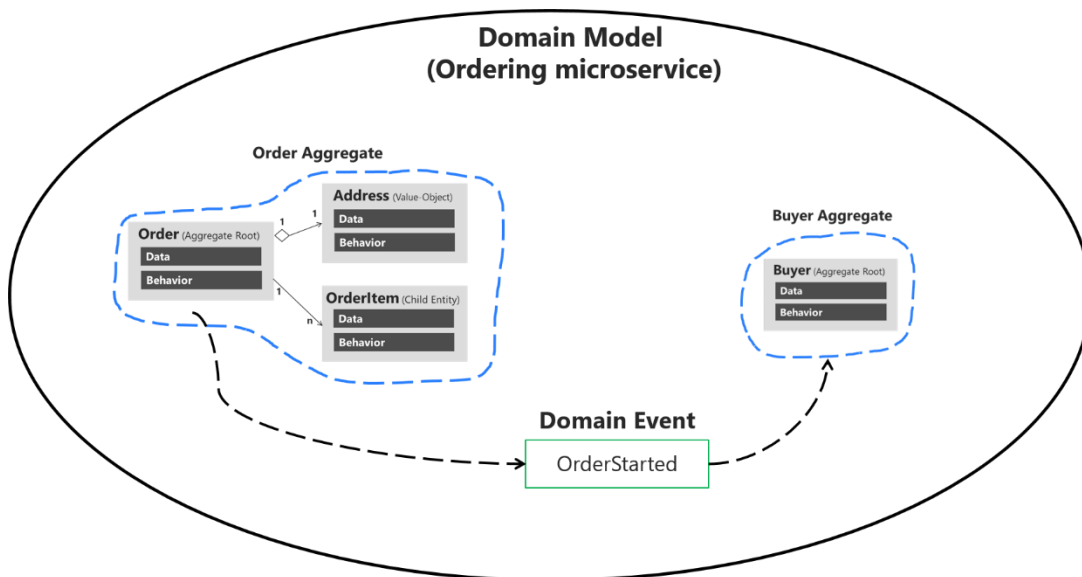
Figure X-XX. Domain Events to enforce consistency between multiple aggregates within the same domain

In the example above, the domain event "OrderStarted" might trigger a Buyer creation (if it doesn't exist) based on the original user's data when the user initiates an order. A buyer, in the ordering microservice, will be created based on the original user info from the identity microservice (info provided in the CreateOrderCommand). But the domain event is generated by the Order aggregate when it is created in the first place.

Alternately, you can also have the Aggregate Root subscribed for events raised by members of its Aggregate (child entities). For instance, each OrderItem child entity could be raising an event when the item price is higher than some amount or when the product item amount is too high, then having the aggregate root to receive those events and make any kind of global calculus or aggregation.

It is important to highlight that this event based communication is not implemented directly within the aggregates but you need to implement domain event handlers. Doing so, you could have any number of handlers triggering actions when a domain event happens.

Domain events can also be used to simply trigger an open number of application actions when that event happens. For instance, when the order is started, we might also want publish an "Integration Event" into an Event Bus and finally handled to propagate that info to other microservices or to send an email to the buyer/user saying that the order process has started. That action is not really related to any other aggregate, it is only a simple application action, but since it has to be performed *after* the transaction is committed, it is safer to use an integration event for that which are raised/published to any Event Bus only after the original transaction is performed. Therefore, this is a sample case of "connecting a Domain Event to an Integration Event" and publishing the Integration Event for "external actions" or to other microservices. Integration Events and the Event Bus are in a different subject, as introduced previosly.

That "open number of actions" to be executed when a domain event happens is the key point. Eventually, the actions and rules in the domain and application will be growing. The complexity or number of actions "when something happens" will be growing and if your code is coupled with "glue", like just instantiating objects, every time you need to add a new action you will need to change the original code. At that moment, you could be introducing new bugs because with each new requirement you would need to change the original code flow which is going against the [Open/Close principle](#) from [S.O.L.I.D.](#). Not only that, the original class that was orchestrating the operations will be growing and growing which is going against the [Single Responsibility Principle (SRP)](#).

On the other hand, if you use domain events, you can create a fine-grained and decoupled implementation by segregating responsibilities like in the following approach:

1. Send Command (CreateOrderCommand)
2. Command Handler
   - o   Single Aggregate transaction
   - o   Raise Domain Event (like OrderStarted)
3. Handle (within the current process) an open number of side effects in multiple aggregates or application actions
   - o   Verify or create buyer and payment method
   - o   Create and send a related integration event to the Event Bus to propagate states across microservices or trigger external actions like sending an email to the buyer
   - o   Other side effects

As shown in the following image X-XX, starting from the same domain event you can handle multiple actions related to other aggregates in the domain or additional application actions you need to perform across microservices connecting with Integration Events and the Event Bus.
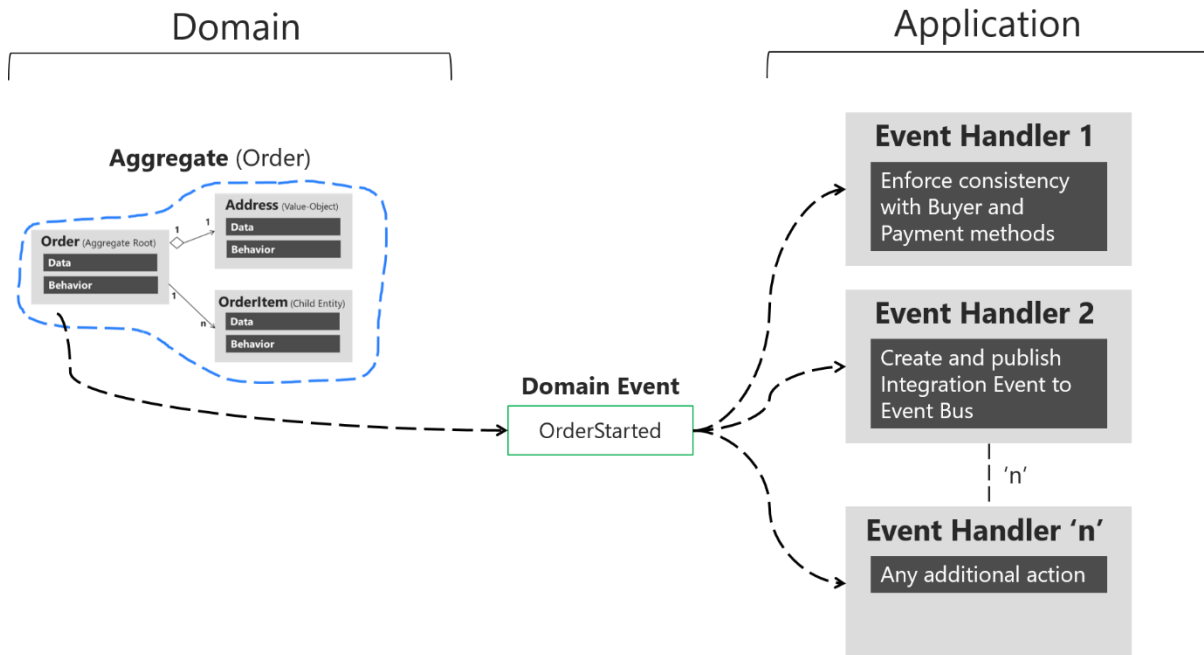
Figure X-XX. Handling multiple actions per domain

The event handlers are typically placed at the application layer as you will be using specific infrastructure objects like Repositories, or any application API for the microservice's behavior. From that sense, event handlers are similar to command handlers, so both are part of the application layer. The important difference is that a command should be processed just once. A domain event could be processed cero or 'n' times targeting multiple purposes.

Having the possibility of an open number of handlers per domain event would allow you to add many more domain rules without impacting/changing your current code. For instance, adding the following domain/business rule would be as easy as adding one or several new handlers for the following event:

"*When a customer's total amount purchased in the store (including any number of orders) exceeds $6,000, apply a 10% off discount to every new order and notify the customer with an email about that discount for future orders*"

Implementing Domain Events

How to implement a domain event

In terms of C# code implementation, a domain event is simply a data-holding structure or class, like a DTO (Data Transfer Object) with all the information related to what just happened in the domain, like in the following code.

Regarding the ubiquitous language to be used, since an event is "*something that happened in the past*" it is very important that the class name of *the event must be represented as a verb in the past tense* such as `OrderStartedDomainEvent,`or `OrderShippedDomainEvent.` For example, the following domain event is how it is implemented in the Ordering microservice at the eShopOnContainers application.

```
public class OrderStartedDomainEvent : IAsyncNotification
```

```
{
    public int CardTypeId { get; private set; }
    public string CardNumber { get; private set; }
    public string CardSecurityNumber { get; private set; }
    public string CardHolderName { get; private set; }
    public DateTime CardExpiration { get; private set; }
    public Order Order { get; private set; }

    public OrderStartedDomainEvent(Order order,
        int cardTypeId, string cardNumber,
        string cardSecurityNumber, string cardHolderName,
        DateTime cardExpiration)
    {
        Order = order;
        CardTypeId = cardTypeId;
        CardNumber = cardNumber;
        CardSecurityNumber = cardSecurityNumber;
        CardHolderName = cardHolderName;
        CardExpiration = cardExpiration;
    }
}
```

Basically, it is a class that holds all the data related to the OrderStarted event.

Events must be immutable. An important characteristic of events is that since an event is something that happened in the past, it shouldn't change, therefore it must be an *immutable class*, as you can notice in the previous code where the properties are read only from the outside of the object and the only way to update the object is through the constructor when you actually create the event object.

Raising domain events

The next question you might have is, "*ok, this is cool, but how do I raise a domain event so it reaches its related event handlers?*". Well, you could choose between multiple techniques or approaches for that.

Udi Dahan originally proposed in several related posts like Domain Events – Take 2, to use a static class for managing and raising the events, like a static class named *DomainEvents* which would raise domain events immediately when calling the DomainEvents.Raise(Event myEvent).

Jimmy Bogard also wrote a good post following a similar approach at Strengthening your domain: Domain Events.

However, when the domain events class is static, it also dispatches to handlers immediately. This makes testing and debugging more difficult because the event handlers with the side effects logic will be executed immediately right after raising the event. When you are testing and debugging you would want to focus and run just what's happening on the current aggregate classes instead of suddenly being redirected to other event handlers running side effects related to other aggregates or application logic. This is why other evolved approaches appeared, as explained in the next section.

The deferred approach for raising and dispatching events

Instead of dispatching to a domain event handler immediately, a better approach is to store/add the domain events in a collection and *right after or before* committing the transaction (like with

SaveChanges() in EF), dispatch those domain events. That approach was neatly described also by *Jimmy Bogard* at the "[A better domain events pattern post](#)".

Deciding if you send the domain events right before or after commiting the transaction is very important as depending on that you will include the side effects as part of the same transaction or in different transactions. In the last case, you would need to deal with eventual consistency implementations. This topic is precisely discussed in the next section.

The deferred approach is what the reference application [eShopOnContainers](#) uses. First, you add/store the events happening in your entities into a collection or list of events per entity. That list would be part of the entity object, better if coming from your base entity class, as shown in the code below.

```csharp
public abstract class Entity
{
    //...
    private List<IAsyncNotification> _domainEvents;
    public List<IAsyncNotification> DomainEvents => _domainEvents;
    public void AddDomainEvent(IAsyncNotification eventItem)
    {
        _domainEvents = _domainEvents ?? new List<IAsyncNotification>();
        _domainEvents.Add(eventItem);
    }
    public void RemoveDomainEvent(IAsyncNotification eventItem)
    {
        if (_domainEvents is null) return;
        _domainEvents.Remove(eventItem);
    }
    //...
}
```

Thus, whenever you want to raise an event, you would just add it to the event collection like in the following code to be placed within any aggregate entity method.

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

this.AddDomainEvent(orderStartedDomainEvent);
```

Notice that the method "AddDomainEvent" the only thing is doing is "adding an event to the list", nothing more. It is still not reaching the event handler.

Later on, when committing the transaction into the database is when you really want to dispatch the events. If using Entity Framework Core, that means at the "SaveChanges" method level of your EF DbContext, as in the following code.

```csharp
//EF Core DbContext
public class OrderingContext : DbContext, IUnitOfWork
{
//...
  public async Task<int> SaveEntitiesAsync()
  {
      // Dispatch Domain Events collection.
    // Choices:
    // A) Right BEFORE committing data (EF SaveChanges) into the DB will make a single transaction including
    // side effects from the domain event handlers which are using the same DbContext with Scope lifetime
    // B) Right AFTER committing data (EF SaveChanges) into the DB. will make multiple transactions.
    // You will need to handle eventual consistency and compensatory actions in case of failures.
```

```
        await _mediator.DispatchDomainEventsAsync(this);

        // After executing this line all the changes (from the Command Handler and Domain Event Handlers)
        // performed thought the DbContext will be commited
        var result = await base.SaveChangesAsync();

    }
}
```

With that code, you dispatch the entity events to their respective event handlers but you decouple the raising of a domain event (a simple add in memory) from dispatching to an event handler.

In addition to that, depending on what kind of dispatcher you are using, you could be dispatching the events synchronously or asynchronously.

Single transaction across aggregates vs. eventual consistency across aggregates

This is an arguable topic. Many DDD authors like *Eric Evans*, *Vaughn Vernon* and others defend the rule of "1 Transaction = 1 Aggregate" and therefore, eventual consistency across aggregates, for instance:

E.E. DDD p128: *Any rule that spans AGGREGATES will not be expected to be up-to-date at all times. Through event processing, batch processing, or other update mechanisms, other dependencies can be resolved within some specific time.*

V.V. [Effective Aggregate Design. Part II: Making Aggregates Work Together. p9](): *…Thus, if executing a command on one aggregate instance requires that additional business rules execute on one or more aggregates, use eventual consistency… …There is a practical way to support eventual consistency in a DDD model. An aggregate method publishes a domain event that is in time delivered to one or more asynchronous subscribers.*

This rationale is based on embracing fine-grained transactions instead of transactions spanning many aggregates or entities because in the second case the database locks amount will be pretty bad in large scale applications with a high scalability needs. Embracing the fact that high-scalable applications must not have instant transactional consistency between multiple aggregates helps accepting the concept of eventual consistency. Atomic changes are in many cases not needed by the business, and it is in any case responsibility of the domain experts to say that something really needs atomic transactions or not. Then, if some operation always needs an atomic transaction between multiple aggregates, you should at least wonder if your aggregate should be larger and was not correctly designed.

However, other developers and architects, like *Jimmy Bogard,* are okay by spanning a single transaction across several aggregates but only when those additional aggregates are related to side effects for the same original command, for instance:

J.B. [A better domain events pattern](): … *Typically, I want the side effects of a domain event to occur within the same logical transaction, but not necessarily in the same scope of raising the domain event…* … *Just before we commit our transaction (DbContext SaveChanges()), we dispatch our events to their respective handlers*.

If you are dispatching the domain events right *before* committing the original transaction is because you want the side effects of those events to be included in the same transaction so, for instance, if the EF DbContext SaveChanges() fails the transaction will roll back all changes, including the rest of the

side effect operations implemented by the related domain event handlers because the DbContext life scope is by default defined as "scoped", so the DbContext object is shared across multiple Repositories objects being instantiated within the same scope or object graph which also coincides with the HttpRequest scope when developing Web API or MVC apps.

In reality, both approaches (single atomic transaction vs. eventual consistency) can be right, it really depends on your domain/business requirements and what the domain experts tell you. Also, depending on how scalable you need it to be (more granular transactions will provoke less impact in regards database locks) and how much investment you are willing to do in your code, since eventual consistency will require a more complex code in order to detect possible inconsistencies across aggregates and the need to implement compensatory actions. Take into account that it you commit changes on the original aggregate in the first place and afterwards, when the events are being dispatched there is any issue and the events handlers cannot commit their side effects, you will have inconsistencies between aggregates.

A way to allow compensatory actions would be to store the domain events into additional database tables so it can be part of the original transaction. Afterwards, you could have batch processing detecting inconsistencies and running compensatory actions in case of issues by comparing the list of events with the current state of the aggregates.

In any case, you can choose the approach you might need, but the initial "deferred approach" implementation for raising and dispatching domain events would be pretty similar.

That is neat, but, how do you actually dispatch those events to their respective event handlers? What is that _mediator object that you see in the previous code? Well, that has to do with the techniques and artifacts you can use to map between events and their event handlers.

The Domain Event Dispatcher: Mapping from events to event handlers

Once you are able to dispatch or publish the events you need any kind of artifact that will publish the event so every related handler would get it and will process side effects based on that event.

One way to do it would be with a real messaging system or even an Event Bus possibly based on a Service Bus. However, that might be too much for processing domain events since you just need to process those events within the same process (same domain and application layer).

One way to map from events to multiple event handlers is by using types registration in an IoC container so you can dynamically infer where to dispatch the events. In other words, you need to know what event handlers need to get any specifc event. You can see a simplified approach for that in the image X-XX.
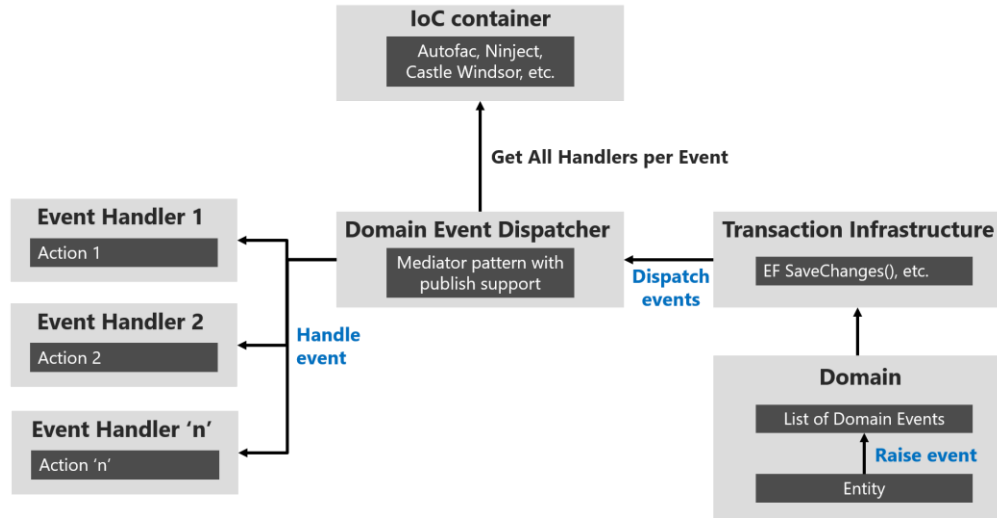
Figure X-XX. Domain Event Dispatcher

You can build all the "plumbing" and artifacts to implement that approach by yourself, however, you can also use already available libraries like [MediatR](#) which underneath uses your IoT container, so you can directly use the pre-defined interfaces and mediator's publish/dispatch methods.

In terms of code, you first need to register the event handler types in your IoC container.

```
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        // Other registrations

        // Register the DomainEventHandler classes (they implement IAsyncNotificationHandler<>) in assembly holding the Domain Events
        builder.RegisterAssemblyTypes(
                            typeof(ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler).
                                                                GetTypeInfo().Assembly)
            .Where(t => t.IsClosedTypeOf(typeof(IAsyncNotificationHandler<>)))
            .AsImplementedInterfaces();

        // Other registrations
    }
}
```

That code first identifies the assembly holding the domain event handlers based on the assembly that holds any of them. Then, since all the event handlers implement the interface *IAsyncNotificationHandler* it just searched for those types and registers all the event handlers.

How to subscribe to domain events

When using MediatR, each event handler is enforced to use an event type to be provided on the generic's parameter of the IAsyncNotificationHandler interface, as you can see in the following code.

```
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
                        : IAsyncNotificationHandler<OrderStartedDomainEvent>
```

Based on that relationship between event and event handler (that can be considered the subscription), the mediator artifact is able to discover all the event handlers per event and trigger each of those event handlers.

How to handle domain events

Finally, the event handler will usually implement application layer code which will be using infrastructure repositories to obtain the required additional aggregates and execute side effect domain logic.

```csharp
public class ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler
                    : IAsyncNotificationHandler<OrderStartedDomainEvent>
{
    private readonly ILoggerFactory _logger;
    private readonly IBuyerRepository<Buyer> _buyerRepository;
    private readonly IIdentityService _identityService;

    public ValidateOrAddBuyerAggregateWhenOrderStartedDomainEventHandler(ILoggerFactory logger,
IBuyerRepository<Buyer> buyerRepository, IIdentityService identityService)
    {
        //Parameter's validations
        //...
    }

    public async Task Handle(OrderStartedDomainEvent orderStartedEvent)
    {
        var cardTypeId = (orderStartedEvent.CardTypeId != 0) ? orderStartedEvent.CardTypeId : 1;

        var userGuid = _identityService.GetUserIdentity();

        var buyer = await _buyerRepository.FindAsync(userGuid);
        bool buyerOriginallyExisted = (buyer == null) ? false : true;

        if (!buyerOriginallyExisted)
        {
            buyer = new Buyer(userGuid);
        }

        buyer.VerifyOrAddPaymentMethod(cardTypeId,
                                      $"Payment Method on {DateTime.UtcNow}",
                                      orderStartedEvent.CardNumber,
                                      orderStartedEvent.CardSecurityNumber,
                                      orderStartedEvent.CardHolderName,
                                      orderStartedEvent.CardExpiration,
                                      orderStartedEvent.Order.Id);

        var buyerUpdated = buyerOriginallyExisted ? _buyerRepository.Update(buyer) :
                                                    _buyerRepository.Add(buyer);

        await _buyerRepository.UnitOfWork
                                .SaveEntitiesAsync();

        //Logging code using buyerUpdated info, etc.
    }
}
```

The former event handler's code is considered application layer as it is using infrastructure repositories explained in the next section focusing on the infrastructure-persistence layer. Event Handlers could also use other infrastructure components.

Domain events could generate Integration events to be published outside of the microservice boundaries

Finally, is important to mention that sometimes you might want to propagate events across multiple microservices. That is considered an integration event and it could be published through an Event Bus from any specific domain event handler.

Conclusions on domain events

As stated, use domain events to explicitly implement side effects of changes within your domain.

In other words, and using DDD lingo, use domain events to explicitly implement side effects across one or multiple *aggregates*. Additionally, and for better scalability and less impact in database locks, use eventual consistency between aggregates within the same domain.

For additional information on domain events, read the following references.

References – Implementing Domain Events
What is a Domain Event? [Greg Young]
http://codebetter.com/gregyoung/2010/04/11/what-is-a-domain-event/
Domain Events [Jan Stenberg]
https://www.infoq.com/news/2015/09/domain-events-consistency
A Better Domain Events Pattern [Jimmy Bogard]
https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/
Effective Aggregate Design Part II: Making Aggregates Work Together [Vaughn Vernon]
http://dddcommunity.org/wp-content/uploads/files/pdf_articles/Vernon_2011_2.pdf
Strengthening your domain: Domain Events [Jimmy Bogard]
https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/
Domain Events Pattern Example [Tony Truong]
http://www.tonytruong.net/domain-events-pattern-example/
Domain Events – Take 2 [Udi Dahan]
http://udidahan.com/2008/08/25/domain-events-take-2/
Domain Events – Salvation [Udi Dahan]
http://udidahan.com/2009/06/14/domain-events-salvation/
How to create fully encapsulated Domain Models [Udi Dahan]
http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/
Don't publish Domain Events, return them! [Jan Kronquist]
https://blog.jayway.com/2013/06/20/dont-publish-domain-events-return-them/
Domain Events vs. Integration Events in DDD and microservices architectures [Cesar de la Torre]
https://blogs.msdn.microsoft.com/cesardelatorre/2017/02/07/domain-events-vs-integration-events-in-domain-driven-design-and-microservices-architectures/

Designing the Infrastructure-Persistence Layer

The data persistence components provide access to the data hosted within the boundaries of your microservice (i.e. your microservice's database). They contain the actual implementation of components such as Repositories and Unit of Work patterns that provide functionality to access data hosted within the boundaries of your microservice.

The Repository pattern

Repositories are classes/components that encapsulate the logic required to access data sources. They centralize common data access functionality, providing better maintainability and decoupling the infrastructure or technology used to access databases from the Domain layer. If you use an ORM like Entity Framework, the code that must be implemented is highly simplified thanks to Linq and strong typing., This lets you focus on the data persistence logic rather than on data access plumbing.

The Repository pattern is one of the well documented ways of working with a data source. Martin Fowler in his [PoEAA book](#) describes a repository as follows:

"A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting in a similar way to a set of domain objects in memory. Client objects declaratively build queries and send them to the repositories for answers. Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way that is closer to the persistence layer. Repositories, also, support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping".

Define one Repository per Aggregate

For each aggregate (or Aggregate-Root) you should create one Repository class that allows you to populate data in-memory, coming from the database in the form of the Domain Entities. This also allows you to persist updated data in the entities of the aggregate back into the database.

If you are using the CQS/CQRS architectural pattern, then most of the public methods you will have in a Repository will create/update/delete in the database from your Domain Model. You won't need any methods for queries in such a Repository.

It is important to re-emphasize that only one Repository should be defined for each Aggregate-Root. Following the goals of the aggregate-root to maintain transactional consistency between all the objects within an aggregate, you should never create a Repository for each table in the database, just one for each aggregate-root.

In a microservice based on DDD, the only channel you should use to update the database should be through the Repositories. This is because they have a one-to-one relationship with the Aggregate-Root, which controls the aggregate's invariants and transactional consistency. It is okay to query the database through other channels (as you can do following a CQRS approach), because queries are idempotent and no matter how many queries you do, the database won't change. However, the transactional area, the updates, must always be controlled by the Repositories and the Aggregate-Roots.
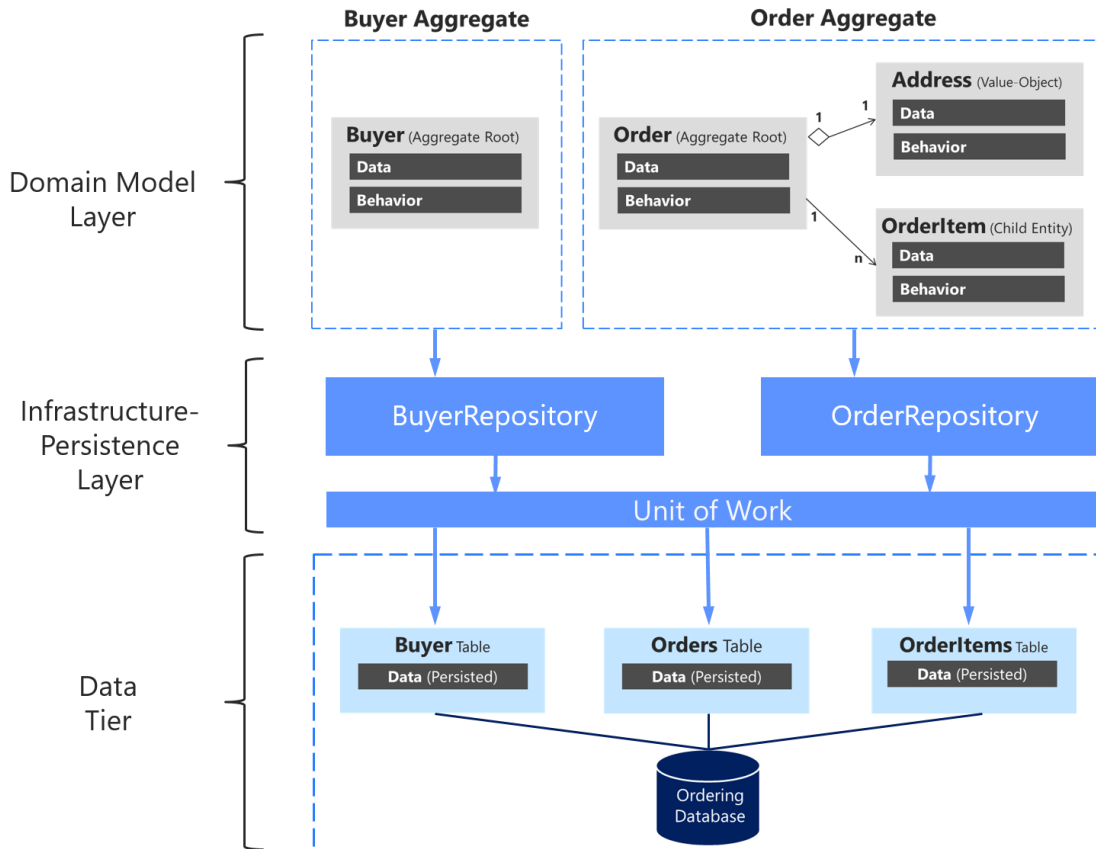
Figure X-XX. Relationship between Repositories, Aggregates and Database Tables

Enforcing one Aggregate Root per Repository

It can be valuable to implement your repository design in such a way that it enforces the rule that only aggregate roots should have repositories. You can create a generic or base repository type that constrains the type of entities it works with to ensure they have the IAggregateRoot market interface.

Thus, each repository class implemented at the infrastructure layer implements its own contract or interface, like in the following code.

```
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class OrderRepository : IOrderRepository
    {
```

Going further, each specific repository interface is implementing the generic IRepository.

```
    public interface IOrderRepository : IRepository<Order>
    {
        Order Add(Order order);
        //...
    }
```

However, one way to have the code better enforce the DDD convention that each Repository should be related to a single Aggregate would be to implement a generic repository type so it is explicit that

you are using a repository to target a specific aggregate. That can be easily done implementing that generic at the IRepository base interface, as in the following code.

```
public interface IRepository<T> where T : IAggregateRoot
```

The Repository pattern makes it easier to test your application logic

The Repository pattern allows you to easily test your application with unit tests. Remember that unit tests only test our code, not infrastructure, so the repository abstractions will meka it easier to achieve that goal.

As introduced in a previous section, it is recommended to define and place the Repository interfaces in the domain layer so the application layer (for instance, your Web API microservice) doesn't directly depend on the Infrastructure layer where you have implemented the actual Repository classes. By doing so and using Dependency Injection in the controllers of your Web API you could implement mock Repositories that would return fake hard-coded data instead of accessing the database. That decoupled approach allows you to create and run unit tests that can test just the logic of your application without requiring connectivity to the database.

Connections to databases can fail and more importantly, running hundreds of tests against a database is a bad thing for two reasons. First, it might take a lot of time because of the large number of tests, and second, the database's records might change and impact on the results of your tests, so they might not be consistent. Testing against the database is not a Unit 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 class and the legacy Data Access (DAL) class. A Data Access object directly performs data access and persistence operations against the storage. A repository marks the data with the operations you want to perform in the memory of a Unit of Work object (as in EF when using the DbContext), but these updates will not be performed immediately.

A Unit of Work is referred to as a single transaction that involves multiple insert/update/delete operations. In simple terms, it means that for a specific user action (for example, registration on a website), all the insert/update/delete transactions are handled in a single transaction. This is more efficient than handling multiple database transactions in a chattier way.

These multiple persistence operations will be performed later in a single action when your code from the Application layer commands it. The decision about applying the in-memory changes to the actual database storage is typically based on the Unit of Work pattern. In EF the Unit of Work is implemented as the DBContext.

In many cases, this pattern or way of applying operations against the storage can increase the application performance and reduce the possibility of inconsistencies. Also, it reduces transaction blocking in the database tables because all the intended operations will be committed as part of one transaction. This is more efficient in comparison to executing many isolated operations against the database. Therefore, the selected ORM will be able to optimize the execution against the database by grouping several update actions, as opposed to many small separate executions.

References – Infrastructure and Persistence patterns
The Repository pattern
http://martinfowler.com/eaaCatalog/repository.html
https://msdn.microsoft.com/en-us/library/ff649690.aspx
http://deviq.com/repository-pattern/
-- The Repository pattern. By Eric Evans in his DDD book --
The Unit of Work pattern
http://martinfowler.com/eaaCatalog/unitOfWork.html
Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application
https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

Implementing the Infrastructure-Persistence Layer with Entity Framework Core

When using relational databases such as SQL Server, Oracle, or PostgreSQL a recommended approach is to implement the persistence layer based on Entity Framework (EF). EF supports LINQ and provides strongly typed objects for your model, as well as simplified persistence into your database.

Entity Framework has a long history as part of the .NET Framework. When using .NET Core, you should also use Entity Framework Core, which runs on Windows or Linux in the same way as .NET Core. EF Core is a complete rewrite of Entity Framework, implemented with a much smaller footprint and important improvements in performance.

Entity Framework Core introduction

Entity Framework (EF) Core is a lightweight, extensible, and cross-platform version of the popular Entity Framework data access technology.

Since an introduction to EF Core is already available in Microsoft's documentation, this guidance is simply pointing to it with no further details:

References – Entity Framework Core
EF Core intro
https://docs.microsoft.com/en-us/ef/core/
Getting started with ASP.NET Core and Entity Framework Core
https://docs.microsoft.com/en-us/aspnet/core/data/ef-mvc/
DbContext
https://docs.microsoft.com/en-us/ef/core/api/microsoft.entityframeworkcore.dbcontext
Compare EF Core & EF6.x
https://docs.microsoft.com/en-us/ef/efcore-and-ef6/index

Infrastructure in Entity Framework Core from a DDD perspective

From a Domain-Driven Design point of view, an important of EF is the ability to use POCO Domain Entities, also known as POCO Code-First entities in EF jargon. By using POCO Domain Entities, your Domain Model classes are persistence ignorant, as the Persistence Ignorance and the Infrastructure Ignorance principles state.

In addition, in EF Core 1.1 you can have plain fields in your entities instead of properties with public/private setters. If you don't want an entity field to be accessible from the outside, you just create the attribute/field. There is no need to use private setters if you prefer this cleaner approach.

In a similar way, you can now have properly encapsulated collections (like a `List<>` or `HashSet<>`) in your entities that rely on EF for persistence. Previous versions of Entity Framework required collection properties to support `ICollection<T>`, which meant any developer using the parent entity class could add or remove items from its property collections. Per DDD patterns you should encapsulate domain behavior and rules within the entity class itself, so it can control invariants, validations and rules when accessing any collection. Therefore, it is not a good practice in DDD to allow public access to collections of child entities or value-objects. Instead, you want to expose methods that control how and when your fields and property collections can be updated, and what behavior and actions should occur when that happens.

You can use a private collection while exposing a read-only `IEnumerable,` as shown in the following code example.

```csharp
public class Order : Entity
{
    // Using private fields, allowed since EF Core 1.1
    private DateTime _orderDate;
    //… Other fields
    private readonly List<OrderItem> _orderItems;
    public IEnumerable<OrderItem> OrderItems => _orderItems.AsReadOnly();

    protected Order() { }
    public Order(int buyerId, int paymentMethodId, Address address)
    {
        //Initializations
    }

    public void AddOrderItem(int productId, string productName, decimal unitPrice,
decimal discount, string pictureUrl, int units = 1)
    {
        //Validation logic…

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

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

Note that the property `OrderItems` can now only be accessed as read-only with `List<>.AsReadOnly()`. This will create a read only wrapper around the private list so it's protected against external updates. It's much cheaper than using `.ToList()` because it won't have to copy all of the items in a new collection, just one heap alloc for the wrapper instance.

EF Core provides a way to map the domain model to the physical database without contaminating the domain model. It's pure .NET POCO code, because the mapping action is implemented in the persistence layer. In that mapping action, you need to configure the fields to database mapping. In the `OnModelCreating,`code shown below, the code in bold tells EF Core to access the `OrderItems` property through its field.

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
   //...
   modelBuilder.Entity<Order>(ConfigureOrder);
   //... Other entities
}

void ConfigureOrder(EntityTypeBuilder<Order> orderConfiguration)
{
    //.. Other configuration ..

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

    //.. Other configuration ..
 }
```

When using fields instead of properties, the OrderItem entity is persisted just as if it had a List<OrderItem> property, but now it exposes a single interface (the AddOrderItem() method) for adding new items to the order, so behavior and data are tied together and will be consistent throughout any application code that uses the Domain Model.

Implementing custom Repositories with Entity Framework Core

At the implementation level, a repository is simply a class with data persistence code coordinated by a Unit of Work (DBContext in EF Core) when performing updates, as shown in the following class:

```
//usings…
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository
    {
        private readonly OrderingContext _context;

        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }
}

        public BuyerRepository(OrderingContext context)
        {
            if (context == null)
            {
                throw new ArgumentNullException(
                            nameof(context));
            }

            _context = context;
        }

        public Buyer Add(Buyer buyer)
        {
            return _context.Buyers
                .Add(buyer)
```

Repository contract implemented in the Domain Layer
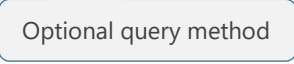
The EF DbContext comes in the constructor through Dependency Injection and is shared between multiple Repositories within the same HTTP request/scope thanks to its by default lifetime (ServiceLifetime.Scoped) that can also be explicitly set at services.AddDbContext<>

Adds a Buyer entity to the UnitOfWork (DbContext)

```
            .Entity;
    }

    public async Task<Buyer> FindAsync(string BuyerIdentityGuid)
    {
        var buyer = await _context.Buyers
            .Include(b => b.Payments)
            .Where(b => b.FullName == BuyerIdentityGuid)
            .SingleOrDefaultAsync();

        return buyer;
    }
}
```

> Optional query method

Methods to implement in a Repository (Updates/Transactions vs. Queries)

Within each Repository class, you should place the persistence methods that update the state of entities contained by its related Aggregate. Remember there is 1:1 relationship between an Aggregate and its related Repository. Take into account that an Aggregate-Root entity object might have embedded child entities within its EF graph, For example, a Buyer might have multiple PaymentMethods related as child entities.

Since the selected approach for the Ordering microservice in the eShopOnContainers sample application is also based on CQS/CQRS, most of the queries are not implemented in custom repositories. Developers have the freedom to create the queries and joins they need for the presentation layer without the restrictions imposed by Aggregates, custom Repositories per aggregate, and DDD in general. Most of the custom repositories suggested by this guidance might only have update/transactional methods but not query methods, unless you need a specific query for the transactional operations, For example, the `BuyerRepository` repository implements a `FindAsync()` method, because the application needs to know if a particular buyer exists before creating a new buyer related to the order. Therefore, having query methods in these repositories would be optional if using CQRS approaches and only used if needed by validations of data required for the transactions.

Custom repository vs. using EF DbContext directly

The Entity Framework `DbContext` class is based on the UnitOfWork and Repository pattern and can be used directly from your code, for example from an ASP.NET Core MVC controller. That is the way you can create the simplest code, as in the CRUD Catalog microservice in the eShopOnContainers sample sample. So, in cases where you just want to have the simplest code possible, you might want to directly use the DbContext class.

However, implementing custom Repositories provides several benefits when implementing more complex microservices or applications. The repository and unit of work patterns are intended to create an abstraction layer between the infrastructure persistence layer and the application and domain layers. Implementing these patterns can help insulate your application from changes in the data store and can facilitate automated unit testing.

Once you have implemented one repository class and repository interface per Aggregate-Root, when you get the injected instance (through DI) of the repository implementation in your controller, you are

using the interface so that the controller will accept a reference to any object that implements that repository interface. When the controller runs under a web server, it receives a repository that works with the Entity Framework. When the controller runs under a unit test class, it could receive a mock repository implementation that works with fake data, probably hard-coded so it is predictable and stored in a way that you can easily manipulate for testing, such as an in-memory collection.
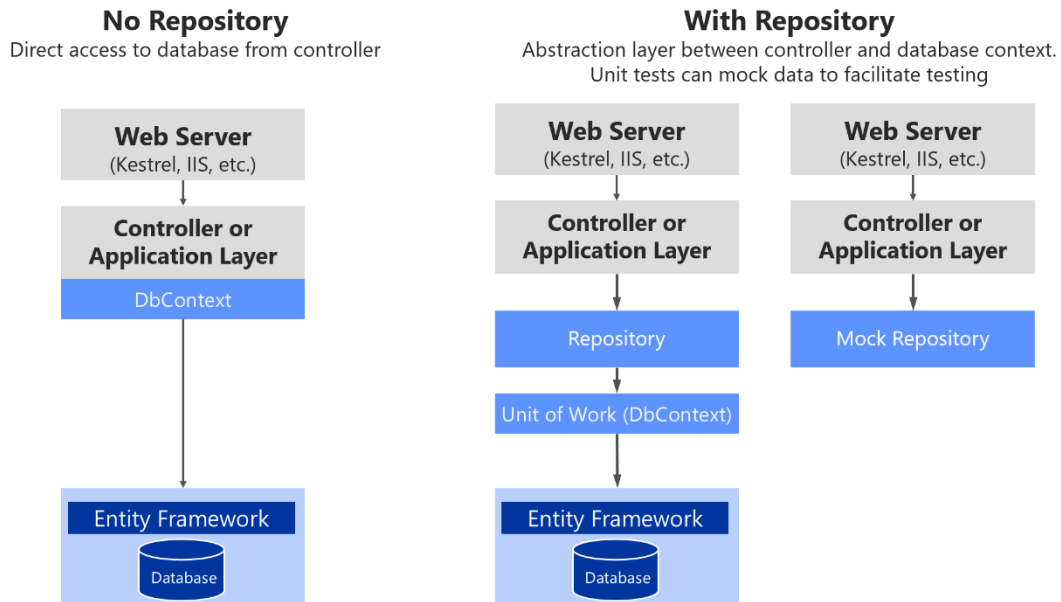
Figure X-XX. Using custom Repositories vs. plain DbContext

There are multiple alternatives when mocking. You could mock just repositories or you could also mock a whole unit of work.

Later, when focusing on the application layer, you'll see how dependency injection works in ASP.NET Core and how it is implemented when using Repositories.

In short, custom repositories allow you to test code easier with unit tests that aren't impacted by the data tier state. If you run tests that also access the actual database through the Entity Framework, they are not unit tests but integration tests, which are a lot slower and more brittle.

If you were using DbContext directly, the only choice you have to run unit tests would be by using an In-memory SQL Server with predictable data for unit tests. You wouldn't be able to control mock objects and fake data in the same way.

EF DbContext and IUnitOfWork instance lifetime in your IoC container

It's important to highlight that the DbContext object (exposed as an `IUnitOfWork`) might need to be shared among multiple repositories within the same HTTP request scope. For example, when the operation being executed has to deal with multiple aggregates, or simply because you are using multiple repository instances. It is also important to mention that the IUnitOfWork interface is part of the domain, not an EF type.

In order to do that, and as shown in the code below, the instance of the DbContext object has to be `ServiceLifetime.Scoped`, which is the default lifetime when registering your DbContext with `services.AddDbContext<>` in your IoC container, from the `ConfigureServices()` method of your Startup.cs file in your ASP.NET Core Web API project.

```csharp
public IServiceProvider ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc(options =>
    {
        options.Filters.Add(typeof(HttpGlobalExceptionFilter));
    }).AddControllersAsServices();

    services.AddEntityFrameworkSqlServer()
        .AddDbContext<OrderingContext>(options =>
        {
            options.UseSqlServer(Configuration["ConnectionString"],
                sqlop => sqlop.MigrationsAssembly(typeof(Startup).GetTypeInfo().
                                                    Assembly.GetName().Name));
        },
        ServiceLifetime.Scoped   // Note that Scope is the 'by default' choice
                                 // in AddDbContext<>. It's shown here only for
                                 // didactic purposes, but no need to be explicit
        );
}
```

The DbContext instantiation mode shouldn't be configured as `ServiceLifetime.Transient` or `ServiceLifetime.Singleton`.

Repository's instance lifetime in your IoC container

In a similar way, repository's lifetime should usually be set as *scoped* (*InstancePerLifetimeScope* in Autofac). It could also be Transient (InstancePerDependency in Autofac), but your service will be more efficient in regards memory when using the *scoped* lifetime.

```csharp
// Registering a Repository in Autofac IoC container
builder.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();
```

Important: Be aware that using the *singleton* lifetime could cause you serious problems when your DbContext is (by default) scoped.

References – Implementing Repositories with EF
Implementing Repositories with Entity Framework Core
https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application
https://www.infoq.com/articles/repository-implementation-strategies
Comparing ASP.NET Core IoC container service lifetimes with Autofac IoC container instance scopes
https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Table Mapping

Table mapping identifies the table data to be queried from and saved to in the database.

Previously you saw how your domain entities (i.e. Product or Order) can be used to generate a related database schema. In EF, most of it is based on the concept of *conventions*. Conventions are topics like "*What will be the name of a table?*" or "*What property is going to be the primary key?*", and they are typically based on conventionional names, for example "*a property ending with the suffix 'Id' will be the primary key*".

By convention, each entity will be set up to map to a table with the same name as the `DbSet<TEntity>` property that exposes the entity on the derived context. If no `DbSet<TEntity>` is provided for the given entity, the class name is used.

Data Annotations vs. Fluent API

There are many additional EF Core conventions and most of them can be changed by using either Data Annotations or Fluent API, implemented within the `OnModelCreating()` method.

Data Annotations must be used on the entity model classes themselves, which is a more intrusive way from a DDD point of view. This is because you are contaminating your model with data annotations related to the infrastructure database. On the other hand, Fluent API is a convenient way to change most conventions and mappings within your Data Persistence Infrastructure Layer, so the Entity Model will be clean and decoupled from the persistence infrastructure.

Fluent API and OnModelCreating()

As mentioned, in order to change conventions and mappings, you can use the method `OnModelCreating()` from the DbContext class, as shown in the code below from the Ordering microservice, part of the eShopOnContainers application.

```csharp
    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        //Other entities
        modelBuilder.Entity<OrderStatus>(ConfigureOrderStatus);
        //Other entities
    }
    void ConfigureOrder(EntityTypeBuilder<Order> orderConfiguration)
    {
        orderConfiguration.ToTable("orders", DEFAULT_SCHEMA);

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

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

        orderConfiguration.Property<DateTime>("OrderDate").IsRequired();
        orderConfiguration.Property<string>("Street").IsRequired();
        orderConfiguration.Property<string>("State").IsRequired();
        orderConfiguration.Property<string>("City").IsRequired();
        orderConfiguration.Property<string>("ZipCode").IsRequired();
        orderConfiguration.Property<string>("Country").IsRequired();
        orderConfiguration.Property<int>("BuyerId").IsRequired();
        orderConfiguration.Property<int>("OrderStatusId").IsRequired();
        orderConfiguration.Property<int>("PaymentMethodId").IsRequired();

        var navigation = orderConfiguration.Metadata.FindNavigation(nameof(Order.OrderItems));
        // DDD Patterns comment:
        //Set as Field (New since EF 1.1) to access the OrderItem collection property as a field
        navigation.SetPropertyAccessMode(PropertyAccessMode.Field);
```

```
        orderConfiguration.HasOne(o => o.PaymentMethod)
            .WithMany()
            .HasForeignKey("PaymentMethodId")
            .OnDelete(DeleteBehavior.Restrict);

        orderConfiguration.HasOne(o => o.Buyer)
            .WithMany()
            .HasForeignKey("BuyerId");

        orderConfiguration.HasOne(o => o.OrderStatus)
            .WithMany()
            .HasForeignKey("OrderStatusId");
    }
}
```

You could set all the Fluent API mappings within the same `OnModelCreating()` method, but it is advisable to partition that code and have multiple sub-methods, one per entity, as shown in the code above. Going further, for particularly large models, it can even be advisable to have separate source files/static classes for configuring different entity types.

The above code is very explicit, however, EF Core conventions do most of this automatically, so the actual code you would need to write to achieve the same thing would be much smaller.

The Hi/Lo pattern in EF Core

An interesting configuration in that code is that it is using HiLo as the key generation strategy, based on the *Hi/Lo pattern*. EF Core supports HiLo with the `ForSqlServerUseSequenceHiLo` method.

The Hi/Lo pattern describes a mechanism for generating safe-ids on the client side rather than in the database. Safe in this context means without collisions. This pattern is interesting for three reasons:

- It doesn't break the Unit of Work pattern
- It doesn't need many round-trips as the Sequence generators in other DBMS.
- It generates a human readable identifier, unlike GUID techniques.

Mapping Fields instead of Properties

With the new feature in EF Core 1.1 to map columns to fields, it is possible to not use any properties in the entity class, and just to map columns from a table to fields. A common use for that would be private fields for any internal state that needs not be accessed from outside the entity.

For example, the `_someOrderInternalState` field can't have a property for either setter or getter. That field could be calculated within the order's business logic and also used by the order's methods, so it shouldn't be a property. However, it needs to be persisted in the database. In EF 1.1 there's a way to map a field (without a related property) to a column in the database.

You can do this with single fields or also with collections, like a `List<>` field.

This point was mentioned when modeling the Domain Model classes, but here you can see how that mapping is performed with the `PropertyAccessMode.Field` configuration highlighted in the previous code.

Shadow Properties and Value-Objects

Shadow properties are properties that do not exist in your entity class. The value and state of these properties are maintained purely in the Change Tracker.

Shadow property values can be obtained and changed through the ChangeTracker API.

From a DDD point of view, shadow properties are a convenient way to implement *Value-Objects* by hiding the Id as a shadow property primary key. This is important since a Value-Object shouldn't have identity or at least it is not important, as mentioned in the Domain Model Layer when shaping Value-Objects. The point here is that at the time, EF Core doesn't have any way to implement Value-Objects as Complex Types, as it is possible in EF 6.x. That's why it currently must be implemented as an entity with a hidden Id (primary key) as a shadow property.

References – Table Mapping
Table Mapping
https://docs.microsoft.com/en-us/ef/core/modeling/relational/tables
Use HiLo to generate keys with Entity Framework Core
http://www.talkingdotnet.com/use-hilo-to-generate-keys-with-entity-framework-core/
Backing Fields
https://docs.microsoft.com/en-us/ef/core/modeling/backing-field
Encapsulated Collections in Entity Framework Core
http://ardalis.com/encapsulated-collections-in-entity-framework-core
Shadow Properties
https://docs.microsoft.com/en-us/ef/core/modeling/shadow-properties

NoSQL databases as your persistence infrastructure

When using NoSQL databases for your infrastructure data tier, you wouldn't typically be using an ORM like Entity Framework Core. Instead you would use the API provided by the chosen NoSQL engines such as Azure Document DB, MongoDB, Cassandra, RavenDB, CouchDB, or Azure Storage Tables.

However, when using a No- SQL database, especially a Document-oriented database like Azure Document DB, CouchDB, or RavenDb, the way you design your model with DDD Aggregates is similar in regards to the identification of AggregateRoots, child entity classes, and value-object classes.

Basically, when using a document-oriented database, you would implement an Aggregate (group of Domain entities and value-objects that must keep consistency) as a single document, serialized in JSON or any other format.

The difference would be the way you persist that model. Therefore, when implementing a Domain Model, you want to have a model based on POCO entity classes, agnostic to the infrastructure persistence, so you could potentially move to a different persistence infrastructure. It wouldn't be trivial, as transactions and persistence operations will be very different, but at least you could have a clean and protected Domain Model, following the Persistence Ignorant principle.

In any case, when using NoSQL databases the entities will be more denormalized, so it's not a simple table mapping. Your domain model might have a few impacts, after all.

However, if you were modelling your Domain Model based on Aggregates, moving to NoSQL and document oriented databases might be easier, because you already defined the aggregate's boundaries which are similar to serialized documents in document-oriented databases.

For instance, the following JSON code is a sample implementation of an Order Aggregate when using a Document oriented database, similar to the order aggregate we implemented in the eShopOnContainers sample but not using EF Core underneath.

```
JSON example of the Order Aggregate when using a Document oriented DB
{
    "id": "2017001",
    "orderDate": "2/25/2017",
    "buyerId": "1234567",
    "address": [
        {
            "street": "100 One Microsoft Way",
            "city": "Redmond",
            "state": "WA",
            "zip": "98052",
            "country": "U.S."
        }
    ],
    "orderItems": [
        {"id": 20170011, "productId": "123456", "productName": ".NET T-Shirt",
"unitPrice": 25, "units": 2, "discount": 0},
        {"id": 20170012, "productId": "123457", "productName": ".NET Mug", "unitPrice":
15, "units": 1, "discount": 0}
    ]
}
```

When using a C# model to implement that aggregate to be used by, for instance, the Azure Document DB SDK, it would be similar to the C# POCO classes used with EF Core. The difference will be the way to use them from the application and infrastructure layers, as in the following code.

```csharp
//C# example of an Order Aggregate being persisted with DocumentDB API

// *** Domain Model Code ***
// Aggregate: Create an Order object with its child entities and/or value-objects.
// Then, use AggregateRoot's methods to add the nested objects so invariants and
// logic is consistent across the nested properties (Value-Objects and entities).
// This can be saved as JSON as is without converting into rows/columns.

Order orderAggregate = new Order
{
    Id = "2017001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "PO18009186470"
}


Address address = new Address
{
    Street = "100 One Microsoft Way",
    City = "Redmond",
    State = "WA",
    Zip = "98052",
    Country = "U.S."
}
```

```
orderAggregate.UpdateAddress(address);

OrderItem orderItem1 = new OrderItem
{
    Id = 20170011,
    ProductId = "123456",
    ProductName = ".NET T-Shirt",
    UnitPrice = 25,
    Units = 2,
    Discount = 0;
};

OrderItem orderItem2 = new OrderItem
{
    Id = 20170012,
    ProductId = "123457",
    ProductName = ".NET Mug",
    UnitPrice = 15,
    Units = 1,
    Discount = 0;
};
//Using methods with domain logic within the entity. No anemic-domain model
orderAggregate.AddOrderItem(orderItem1);
orderAggregate.AddOrderItem(orderItem2);
// *** End of Domain Model Code ***
//...

// *** Infrastructure Code using Document DB Client API ***
Uri collectionUri = UriFactory.CreateDocumentCollectionUri(databaseName,
                                                    collectionName);

await client.CreateDocumentAsync(collectionUri, order);

// As your app evolves, let's say your object has a new schema. You can insert OrderV2
objects without any changes to the database tier.
Order2 newOrder = GetOrderV2Sample("IdForSalesOrder2");
await client.CreateDocumentAsync(collectionUri, newOrder);
```

You can see that the way you work with your Domain Model can be similar to the way you are using it in your Domain Model Layer when the infrastructure was EF underneath. You still use the same AggregateRoot's methods to ensure consistency, invariants and validations within the aggregate.

However, when persisting your model into the NoSQL db, implemented in the infrastructure and persistence layer, this is where the code and API will dramatically change internally.

References – NoSQL Databases
Azure Document DB
https://docs.microsoft.com/en-us/azure/documentdb/documentdb-modeling-data
DDD Aggregate storage
https://vaughnvernon.co/?p=942
Event storage
https://github.com/NEventStore/NEventStore

Developing Microservices with .NET Core

Designing the microservice's Application Layer and Web API

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

The S.O.L.I.D. principles and Dependency Injection (DI) are critical techniques to be used in any modern and mission-critical application, such as developing a microservice with DDD patterns. However, you should also use DI and apply the S.O.L.I.D. principles even when you aren't using DDD approaches or patterns.

S.O.L.I.D. is an acronym that groups five fundamental principles:
- Single Responsibility Principle
- Open/close principle
- Liskov substitution principle
- Inversion Segregation principle
- Dependency Inversion principle

S.O.L.I.D. and DI tackle more about how you design your application/microservice internal layers and decoupled dependencies between them, so this is not related to the Domain but related to the application's technical design. But, DI allows you to decouple the infrastructure layer from the rest of the layers allowing a better decoupled implementation of the DDD layers.

Dependency injection (DI) is a technique for achieving loose coupling between objects and their dependencies. Rather than directly instantiating collaborators, or using static references, the objects a class needs to perform its actions are provided to or injected into the class. Most often, classes will declare their dependencies via their constructor, allowing them to follow the Explicit Dependencies Principle. DI is usually based on specific Inversion of Control (IoC) containers. ASP.NET Core provides a simple built-in IoC container, but you can also use your favorite IoC container, like Autofac or Ninject.

By following the S.O.L.I.D. Principles, your classes will naturally tend to be small, well-factored, and easily tested. What if you find that your classes tend to have way too many dependencies being injected? Using DI through the constructor it will be easy to detect by just taking a look at the number of parameters of your constructor. If there are too many dependencies, this is generally a sign (code smell) that your class is trying to do too much, and is probably violating SRP - the Single Responsibility Principle.

There is much to be said about S.O.L.I.D. and DI. It would really take another guide/book to cover it in detail, so this guide requires the reader to have a minimum knowledge or skills with these topics.

References – S.O.L.I.D. principles and Dependency Injection
S.O.L.I.D. principles http://deviq.com/solid/ **Dependency Injection** https://martinfowler.com/articles/injection.html **New is Glue** htthttp://ardalis.com/new-is-glue

Implementing the microservice's Application Layer and Web API

Using Dependency Injection to inject infrastructure objects into your application layer

The application layer, as mentioned previously, is whatever artifact you are building. In the case of a microservice built with ASP.NET Core, the application layer will usually be your Web API library. If you'd like to separate what is coming from ASP.NET Core (its infrastructure plus your controllers) from your custom application layer code, that could also be placed in a separate library.

ASP.NET Core includes a simple built-in IoC container (represented by the `IServiceProvider` interface) that supports constructor injection by default, and ASP.NET makes certain services available through DI. ASP.NET's container refers to the types it manages as services. You configure the built-in container's services in the `ConfigureServices` method in your application's Startup class.

Typically, you'd want to inject dependencies that implement infrastructure objects. The most typical dependencies to inject are Repositories, or for simpler implementations you could directly inject your Unit of Work pattern object (the EF DbContext object), as they are the implementation of your infrastructure persistence objects.

In the following example, you can see how .NET Core is injecting the needed Repository objects.

```csharp
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IOrderRepository _orderRepository;

    public CreateOrderCommandHandler(IOrderRepository orderRepository)
    {
        if (orderRepository == null)
        {
            throw new ArgumentNullException(nameof(orderRepository));
        }

        _orderRepository = orderRepository;
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        //
        // ... Additional code
        //

        // Create the Order AggregateRoot
        // Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate

        var address = new Address(message.Street, message.City, message.State,
                                  message.Country, message.ZipCode);
        var order = new Order(address, message.CardTypeId, message.CardNumber,
                              message.CardSecurityNumber, message.CardHolderName,
                              message.CardExpiration);

        foreach (var item in message.OrderItems)
        {
```

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

        //Persist the Order through the Repository
        _orderRepository.Add(order);

        var result = await _orderRepository.UnitOfWork
                                          .SaveEntitiesAsync();

        return result > 0;
    }
}
```

Finally, it is using the injected repositories to execute the transaction and persist the state changes.

Registering the Dependency implementation types and interfaces/abstractions

You also need to know where to register the Interfaces and classes that will be injected to your objects through DI based on the constructors.

Using the built-in IoC container provided by ASP.NET Core

When using the built-in IoC container provided by ASP.NET Core (as in the simple Catalog microservice in the eShopOncontainers sample), you register the types in the `ConfigureServices()` method in the MVC Startup.cs file.

```
// Registration of types into ASP.NET Core built-in container
public void ConfigureServices(IServiceCollection services)
{
    // Register out-of-the-box framework services.
    services.AddDbContext<CatalogContext>(c =>
    {
        c.UseSqlServer(Configuration["ConnectionString"]);
    },
    ServiceLifetime.Scoped
    );

    services.AddMvc();

    // Register custom application dependencies.

    services.AddScoped<IMyCustomRepository, MyCustomSQLRepository>();

}
```

In this example, the last line of code states that when any of your constructors have a dependency on `IMyCustomRepository` (interface or abstraction), the IoC container will inject an instance of the `MyCustomSQLServerRepository` implementation class.

Using Autofac as IoC container

You can also use additional IoC containers and plug them to the ASP.NET Core pipeline, as in the Ordering microservice in the eShopOncontainers sample which uses *Autofac*. When using Autofac you

typically register the types via modules, which allow you to split the registration types between multiple files depending on where your types are, just as you could have the application types distributed across multiple class libraries.

For example, the following is the application module for one class library with the implemented custom types.

```
public class ApplicationModule
    :Autofac.Module
{
    public string QueriesConnectionString { get; }

    public ApplicationModule(string qconstr)
    {
        QueriesConnectionString = qconstr;
    }
    protected override void Load(ContainerBuilder builder)
    {

        builder.Register(c => new OrderQueries(QueriesConnectionString))
            .As<IOrderQueries>()
            .InstancePerLifetimeScope();

        builder.RegisterType<BuyerRepository>()
            .As<IBuyerRepository>()
            .InstancePerLifetimeScope();

        builder.RegisterType<OrderRepository>()
            .As<IOrderRepository>()
            .InstancePerLifetimeScope();
    }
}
```

In the code above, the abstraction `IOrderRepository` is registered along with the implementation class `OrderRepository`, which means that whenever a constructor is declaring a dependency through the abstraction or interface `IOrderRepository,` the IoC container will inject an instance of the `OrderRepository` class.

The instance scope type determines how an instance is shared between requests for the same service or dependency. When a request is made for a dependency, the IoC container can return a single instance per LifetimeScope (referred to in ASP.NET Core as "scoped"), a new instance per dependency (referred to in ASP.NET Core as "transient"), or a single instance shared across all objects using the IoC container (referred to in ASP.NET Core as "singleton").

For additional information about DI, lifetime scopes and usage in ASP.NET Core, read the following references.

References – ASP.NET Core DI and Autofac
Using Dependency Injection in ASP.NET Core and .NET Core
https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection
Autofac
http://docs.autofac.org/en/latest/getting-started/index.html
http://docs.autofac.org/en/latest/lifetime/instance-scope.html
Comparing lifetime scopes between ASP.NET Core built-in container and Autofac

Implementing the Command and Command-Handlers patterns

In the DI through constructor example shown in the previous section, the IoC container was injecting Repositories through a constructor, but exactly where were they injected? In a very simple Web API (for example, the Catalog microservice in the eShopOnContainers sample), you would injecting them at the MVC Controllers level, in a Controller constructor. However, in the previous example it is done at a CommandHandler level, so let's explain what a ComamndHandler is and why you would want to use it.

The Command pattern is intrinsically related to the CQRS pattern that was previously introduced in this guide. CQRS has two sides. The queries (previously explained using in this approach for simplified queries with Dapper Micro ORM) and the Commands as the starting point for the transactions/writes.

Remember, CQRS is not an architecture, it's a pattern which you can use in some microservices of your application architecture, or in all of them. You decide if you implement CQRS per Bounded Context or microservice. not as the top-level architecture for your whole application.

As shown in the high-level diagram below, the pattern is based on accepting commands from the clisnt side and process those commands based on the Domain Model rules and finally persisting the states with transactions.

High level "Writes-side" in CQRS
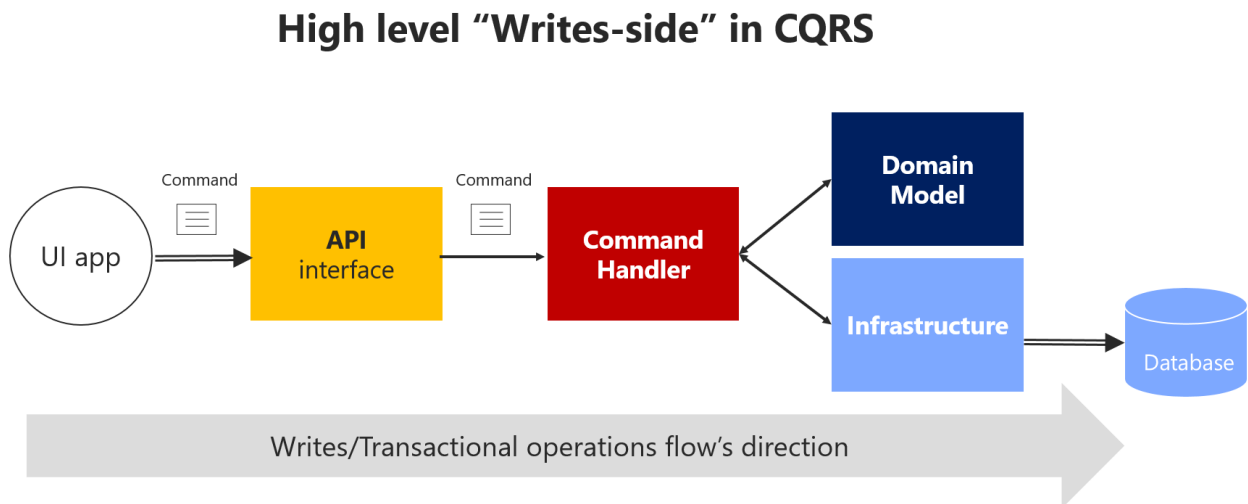
Figure X-XX. High level "Writes side" in a CQRS pattern

The Command

What is a command? – A command is a request for the system to perform an action that changes the state of the system. Since Commands are imperatives, they are typically named with a verb in the imperative tense and may include the aggregate type, for example *CreateOrderCommand*. Unlike an event, a command is not a fact from the past; it's only a request, and thus may be refused.

Developing Microservices with .NET Core

Commands can originat from either the user interface (UI) as a result of a user initiating a request, or from a process manager when the process manager is directing an aggregate to perform an action.

Another very important characteristic of a command is that *a command must be processed just once* by a single recipient. This is because commands might not be idempotent, therefore it's important that they be processed only once. For example, the same Order creation request shouldn't be processed more than once. This is a very important difference when comparing commands versus events. Usually you will want to process an event (something that happened in the past) multiple times, as many systems might be interested in that event.

Idempotency. Idempotency is a characteristic of an operation that means the operation can be applied multiple times without changing the result. For example, the operation "set the value x to ten" is idempotent, while the operation "add one to the value of x" is not. A Command is idempotent if it can be executed multiple times without changing the result, either because of the nature of the Command itself, or because of the way the system handles the Command.

Therefore, it is a good practice to make your commands and updates idempotent, when it makes sense. If for any reason (retry logic, hacking, etc.) the same CreateOrder command reaches your system multiple times, you should be able to identify it, insuring that you don't create multiple orders based on the same original CreateOrder command. To do so, you need to attach some kind of identity in the operations and identify whether that same command or update was already processed.

You send a command, you don't publish a command. Publishing is reserved for events which state a fact – that something has happened, and that the publisher has no concern about what receivers of that event do with it. But events are a different story related to Domain events and Integration events.

How then do you implement a Command? It's quite simple; a Command is implemented with a class that contains data fields or collections with all the information you need to execute that command. So, yes, a command is like a special kind of DTO (Data Transfer Object) used to request changes or transactions. The command itself is based on exactly what information is needed to process the command, and nothing more.

The following is an example of the simplified `CreateOrderCommand,` which is an immutable command, used in the Ordering microservice in the eShopOnContainers sample.

```
// DDD and CQRS patterns comment: Note that it is recommended to implement immutable Commands
// In this case, its immutability is achieved by having all the setters as private
// plus only being able to update the data just once, when creating the object through its
constructor.
// References on Immutable Commands:
// http://cqrs.nu/Faq
// https://docs.spine3.org/motivation/immutability.html
// http://blog.gauffin.org/2012/06/griffin-container-introducing-command-support/
// https://msdn.microsoft.com/en-us/library/bb383979.aspx

[DataContract]
public class CreateOrderCommand
    :IAsyncRequest<bool>
{
    [DataMember]
    private readonly List<OrderItemDTO> _orderItems;
    [DataMember]
    public string City { get; private set; }
    [DataMember]
```

```csharp
    public string Street { get; private set; }
    [DataMember]
    public string State { get; private set; }
    [DataMember]
    public string Country { get; private set; }
    [DataMember]
    public string ZipCode { get; private set; }
    [DataMember]
    public string CardNumber { get; private set; }
    [DataMember]
    public string CardHolderName { get; private set; }
    [DataMember]
    public DateTime CardExpiration { get; private set; }
    [DataMember]
    public string CardSecurityNumber { get; private set; }
    [DataMember]
    public int CardTypeId { get; private set; }
    [DataMember]
    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;

    public void AddOrderItem(OrderItemDTO item)
    {
        _orderItems.Add(item);
    }
    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }

    public CreateOrderCommand(string city, string street, string state, string country, string
zipcode,
        string cardNumber, string cardHolderName, DateTime cardExpiration,
        string cardSecurityNumber, int cardTypeId) : this()
    {
        City = city;
        Street = street;
        State = state;
        Country = country;
        ZipCode = zipcode;
        CardNumber = cardNumber;
        CardHolderName = cardHolderName;
        CardSecurityNumber = cardSecurityNumber;
        CardTypeId = cardTypeId;
        CardExpiration = cardExpiration;
    }
    public class OrderItemDTO
    {
        public int ProductId { get; set; }
        public string ProductName { get; set; }
        public decimal UnitPrice { get; set; }
        public decimal Discount { get; set; }
        public int Units { get; set; }
        public string PictureUrl { get; set; }
    }
}
```

Basically, the Command class contains all the data you will need to perform a business transaction by using the Domain Model objects. Thus, *Commands are simply data structures that contain read-only data, and no behavior*. The Command's name indicates it's purpose. In many languages like C#, Commands are represented as classes, but they are not true classes in the real OO sense.

As an additional characteristic, *commands are immutable* because their expected usage is to be processed directly by the domain model. Usually, they do not need to change during their

projected lifetime. The same happens with Events, but that is a different story.

In a C# class, immutability can be achieved by not having any setters, or other methods which change internal state. This immutability and lack of setters is an improvement in the Command's design, but it is not critical.

An example is a "Create an order" command. In this case, the Command class might be similar in terms of data to the Order you want to create, but you probably don't need the same attributes. For instance, the `CreateOrderCommand` doesn't have an Order Id because it hasn't been created yet.

Many other Command classes can be very simple, requiring only a few fields about some state that needs to be changed. For example, that would be the case if you are just changing the status of an Order from "InProcess" to "Paid" or "Shipped" status by using a command similar to the following:

```csharp
[DataContract]
public class UpdateOrderStatusCommand
    :IAsyncRequest<bool>
{
    [DataMember]
    public string Status { get; private set; }
    [DataMember]
    public string OrderId { get; private set; }
    [DataMember]
    public string BuyerIdentityGuid { get; private set; }
}
```

The Command-Handler class

The Command class example is pretty obvious. But where do you actually use that command object and provide the needed data to the Domain objects? In a Web API controller? In an Application Layer Service?

It turns out that it is pretty convenient to have a specific Command Handler class per Command. That is how the pattern works and it is precisely where you will use the Command object, the Domain objects and the infrastructure repository objects. The Command-Handler is in fact the heart of the Application Layer in terms of DDD.

A command handler receives a command and brokers a result from the appropriate aggregate. A result is either a successful application of the command, or an exception.

The command handler usually performs the following tasks:

- It receives the Command instance (from the mediator or any other infrastructure).
- It validates that the Command is a valid Command (if not validated by the mediator).
- It locates the aggregate instance that is the target of the Command.
- It invokes the appropriate method on the aggregate instance passing in any parameter from the command.
- It persists the new state of the aggregate to storage, which is the actual transaction.

The important point here is that all the domain logic in processing the command should be inside the domain model (the aggregates), fully encapsulated and unit-testable. The command-handler just acts

as a way to get the domain model out of the persistent store and tell the infrastructure layer (Repositories) to persist the changes when the model is ready. The advantage of this approach is that you can now refactor the domain logic in a fully encapsulated, behavioral domain model without changing anything else in the application plumbing level (Web API, etc.).

When command handlers get complex with too much logic, review them and just push the behavior down to the domain objects (aggregate-root's and child entity's) methods as needed by refactoring them.

As an example of a Command-Handler class, the following code shows the same `CreateOrderCommandHandler` class that uou saw earlier. In this case you can see highlighted the actual `Handle()` method and the operations with the Domain model objects/aggregates.

```csharp
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IBuyerRepository _buyerRepository;
    private readonly IOrderRepository _orderRepository;

    public CreateOrderCommandHandler(IBuyerRepository buyerRepository,
                                     IOrderRepository orderRepository)
    {
        if (buyerRepository == null)
        {
            throw new ArgumentNullException(nameof(buyerRepository));
        }

        if (orderRepository == null)
        {
            throw new ArgumentNullException(nameof(orderRepository));
        }

        _buyerRepository = buyerRepository;
        _orderRepository = orderRepository;
    }
    public async Task<bool> Handle(CreateOrderCommand message)
    {
        //
        // ... Additional code
        //

        // Create the Order AggregateRoot
        // Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate

        var order = new Order(buyer.Id, payment.Id,
                          new Address(message.Street,
                                      message.City, message.State,
                                      message.Country, message.ZipCode));

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                               item.Discount, item.PictureUrl, item.Units);
        }
```

Developing Microservices with .NET Core

```
        //Persist the Order through the Aggregate's Repository
        _orderRepository.Add(order);

        var result = await _orderRepository.UnitOfWork.SaveChangesAsync();
        return result > 0;
    }
}
```

This is the common sequence of steps a command handler might follow:

- Validate the command's incoming data.
- Use the command's data to operate with the aggregate root's methods and behavior.
- Internally within the Domain objects, Domain events could be raised while the transaction is executed, but that is transparent from a Command Handler point of view.
- If the aggregate's operation result is successful, integration events can be raised either from the infrastructure classes like Repositories or from the Command-Handler itself, after the transaction is finished.

References – Command and Command-Handler
At the Boundaries, Applications are Not Object-Oriented (by Mark Seemann)
http://blog.ploeh.dk/2011/05/31/AttheBoundaries,ApplicationsareNotObject-Oriented/
The Command pattern
http://cqrs.nu/Faq/commands-and-events
The Command-Handler pattern
http://cqrs.nu/Faq/command-handlers

The Command's process pipeline – How to trigger a Command Handler

The next question is, where do I call a Command-Handler? – You could manually call it from each related ASP.NET Core controller, however, that approach would be too coupled and not ideal.

The other two main options, which are the recommended options, are:

- Through an in-memory Mediator pattern artifact.
- With an asynchronous queue, in between controllers and handlers.

Using the mediator pattern (in-memory) in the Command's pipeline

As shown in figure X-XX, in a CQRS approach you use an intelligent mediator, similar to an in-memory bus, which is smart enough to redirect to the right Command-Handler based on the type of the Command/DTO being received. The small single black arrows between components mean the dependencies between objects (in many cases, injected through DI) with their related interactions.
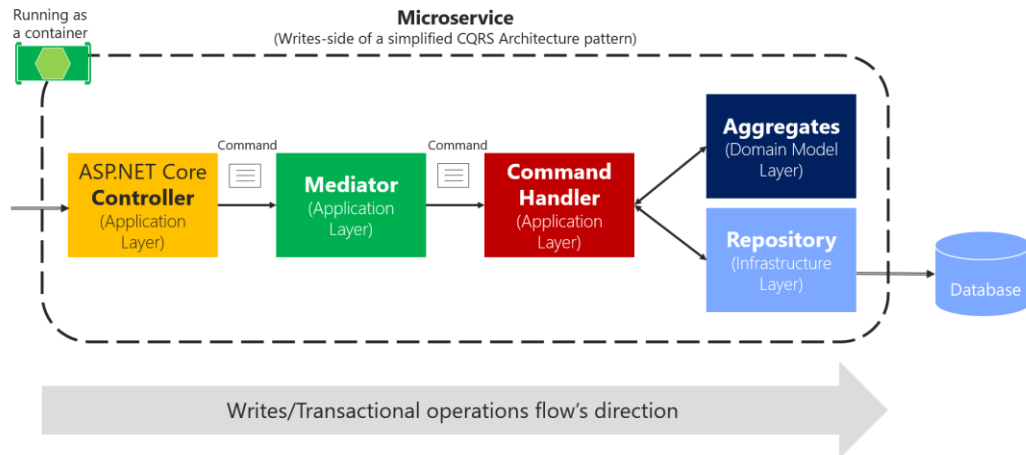
Figure X-XX. Using the Mediator pattern in CQRS microservice

The reason that using a mediator pattern makes sense is because in enterprise applications the processing requests can get increasingly complicated. You will want to be able to add an open number of cross-cutting concerns like logging, validations, transactions, audit, and security. In these cases, you can rely on a mediator pipeline (see mediator pattern) to provide a means for these extra behaviors or cross-cutting concerns.

A mediator is an object that encapsulates the "how" and coordinates execution based on state, the way it's invoked, or the payload you provide to it.

Basically, with a Mediator component you can apply those mentioned cross-cutting concerns in a centralized and transparent way by just applying *decorators*. See the decorator pattern.

Decorators are similar to Aspect Oriented Programming – AOP , only applied to a specific process-pipeline managed by the mediator component. Aspects in AOP implementing cross-cutting concerns are magically applied based on *aspect weavers* injected in compilation time or based on object call interception. Both typical AOP approaches are like magic and when dealing with serious issues or bugs can be difficult to debug. On the other hand, these decorators are explicit and applied only in the context of the mediator, so debugging is much more predictable and easy to do for any developer.

Using message queues (out-of-proc) in the Command's pipeline

Another choice is to use message queues, as shown in the image X-XX. That option could also be combined with the mediator component right before the command-handlers.

Writes-side of a CQRS Architecture pattern using messaging
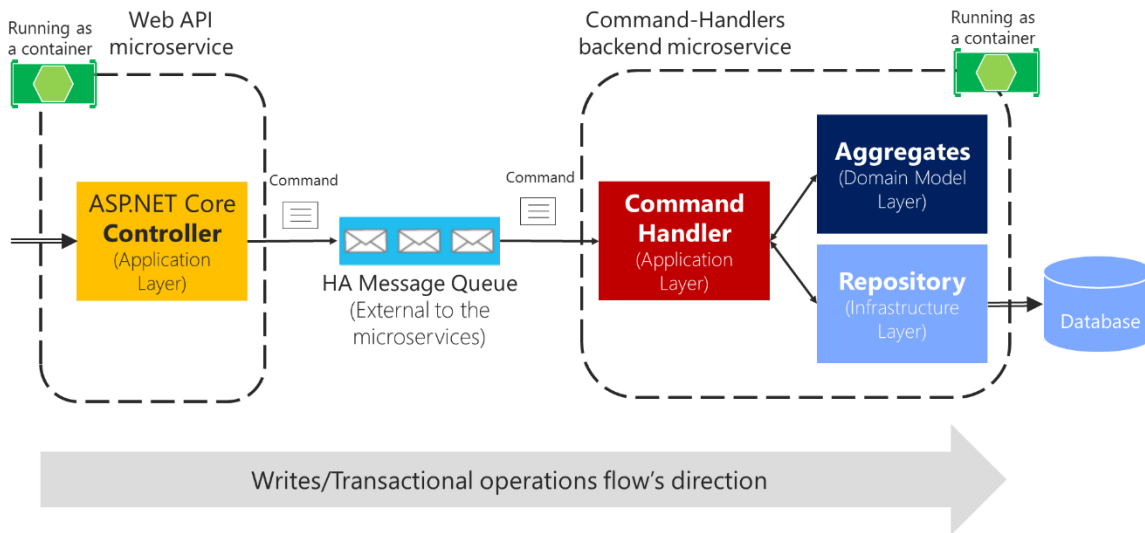
Figure X-XX. Using Message Queues with CQRS Commands

Using message queues to accept the commands can further complicate your command's pipeline, as you will probably need to split the pipeline in two processes connected through the external message queue. Still it should be used if you need to have better resiliency when submitting the command messages, plus providing better scalability and better performance because you can implement asynchronous messaging. Consider that in this case the controller just posts the command message into the queue and returns. Then, the command-handlers will be processing the messages at their own pace. That is a great benefit typical of queues, as the message queue can act as a buffer in cases when hyper scalability is needed for ingress data, for example for stocks or any other scenario with a high volume of ingress data.

However, because of the asynchronous nature of message queues, you will need to figure out how to communicate with the client application about the success or failure of the command's process. As a rule, you should never use "fire and forget" commands. Every business application needs to know if a command was processed successfully, or at least validated and accepted.

Thus, being able to respond to the client after validating a command message that was submitted to an asynchronous queue adds complexity to your system as compared to an in-process command process that returns the operation's result after running the transaction. Using queues, you might need to return the result of the command process through other operation result messages, which will require additional components and custom communication in your system.

Additionally, async commands are one way commands, which in many cases might not be needed as explained by Greg Young in the following extracts:

I find lots of code where people use "async command handling" or "one way command" messaging without any reason to do so (they are not doing some long operation, they are not executing external async code, they do not even cross application boundary to be using message bus). Why do they

introduce this unnecessary complexity? And actually, I haven't seen a CQRS code example with blocking command handlers so far, though it will work just fine in most cases.

An asynchronous command doesn't exist; it's actually another event. If I must accept what you send me and raise an event if I disagree, it's no longer you telling me to do something, it's you telling me something has been done. This seems like a slight difference at first, but it has many implications.

- Greg Young -

In the eShopOnContainers implementation it was chosen to use synchronous command processing driven by the Mediator pattern, as that easily allows you to return the success or failure of the process.

In any case, this should be a decision based on your application's or microservice's business requirements. Sometimes a command might not need any confirmation and then it would be a lot simpler to implement it as a asynchronous command.

Implementing the Command's process pipeline with a mediator pattern (MediatR)

As a sample implementation, this guidance is proposing the in-process pipeline based on the mediator pattern driving the commands ingestion and routing them, in memory, to the right command-handlers, plus applying decorators to separate cross-cutting concerns.

For implementation in .NET Core, there are multiple open source libraries available implementing the mediator pattern The chosen library used in this guidance is the *MediatR* open source library (created by Jimmy Bogard), but you could use any other approach. MediatR is a small, simple in-process messaging library that allows you to process messages like a Command, while applying decorators.

MediatR is also capable of using synchronous or asynchronous execution which is important depending on your desired application behavior.

Basically, using the mediator pattern helps you to reduce coupling and isolate the concerns of the requested work to be done while automatically connecting to the handler that performs that work (the Command-Handler, in this case).

First, let's take a look to the controller's code where you actually would use the mediator object.

The constructor of your controller can be a lot simpler with just a few dependencies instead of many dependencies that you would have if you had one per cross-cutting operation.

For instance, instead of a messy constructor with many cross-cutting dependencies, you can have a clean constructor like this:

```
public class OrdersController : Controller
{
    public OrdersController(IMediator mediator,
                            IOrderQueries orderQueries)
```

You can see that it provides a very clean and lean Web API controller. Within the controller's methods, the code is also pretty simple, basically just one line sending a Command to the mediator object:

```
[Route("new")]
[HttpPost]
```

Developing Microservices with .NET Core

```csharp
public async Task<IActionResult> CreateOrder([FromBody]CreateOrderCommand
                                                    createOrderCommand)
{
    var result = await _mediator.SendAsync(createOrderCommand);
    if (result)
    {
        return Ok();
    }
    return BadRequest();
}
```

In order for Mediator to be aware of your command-handler classes, you need first to wire it up by registering the mediator classes and the command-handler classes in your IoC container.

By default, Mediator uses Autofac as the IoC container, but you can also use the built-in ASP.NET Core IoC container or any other container supported by MediatR.

The following code shows how to register those types, Mediator's types and Commands when using Autofac modules.

```csharp
public class MediatorModule : Autofac.Module
{
    protected override void Load(ContainerBuilder builder)
    {
        builder.RegisterAssemblyTypes(typeof(IMediator).GetTypeInfo().Assembly)
            .AsImplementedInterfaces();

        builder.RegisterAssemblyTypes(typeof(CreateOrderCommand).GetTypeInfo().Assembly)
            .As(o => o.GetInterfaces()
                .Where(i => i.IsClosedTypeOf(typeof(IAsyncRequestHandler<,>)))
                .Select(i => new KeyedService("IAsyncRequestHandler", i)));

         builder.RegisterGenericDecorator(typeof(LogDecorator<,>),
                                          typeof(IAsyncRequestHandler<,>),
                                          "IAsyncRequestHandler");
        //Other types registration
    }

}
```

Because each Command Handler is implementing the interface with generics `IAsyncRequestHandler<T>`, then by inspecting the `RegisteredAssemblyTypes` it is able to relate each Command with its Command-Handler, because that relationship is stated in the CommandHandler class, as in the following example:

```csharp
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
```

This is the code that closes the loop and correlates Commands with CommandHandlers. The handler is just a simple class, but it inherits from `RequestHandler<T>` and MediatR makes sure it gets invoked with the correct payload.

Developing Microservices with .NET Core

Applying cross-cutting concerns when processing commands with the Mediator and Decorator patterns

There's one more thing:, the capability of being able to apply cross-cutting concerns to the mediator pipeline. In the Autofac registration module code you can also see at the end of that code how it is registering a decorator type, specifically, a custom Log Decorator.

That `LogDecorator` class can be implemented as the following simple code which is simply logging info about the command handler being executed and whether it was successful or not.

```
public class LogDecorator<TRequest, TResponse>
    : IAsyncRequestHandler<TRequest, TResponse>
        where TRequest : IAsyncRequest<TResponse>
{
    private readonly IAsyncRequestHandler<TRequest, TResponse> _inner;
    private readonly ILogger<LogDecorator<TRequest, TResponse>> _logger;

    public LogDecorator(
        IAsyncRequestHandler<TRequest, TResponse> inner,
        ILogger<LogDecorator<TRequest, TResponse>> logger)
    {
        _inner = inner;
        _logger = logger;
    }
    public async Task<TResponse> Handle(TRequest message)
    {
        _logger.LogInformation($"Executing command {_inner.GetType().FullName}");

        var response = await _inner.Handle(message);

        _logger.LogInformation($"Succeeded executed command {_inner.GetType().FullName}");
        return response;
    }
}
```

Just by implementing this decorator class and by decorating my pipeline with it, all the commands processed through MediatR will be logging information about it.

In a similar way, you could implement other decorators like a validator decorator, transaction decorator, or any other aspect or cross-cutting concern you would like to apply to commands when handling them.

For additional information on the Mediator pattern and the MediatR library, see the following references.

References – Mediator
The mediator pattern
https://en.wikipedia.org/wiki/Mediator_pattern
The decorator pattern
https://en.wikipedia.org/wiki/Decorator_pattern
MediatR
https://github.com/jbogard/MediatR
https://lostechies.com/jimmybogard/2015/05/05/cqrs-with-mediatr-and-automapper/
https://lostechies.com/jimmybogard/2013/12/19/put-your-controllers-on-a-diet-posts-and-commands/
https://lostechies.com/jimmybogard/2014/09/09/tackling-cross-cutting-concerns-with-a-mediator-pipeline/

https://lostechies.com/jimmybogard/2016/06/01/cqrs-and-rest-the-perfect-match/
https://lostechies.com/jimmybogard/2016/10/13/mediatr-pipeline-examples/
https://lostechies.com/jimmybogard/2016/10/24/vertical-slice-test-fixtures-for-mediatr-and-asp-net-core/
https://lostechies.com/jimmybogard/2016/07/19/mediatr-extensions-for-microsoft-dependency-injection-released/
FluentValidation
https://github.com/JeremySkinner/FluentValidation

Sagas

Why Sagas?

A Saga is a technique that can be used to handle out of order messages. It is similar (but not equal) to a process manager or a workflow, and typically means a long-running business process that could be implemented either with custom code or based on a service bus.

When designing processes with more than one remote call it is usually recommended to use sagas. The length of time is not important in many cases. Sometimes "a single second means a lifetime" and you might need a saga, as well.

Sagas and long running processes
A Saga on Sagas
https://msdn.microsoft.com/en-us/library/jj591569.aspx
Saga implementation patterns – variations
https://lostechies.com/jimmybogard/2013/03/21/saga-implementation-patterns-variations/
Saga definition and implementing a saga with NServiceBus
https://docs.particular.net/nservicebus/sagas/

Implementing Resilient applications

Handling Partial Failure

In distributed systems, like in a microservices based application, there is the ever-present risk of partial failure. Since clients and services are separate processes/containers, a service might not be able to respond in a timely way to a client's request. A service might be down because of a failure or for maintenance, the service might be overloaded and responding extremely slowly to requests or simply not accessible for a very short time because of network issues.

Consider, for example, the Order page from the *eShopOnContainers* sample application. Let's imagine that the Ordering microservice is unresponsive when the user tries to submit an order. A bad implementation of the client (if the client code is synchronous RPC and with no time-out) might block indefinitely waiting for a response. In addition to that bad user experience, every unresponsive wait will consume or block a thread which is something very valuable in high scalable applications because in the case of having many issues like the one exposed eventually the runtime would run out of threads and became globally unresponsive instead of just partially unresponsive, as show in figure X-XX below.

Partial failures
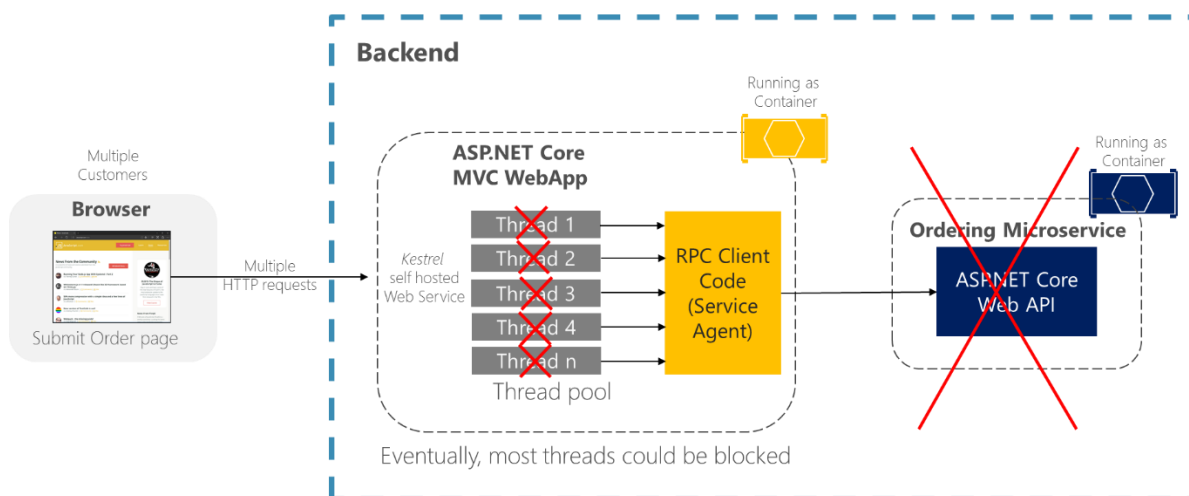
In a large microservice based application this partial failure can be very much amplified. Think about a system that receives millions of incoming calls per day which in turn fans out to many more millions of outgoing calls (let's suppose a ratio of 1:5) to tens of underlying or internal microservices as dependencies, as shown in figure X-XX.

Multiple distributed dependencies
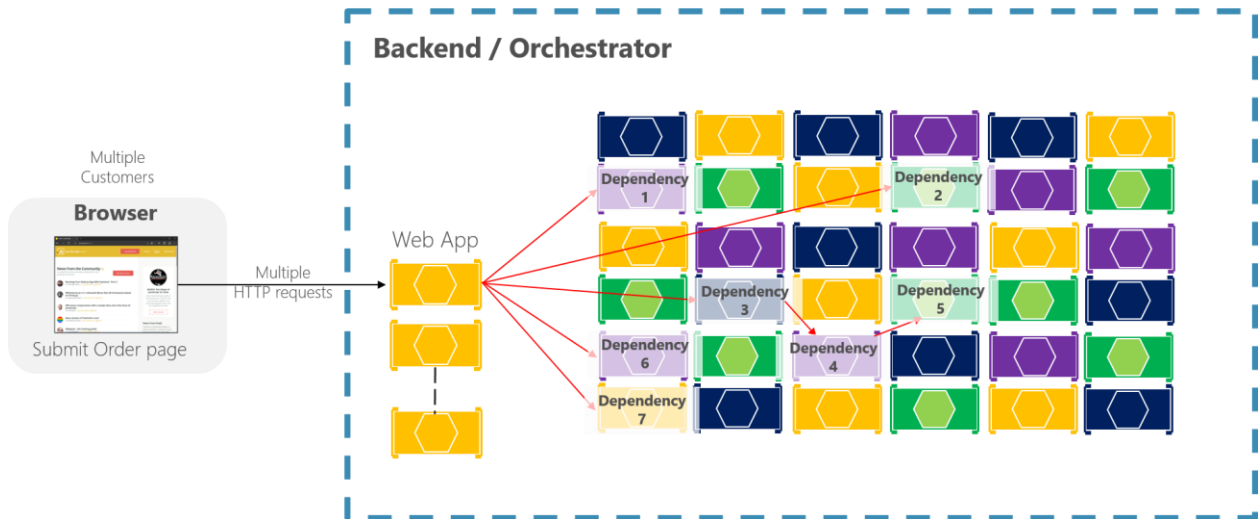
Intermittent failure is guaranteed that will happen in a distributed and cloud based system, even if every dependency itself has excellent availability and uptime.

Without taking steps to ensure fault tolerance, 50 dependencies each with 99.99% uptime would result in several hours of downtime/month because of the ripple effect.

When a single API dependency fails at high volume of requests with increased latency (causing blocked request threads) it can rapidly saturate all available request threads and take down the entire API or application.

Partial Failure Amplified in Microservices
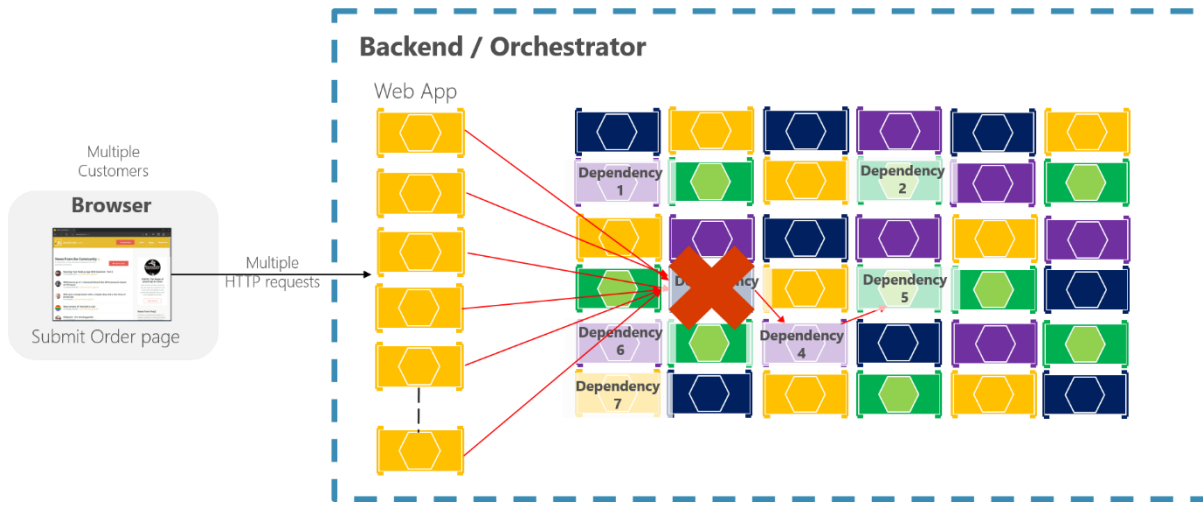
To prevent this problem, it is essential that you design your microservices and client applications to handle partial failures, that eventually will happen unavoidably in production systems. Therefore, it is a requirement of high volume, high availability applications to design and build resilient microservices and client applications into their architecture.

The strategies for dealing with partial failures include:

Circuit breaker pattern – Track the number of failed requests. If the error rate exceeds a configured limit, trip the circuit breaker so that further attempts fail immediately. If a large number of requests are failing, that suggests the service is unavailable and that sending requests is pointless. After a timeout period, the client should try again and, if successful, close the circuit breaker.

Provide fallbacks – Perform fallback logic when a request fails. For example, return cached data or a default value such as empty set of recommendations. However, this is not viable for updates/commands but mostly for queries.

Network timeouts – Never block indefinitely and always use timeouts when waiting for a response. Using timeouts ensures that resources are never tied up indefinitely.

Limiting the number of queued requests – Impose an upper bound on the number of outstanding requests that a client microservice can have with a particular service. If the limit has been reached, it is probably pointless to make additional requests, and those attempts need to fail immediately.

References – Implementing Resilient services
Adding Resilience and Optimizing Performance https://msdn.microsoft.com/en-us/library/jj591574.aspx

Implementing Retries Logic with Exponential Backoff

There are many possible approaches to implement retries logic with exponential backoff.

Retries with exponential backoff is a technique that assumes failure by nature and attempts to retry the operation, with an exponentially increasing wait time, until a maximum retry count has been reached. This technique embraces the fact that intermittently, cloud resources may be unavailable more than a few seconds, for any reason out of your control. In the case of microservices and containers in an orchestrator cluster, they could be moving to one or the other node by load balancing depending on how much load has every node/host in the cluster. Sometimes, if a node/VM crashes or the cluster is simply re-arranging resources, some HTTP requests might fail. Another example could be a database like SQL Azure where a database may be moved to another server at any time for load balancing reasons, causing the database from being unavailable for a few seconds.

Resilient Entity Framework Core Sql Connections

In regards the Azure SQL DB case, Entity Framework Core already provides internal database connection resiliency and retry logic, but you need to enable your desired execution strategy per DbContext connection if you want to have resilient EF Core connections.

For instance, the following code at the EF Core connection level is enabling resilient SQL connections that will re-try if it gets any temporal failure like it is possible when using Azure SQL Database or SQL Server deployed as a container if it is not ready.

```csharp
//Startup.cs from any ASP.NET Core Web API

public class Startup
{
    //Other code…

    //...
    public IServiceProvider ConfigureServices(IServiceCollection services)
    {
        //...
        services.AddDbContext<OrderingContext>(options =>
            {
                options.UseSqlServer(Configuration["ConnectionString"],
                                 sqlServerOptionsAction: sqlOptions =>
                                 {
                                 sqlOptions.EnableRetryOnFailure(maxRetryCount: 5,
                                         maxRetryDelay: TimeSpan.FromSeconds(30),
                                         errorNumbersToAdd: null);
                                 });
            });
        //...
    } //...
```

Execution strategies and explicit transactions using BeginTransaction and multiple DbContexts

When retries are enabled in EF Core connections, each operation you perform via EF Core becomes its own retriable operation, i.e. each query and each call to SaveChanges() will be retried as a unit if a transient failure occurs.

However, if your code initiates a transaction using *BeginTransaction()* you are defining your own group of operations that need to be treated as a unit, i.e. everything inside the transaction would need to be played back shall a failure occur. You will receive an exception like the following if you attempt to do this when using an execution strategy and you include "SaveChanges" fof multiple DbContexts.

System.InvalidOperationException: The configured execution strategy 'SqlServerRetryingExecutionStrategy' does not support user initiated transactions. Use the execution strategy returned by 'DbContext.Database.CreateExecutionStrategy()' to execute all the operations in the transaction as a retriable unit.

The solution, is to manually invoke the execution strategy with a delegate representing everything that needs to be executed. If a transient failure occurs, the execution strategy will invoke the delegate again, as in the following code implemented in *eShopOnContainers* when using two multiple DbContexts related to the *CatalogContext* and the *IntegrationEventLogContext* when updating a product and saving the ProductPriceChanged integration event that needs to use a different DbContext.

```
public async Task<IActionResult> UpdateProduct([FromBody]CatalogItem productToUpdate)
{
    //Other code…
    //Update current product
    catalogItem = productToUpdate;
    // Use of an EF Core resiliency strategy when using multiple DbContexts
    // within an explicit BeginTransaction():
    // See: https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-resiliency
    var strategy = _catalogContext.Database.CreateExecutionStrategy();

    await strategy.ExecuteAsync(async () =>
     {
        // Achieving atomicity between original Catalog database operation and the
        // IntegrationEventLog thanks to a local transaction
        using (var transaction = _catalogContext.Database.BeginTransaction())
        {
            _catalogContext.CatalogItems.Update(catalogItem);       DbContext #1
            await _catalogContext.SaveChangesAsync();

            //Save to EventLog only if product price changed       DbContext #2
            if (raiseProductPriceChangedEvent)
                await _integrationEventLogService.SaveEventAsync(priceChangedEvent);

            transaction.Commit();
        }
     });       Commit across multiple DbContexts and using an Execution Strategy
```

Implementing custom Http retries with exponential backoff

When implementing resiliency to handle possible HTTP scenarios in your own microservices, you need to implement your own implementation or retries with exponential backoff.

In addition to resource availability, the exponential backoff also needs to take into account the fact that the cloud provider may decide to limit or throttle availability of resources due to usage overload. For example, requesting too many connection requests very quickly may be viewed as a *Denial of Service attack* (DoS) by the cloud provider (like Azure). As result, backing off exponentially connection requests needs to provide a mechanism to scale back connection requests when a capacity threshold has been encountered.

As initial basic exploration, you could implement your own code from scratch with a utility exponential backoff class like this RetryWithExponentialBackoff.cs example plus sample code using it like this, and also shown in the following code.

```
public sealed class RetryWithExponentialBackoff
{
    private readonly int maxRetries, delayMilliseconds, maxDelayMilliseconds;

    public RetryWithExponentialBackoff(int maxRetries = 50, int delayMilliseconds = 200,
int maxDelayMilliseconds = 2000)
    {
        this.maxRetries = maxRetries;
        this.delayMilliseconds = delayMilliseconds;
        this.maxDelayMilliseconds = maxDelayMilliseconds;
    }

    public async Task RunAsync(Func<Task> func)
    {
        ExponentialBackoff backoff = new ExponentialBackoff(this.maxRetries,
                                        this.delayMilliseconds, this.maxDelayMilliseconds);
        retry:
        try
        {
            await func();
        }
        catch (Exception ex) when (ex is TimeoutException || ex is
                                                System.Net.Http.HttpRequestException)
        {
            Debug.WriteLine("Exception raised is: " + ex.GetType().ToString() + " –
              Message: " + ex.Message + " -- Inner Message: " + ex.InnerException.Message);
            await backoff.Delay();
            goto retry;
        }
    }
}
public struct ExponentialBackoff
{
    private readonly int m_maxRetries, m_delayMilliseconds, m_maxDelayMilliseconds;
    private int m_retries, m_pow;

    public ExponentialBackoff(int maxRetries, int delayMilliseconds,
                        int maxDelayMilliseconds)
    {
        m_maxRetries = maxRetries;
        m_delayMilliseconds = delayMilliseconds;
        m_maxDelayMilliseconds = maxDelayMilliseconds;
        m_retries = 0;
        m_pow = 1;
    }

    public Task Delay()
    {
        if (m_retries == m_maxRetries)
```

```
        {
            throw new TimeoutException("Max retry attempts exceeded.");
        }
        ++m_retries;
        if (m_retries < 31)
        {
            m_pow = m_pow << 1; // m_pow = Pow(2, m_retries - 1)
        }
        int delay = Math.Min(m_delayMilliseconds * (m_pow - 1) / 2,
                                                m_maxDelayMilliseconds);
        return Task.Delay(delay);
    }
}
```

The usage at the client C# application (another Web API client microservice, an ASP.NET MVC app or even a C# Xamarin app) would be pretty straightforward, something like the following code using the class HttpClient internally.

```
public async Task<Catalog> GetCatalogItems(int page,int take, int? brand, int? type)
{
    _apiClient = new HttpClient();
    var itemsQs = $"items?pageIndex={page}&pageSize={take}";
    var filterQs = "";

    var catalogUrl =
            $"{_remoteServiceBaseUrl}items{filterQs}?pageIndex={page}&pageSize={take}";
    var dataString = "";
    //
    // Using HttpClient with Retry and Exponential Backoff
    //
    var retry = new RetryWithExponentialBackoff();
    await retry.RunAsync(async () =>
    {
        // work with HttpClient call
        dataString = await _apiClient.GetStringAsync(catalogUrl);
    });

    return JsonConvert.DeserializeObject<Catalog>(dataString);
}
```

However, it is recommended to use more sophisticated and proven libraries like explained in the next section.

Implementing Http retries with exponential backoff with Polly

The recommended approach for retries with exponential backoff is to take advantage of more advanced .NET libraries like the open source library Polly.

Polly is a .NET resilience and transient-fault-handling library that allows developers to express policies such as Retry, Circuit Breaker, Timeout, Bulkhead Isolation, and Fallback in a fluent and thread-safe manner. Polly targets .NET 4.0, .NET 4.5 and .NET Standard 1.0 (supports .NET Core). It really has a very long list of resilient policies for most of the needs you might have including new policies in the latest release like Timeout, Bulkhead Isolation, Fallback and PolicyWrap.

The Retries policy is one of the fundamental policies and it is precisely the implementation approach used by *eShopOnContainers* when implementing HTTP retries by using Polly library.

You can implement an interface so you can inject and use either a regular/standard HttpClient functionality or a resilient version of HttpClient using Polly and depending on your desired retry policy configuration.

This is the interface implemented and used by *eShopOnContainers*.

```
public interface IHttpClient
{
    HttpClient Inst { get; }
    Task<string> GetStringAsync(string uri);
    Task<HttpResponseMessage> PostAsync<T>(string uri, T item);
    Task<HttpResponseMessage> DeleteAsync(string uri);
}
```

You will have a regular implementation if you don't want to use any resilient mechanism, when developing or testing simpler approaches, like the following:

```
public class StandardHttpClient : IHttpClient
{
    private HttpClient _client;
    private ILogger _logger;
    public HttpClient Inst => _client;
    public StandardHttpClient()
    {
        _client = new HttpClient();
        _logger = new LoggerFactory().CreateLogger(nameof(StandardHttpClient));
    }

    public Task<string> GetStringAsync(string uri) =>
        _client.GetStringAsync(uri);

    public Task<HttpResponseMessage> PostAsync<T>(string uri, T item)
    {
        var contentString = new StringContent(JsonConvert.SerializeObject(item),
                                        System.Text.Encoding.UTF8, "application/json");
        return _client.PostAsync(uri, contentString);
    }
    //… Other methods
}
```

And finally, and this is the interesting implementation, you would code another similar class but using Polly to implement your desired resilient mechanisms, in this case in regards the retry with exponential backoff functionality based on Polly's policies.

```
public class ResilientHttpClient : IHttpClient
{
    private HttpClient _client;
    private PolicyWrap _policyWrapper;
    private ILogger<ResilientHttpClient> _logger;
    public HttpClient Inst => _client;

    public ResilientHttpClient(List<ResiliencePolicy> policies,
                               ILogger<ResilientHttpClient> logger)
        {
            _client = new HttpClient();
            _logger = logger;

            // Add Policies to be applied
            _policyWrapper = Policy.WrapAsync(GeneratePolicies(policies));
        }
```

```
        private Policy CreateRetryPolicy(int retries,
                                        int backoffSeconds,
                                        bool exponentialBackoff) =>
        Policy.Handle<HttpRequestException>()
            .WaitAndRetryAsync(
            retries,  // number of retries
            retryAttempt => exponentialBackoff ?
                    TimeSpan.FromSeconds(Math.Pow(backoffSeconds, retryAttempt)) :
                    TimeSpan.FromSeconds(backoffSeconds),
            // on retry
            (exception, timeSpan, retryCount, context) =>
            {
                var msg = $"Retry {retryCount} with Polly's RetryPolicy " +
                    $"of {context.PolicyKey} " +
                    $"at {context.ExecutionKey}, " +
                    $"due to: {exception}.";
                _logger.LogWarning(msg);
                _logger.LogDebug(msg);
            }
        );

    private Task<T> HttpInvoker<T>(Func<Task<T>> action) =>
                                // Executes the action applying all
                                // the policies defined in the wrapper
                                _policyWrapper.ExecuteAsync(() => action());

    public Task<string> GetStringAsync(string uri) =>
                                HttpInvoker(() =>
                                        _client.GetStringAsync(uri));
    //Other Http methods, like PostAsync using HttpInvoker internally…
}
```

Basically, with Polly, you define a Retry policy with the number of retries, exponential backoff configuration and actions to take when getting http exceptions, like logging. In this case, it is configured so it will try the number of times specified when creating the policies when registering the types in the IoC container. Then, because of the Exponential Backoff configuration, whenever you get an HttpRequest exception it will also be waiting for an exponential time per each request, like waiting 2 seconds the first time, 4 seconds the second time, 8 seconds the third time and so on.

The important method is the **HttpInvoker<T>()** which is what is used when making Http request throguth this utility class, so it internally is executing the http request with **_policyWrapper.ExecuteAsync()** which is taking into account the retry policy.

Implementing the Circuit Breaker pattern

As introduced, you should handle faults that might take a variable amount of time to recover from, when connecting to a remote service or resource. This can improve the stability and resiliency of an application.

In a distributed environment, calls to remote resources and services can fail due to transient faults, such as slow network connections, timeouts, or the resources being overcommitted or temporarily unavailable. These faults typically correct themselves after a short period of time, and a robust cloud application should be prepared to handle them by using a strategy such as the Retry pattern.

However, there can also be situations where faults are due to unanticipated events, and that might take much longer to fix. These faults can range in severity from a partial loss of connectivity to the complete failure of a service. In these situations, it might be pointless for an application to continually retry an operation that is unlikely to succeed, and instead the application should quickly accept that the operation has failed and handle this failure accordingly.

Additionally, if a service is very busy, failure in one part of the system might lead to cascading failures, so it is critical for the system to handle these kinds of failures.

The purpose of the Circuit Breaker pattern is definitely different than the Retry pattern. The Retry pattern enables an application to retry an operation in the expectation that it'll succeed. The Circuit Breaker pattern prevents an application from performing an operation that is likely to fail. An application can combine these two patterns by using the Retry pattern to invoke an operation through a circuit breaker. However, the retry logic should be sensitive to any exceptions returned by the circuit breaker and abandon retry attempts if the circuit breaker indicates that a fault is not transient.

Implementing a Circuit Breaker pattern with Polly

In a similar way than when implementing retries, the recommended approach for circuit brakers implementation is to take advantage of mature .NET libraries like Polly.

The circuit breaker policy is also used by *eShopOnContainers* when implementing HTTP retries by using Polly library. It is in fact applying both policies to the same **ResilientHttpClient** utility class that you just reviewed with the retry policy as shown in the code below. So, whenever you use an object of type ResilientHttpClient for Http requests, you will be applying both those policies.

```csharp
public class ResilientHttpClient : IHttpClient
{
    private HttpClient _client;
    private PolicyWrap _policyWrapper;
    private ILogger _logger;
    public HttpClient Inst => _client;

    public ResilientHttpClient(List<ResiliencePolicy> policies,
                               ILogger<ResilientHttpClient> logger)
        {
            _client = new HttpClient();
            _logger = logger;

            // Add Policies to be applied
            _policyWrapper = Policy.WrapAsync(GeneratePolicies(policies));
        }
```

```csharp
    private Policy CreateCircuitBreakerPolicy(int exceptionsAllowedBeforeBreaking,
                                              int durationOfBreakInMinutes) =>
        Policy.Handle<HttpRequestException>()
        .CircuitBreakerAsync(
            exceptionsAllowedBeforeBreaking, //Number of exceptions before breaking circuit

            TimeSpan.FromMinutes(durationOfBreakInMinutes),
            (exception, duration) =>
            {
                // on circuit opened
                _logger.LogTrace("Circuit breaker opened");
            },
            () =>
            {
                // on circuit closed
                _logger.LogTrace("Circuit breaker reset");
            }
        );

    // Other code related to the Retry policy
    //…

    private Task<T> HttpInvoker<T>(Func<Task<T>> action) =>
        // Executes the action applying all
        // the policies defined in the wrapper
        _policyWrapper.ExecuteAsync(() => action());

    public Task<string> GetStringAsync(string uri) =>
        HttpInvoker(() =>
            _client.GetStringAsync(uri));
}
```

Basically, that code adds a policy to the Http wrapper that defines a circuit breaker which will be open (at that moment Http requests won't work but an exception will be raised) when you get more than "n" exceptions, like "5" which will be the number provided by the "exceptionsAllowedBeforeBreaking" parameter.

Circuit breakers should also be used to redirect requests to a fallback infrastructure in cases when you might have issues in a particular resource that is deployed in a different environment than the client application/service performing the http call, so if there's any outage in the datacenter impacting only your backend microservices but not your client applications, those client applications could open the "by default circuit" and redirect to the fallback services.

Using your ResilientHttpClient utility class

The way you use your *ResilientHttpClient* utility class would be pretty similar to the way you use the regular .NET HttpClient class, so in code like the following, in this case from a the eShopOnContainer MVC web application, at the Ordering ServiceAgent used by the OrderController.

Basically, the ResilientHttpClient object will be injected through the IHttpClient httpClient parameter at the constructor. Then used to perform regular Http requests.

```csharp
public class OrderingService : IOrderingService
{
    private IHttpClient _apiClient;
    private readonly string _remoteServiceBaseUrl;
    private readonly IOptionsSnapshot<AppSettings> _settings;
    private readonly IHttpContextAccessor _httpContextAccesor;

    public OrderingService(IOptionsSnapshot<AppSettings> settings,
                        IHttpContextAccessor httpContextAccesor,
                        IHttpClient httpClient)
    {
        _remoteServiceBaseUrl = $"{settings.Value.OrderingUrl}/api/v1/orders";
        _settings = settings;
        _httpContextAccesor = httpContextAccesor;
        _apiClient = httpClient;
    }

    async public Task<List<Order>> GetMyOrders(ApplicationUser user)
    {
        var context = _httpContextAccesor.HttpContext;
        var token = await context.Authentication.GetTokenAsync("access_token");

        _apiClient.Inst.DefaultRequestHeaders.Authorization = new
                System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
        var ordersUrl = _remoteServiceBaseUrl;
        var dataString = await _apiClient.GetStringAsync(ordersUrl);
        var response = JsonConvert.DeserializeObject<List<Order>>(dataString);

        return response;
    }

    // Other methods
    //…

    async public Task CreateOrder(Order order)
    {
        var context = _httpContextAccesor.HttpContext;
        var token = await context.Authentication.GetTokenAsync("access_token");

        _apiClient.Inst.DefaultRequestHeaders.Authorization = new
                System.Net.Http.Headers.AuthenticationHeaderValue("Bearer", token);
        _apiClient.Inst.DefaultRequestHeaders.Add("x-requestid",
                                                order.RequestId.ToString());

        var ordersUrl = $"{_remoteServiceBaseUrl}/new";
        order.CardTypeId = 1;
        order.CardExpirationApiFormat();
        SetFakeIdToProducts(order);

        var response = await _apiClient.PostAsync(ordersUrl, order);

        response.EnsureSuccessStatusCode();
```

```
      }
  }
```

Thus, whenever the member object **_apiClient** is used, it will internally be using the wrapper class with the Polly's policies for the retries, the circuit breaker or any additional policy that you might want to apply from Polly's policies collection.

Testing the Retries with eShopOnContainers

In fact, whenever you start the *eShopOnContainers* solution in a Docker host, it needs to spin up multiple containers. Some of them are slower to start and get ready, like the SQL Server container especially the first time you deploy it into Docker and need to setup the images and the database, which could cause the rest of the services to initially throw Http exceptions even when you might be setting dependencies between containers at the docker-compose level, as introduced in previous sections. Those docker-compose dependencies between containers are just at the process level. The container entry point process might be started but SQL Server might not be ready for queries causing an error's cascade and you would be getting an exception when trying to consume that particular container.

Eventually, you could get the same behavior in transient network errors when deploying to the cloud and orchestrators/clusters that might be moving containers from one node/VM to another when balancing the cluster nodes.

The way *eShopOnContainers* is solving that issue is precisely with those implemented retries, and that is why that when starting the whole solution you could get log traces or warnings like the following.

```
"Retry 1 implemented with Polly's RetryPolicy, due to:
System.Net.Http.HttpRequestException: An error occurred while sending
the request. ---> System.Net.Http.CurlException: Couldn't connect to
server\n   at System.Net.Http.CurlHandler.ThrowIfCURLEError(CURLcode
error)\n   at…. Etc."
```

Testing the Circuit Breaker with eShopOnContainers

There are a few ways you can open the circuit and test it with *eShopOnContainers*.

1. The first option is to lower the allowed number of retries to 1 in the circuit breaker policy and re-deploy the whole solution into Docker. With a single retry it is probably that you will break/open the circuit and get an error.
2. Use a custom middleware implemented in the Ordering microservice that, when enabled, is catching all HTTP requests and returning a 500. You can configure/enable that middleware by calling the url "failing", like:

a. **GET /failing** -> Returns the current state of the middleware, if "enabled" it will return 500. If disabled, it won't do anything.
   b. **GET /failing?enable** -> It enables this middleware
   c. **GET /failing?disable** -> It disables this middleware

For instance, once the application is up and running, let's enable the middleware by running the following URL at any browser. Note the specific 5102 port for the Ordering microservice.

```
http://localhost:5102/failing?enable
```



```
FailingMiddleware enabled. Further requests will return HTTP 500
```
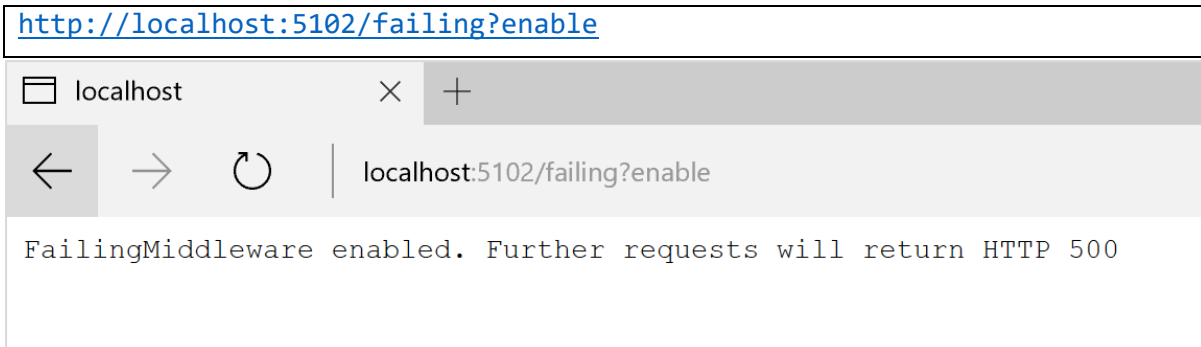
Then, check its status with http://localhost:5100/failing


```
FailingMiddleware is enabled
```
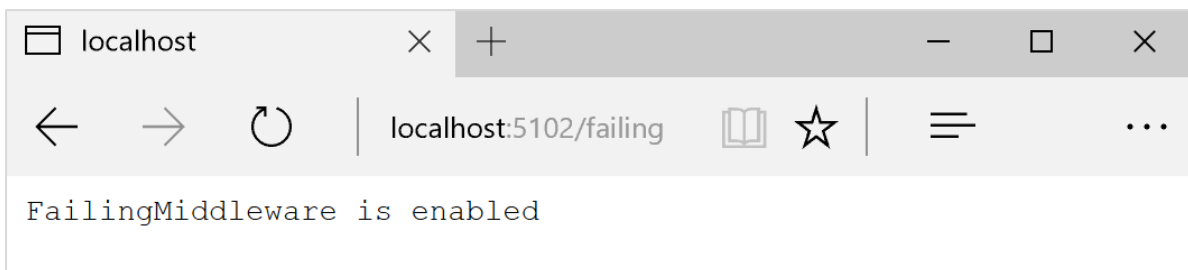
At this point, the Ordering microservice will be responding an HTTP 500 whenever you call it.

Therefore, after a few retrys from the MVC web application it will break/open the circuit when trying to call the Ordering microservice which you can do by placing an order from the web application.

You can check below that the code implemented at the MVC web application has a catch when trying to place an order and in the case of an open-circuit, it will show a user friendly meesage telling him/her to wait.

```
[HttpPost]
public async Task<IActionResult> Create(Order model, string action)
{
    try
    {
        if (ModelState.IsValid)
        {
            var user = _appUserParser.Parse(HttpContext.User);
            await _orderSvc.CreateOrder(model);

            //Redirect to historic list.
            return RedirectToAction("Index");
        }
}
```

```
        }
    catch(BrokenCircuitException ex)
    {
        ModelState.AddModelError("Error",
                "It was not possible to create a new order, please try later on");
    }
    return View(model);
}
```

So, after the retry policy tries several times to run the Http request but getting Http errors, when the number or tries reaches the maximum number of the circuit breaker policy (in this case, 5 times), then you get the "BrokenCircuitException" and show that friendly message.
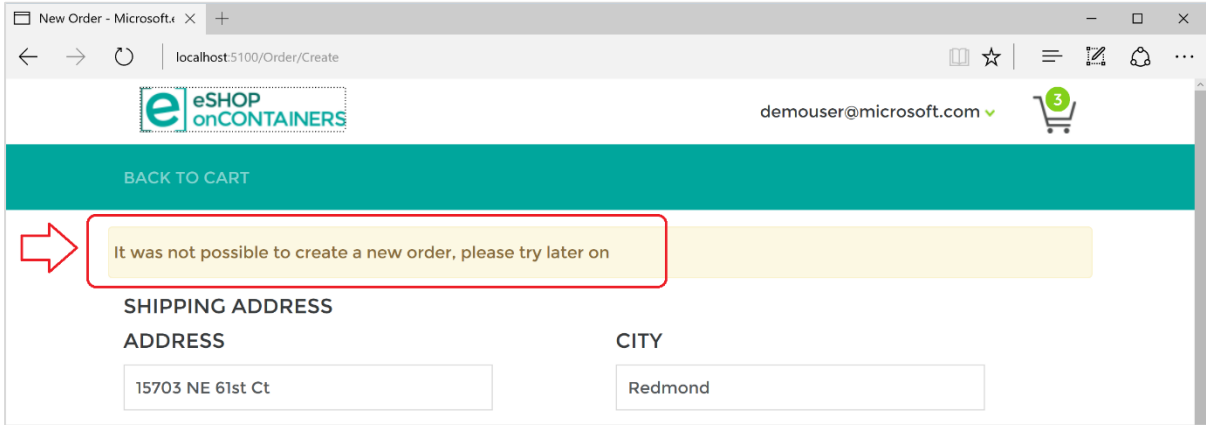
Based on that you could implement different logic when you want to break/open the circuit or even try a different Http request against a different backend microservice if you had a fallback datacenter or redundant backend system.

References – Resiliency
Retry pattern https://docs.microsoft.com/en-us/azure/architecture/patterns/retry
Entity Framework Core Connection Resiliency https://docs.microsoft.com/en-us/ef/core/miscellaneous/connection-resiliency
Polly (.NET resilience and transient-fault-handling library) https://github.com/App-vNext/Polly
Circuit Breaker pattern https://docs.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

Health Monitoring

Microservices based applications often use heartbeats or health checks to enable their performance monitors, schedulers, and orchestrators to keep track of multitudes of services. Without services being able to send back some sort of signal that "I'm alive," either by request or by regular pinging, your application might face many risks when deploying updates or simply detecting failures too late and not being able to stop cascading failures that can end up in major outages.

Health monitoring can allow near-real-time information about the state of your containers and microservices. Doing so, you can easily obtain health information and correct potential issues before they cascade and cause massive outages.

In the typical model, services send reports based on their status, and that information is aggregated to provide an overall view of your application's microservices health state and you can even provide health info to your orchestrator's cluster if using an orchestrator, so it can act accordingly. If you invest in high-quality health reporting that captures your custom conditions, you can detect and fix issues for your running application much more easily.

Health monitoring is critical to multiple aspects of operating microservices and is especially important when orchestrators perform partial application upgrades in phases, as mentioned in next sections.

Implementing Health Checks in ASP.NET Core services

When developing an ASP.NET Core microservice or web application you can use a new library named HealthChecks which is being developed by the ASP.NET team (as of early April 2017, is in ALPHA state available at GitHub).

This library is very easy to use and provides what you need to validate that any specific external resource (like a SQL Server database or remote API) is up and running and working proplery. With this API you can also decide what means for you that "this resource is working".

In order to use this library effectively, you basically need to first adopt/use the library into your microservices/containers and second, have a frontend application querying for the health reports.

Lest's focus first on how you use the HealthLibrary within your backend microservices/containers.

Using the HealthChecks library in your backend ASP.NET microservices

In terms of code, you can see how the HealthCheck is used in the eShopOnContainers services.

First thing you need to define is what is going to mean your "Healthy status" per each microservice.

In this specific example, our microservices/containers are going to be "healthy" if the microservice API itself is accessible via HTTP plus if its related SQL Server database is also available.

In the upcoming future, and as a normal usage, you will be able to install this library through a NuGet package. But in its current state you need to download a compile the code as part of your solution.

Basically, clone the code available at https://github.com/aspnet/HealthChecks and copy the following folders within your solution.

src/common

src/Microsoft.AspNet.HealthChecks

src/Microsoft.Extensions.HealthChecks

src/Microsoft.Extensions.HealthChecks.Data

For instance, in image X-XX is how you can see the HealthLibrary being used as a Building-Block to be used by any for any microservices.
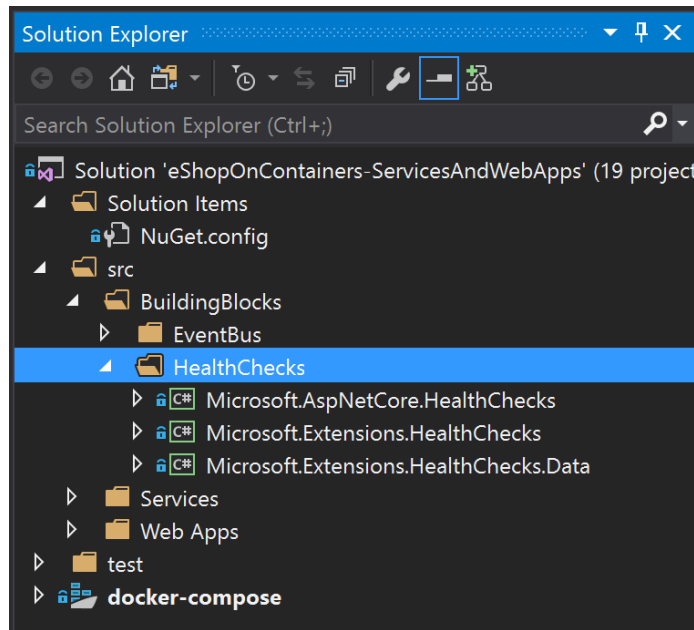
Figure X-XX. ASP.NET Core HealthChecks library source code in your solution

Again, take into account that in the upcoming future you'll be able to simply intall the library into your ASP.NET Core projects thorugh NuGet and you won't need to get the source code of it and compile it as part of your solution.

The first thing to do from each backend microservice project is actually to add the reference to the three HealthCheck libraries. After that, you need to add the HealthCheck actions you actually want to do within that microservices which are basically dependencies with other microservices (HttpCheck) or databases (currently SqlCheck). You do that within the Startup.cs class of each ASP.NET microservice or ASP.NET web application. For instance, in the following code you can see how the Catalog microservice has a dependency on its SQL database.

```
// Startup.cs from Catalog.api microservice
//
public class Startup
{
  public void ConfigureServices(IServiceCollection services)
  {
    // Add framework services.
    services.AddHealthChecks(checks =>
    {
        checks.AddSqlCheck("Catalog_Db", Configuration["ConnectionString"]);
    });
    // Other services
  }
}
```

However, in the case of the MVC web application, it has multiple dependencies on the rest of the microservices, so it has one AddUrlCheck() per each.

```
// Startup.cs from the MVC web app
public class Startup
{
```

```
    public void ConfigureServices(IServiceCollection services)
    {
      services.AddMvc();
      services.Configure<AppSettings>(Configuration);

      services.AddHealthChecks(checks =>
      {
          checks.AddUrlCheck(Configuration["CatalogUrl"]);
          checks.AddUrlCheck(Configuration["OrderingUrl"]);
          checks.AddUrlCheck(Configuration["BasketUrl"]);
          checks.AddUrlCheck(Configuration["IdentityUrl"]);
          checks.AddUrlCheck(Configuration["CallBackUrl"]);
      });
    }
}
```

Thus, a microservice won't provide a healthy status until all of its checks are healthy, too.

If you don't have any dependency like a database, you should then just add a "AlwaysOK" check, like it is done in the eShopOnContainers Basket.api microservice as there's still not a Redis health check provider.

```
services.AddHealthChecks(checks =>
{
    checks.AddValueTaskCheck("Always OK", () => new
                    ValueTask<IHealthCheckResult>(HealthCheckResult.Healthy("Ok")));
});
```

In order for each service or web app to be able to expose the HealthCheck endpoint, it has to enable the "UserHealthChecks(*[url_for_health_checks]*)" extension method at the WebHostBuilder level in the main method of the Program class of your ASP.NET Core service or web app, right after "UseKestrel()", like in the following code.

```
namespace Microsoft.eShopOnContainers.WebMVC
{
    public class Program
    {
        public static void Main(string[] args)
        {
            var host = new WebHostBuilder()
                .UseKestrel()
                .UseHealthChecks("/hc")
                .UseContentRoot(Directory.GetCurrentDirectory())
                .UseIISIntegration()
                .UseStartup<Startup>()
                .Build();
            host.Run();
        }
    }
}
```

Basically, how end-to-end things are working underneath is like this process. Each microservice exposes the endpoint /hc. That endpoint is created by the HealthCheck library ASP.NET Core middleware and when that endpoint is invoked it will run all the Health Checks configured within AddHealthChecks() at the startup.cs of that service.

The way each service or web app should be configured is by adding all its Http or Database dependencies as one "AddHealthCheck()" per each. E.i. the MVC depends on many services.

That healthcheck method expects a port or a path. That port of path will be the endpoint to be used to check the health state of the service. For instance, in the case of the Catalog microservice/container we're using the path "/hc".

Querying your microservices to report about their health status

With all the prvious configuration in place, once the microservice/container is up and running deployed into Docker, you could directly check from a browser if it is healthy, provided that you are publishing the container port out of the Docker Host, of course, so you can access through "localhost" or the external docker host IP.
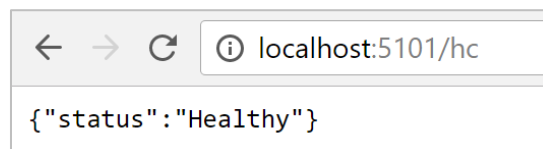
Figure X-XX. Checking Health status of a single service/containers

With that test you can see that the Catalog.api microservice/container is healthy, returning a HTTP 200. But that also means that internally it also checked its dependency with the SQL Server databas and that heclth check was also healthy.

Watchdogs

A watchdog is a separate service that can watch health and load across services, and report health about the microservices. This can help prevent errors that would not be detected based on the view of a single service. Watchdogs also are a good place to host code that can perform remediation actions for known conditions without user interaction.

There is a sample health check reports web available at eShopOnContainers that is the simplest Watchdog you could have as it just shows the states. Usually it is recommended to take further proactive actions when detecting unhealthy states.

As shown in figure X-XX, that mentioned Watchdog web app in eShopOncontainers shows the health status of all the different microservices and web applications that compose eShopOnContainers.

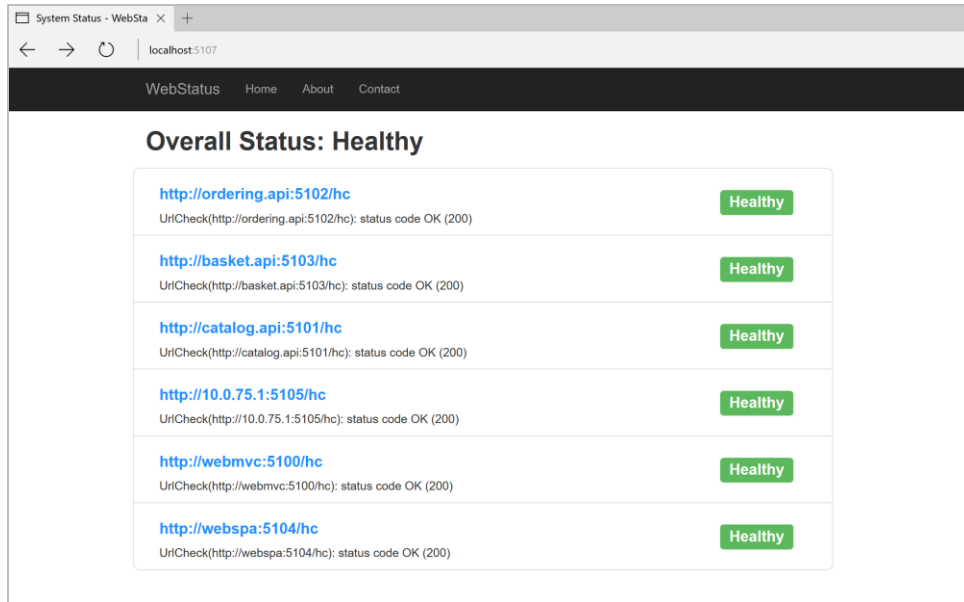Figure X-XX. Sample HealthCheck report web application in eShopOnContainers

In summary, the ASP.NET middleware of the ASP.NET Core HealthChecks library provides a single health check endpoint per microservice that will execute all the health checks defined within it and will return its health state depending on all of those checks.

It is implemented in a way that it is extensible through new health checks to future external resources and allows to report of the health status taking into account multiple service or applicarion dependencies and then take actions based on those health checks.

Future Health Checks for the ASP.NET HealthChecks library

In a similar fashion than you can now use a SQL Server healthcheck, in the future it is expected to have additional health checks for other external resources like Redis Cache, other databases, etc.

Orchestrators using Health Checks

To discover the availability of your microservices, the orchestrators (like Docker Swarm, Kubernetes, Service Fabric, etc.) periodically sends pings, attempts connections, or sends requests to test the microservices/containers. These tests are called, as introduced, health checks.

When an orchestrator determines that a container is unhealthy, it stops routing requests to that instance and might try to stop the container and create a new one in its place. If a container's health check reports "unhealthy", the orchestrator creates a new instance of a similar container in its place.

For instance, most orchestrators can utilize a health check to manage zero-downtime deployments. Only when the status changes to "healthy" will it start routing traffic to container instances.

Health monitoring is especially important when an orchestrator performs an application upgrade. Some orchestrators (like Azure Service Fabric) update the services in phases, like 1/5 of the cluster surface per application upgrade which is usually called "upgrade domains". After each partial upgrade (upgrade domain of the service) is upgraded and is available to the users, that upgrade domain must pass health checks before the deployment moves to the next upgrade domain.

The more health checks that are incorporated into your code, the more resilient your services are to deployment issues.

Another aspect of service health is reporting metrics from the service which is an advanced capability of some orchestrators' Health model, like in Service Fabric. Metrics are important when using an orchestrator because they are used to balance resource usage. Metrics also can be an indicator of system health. For example, you might have an application that has many microservices, and each instance reports a "requests per second" (RPS) metric. If one service is using more resources than another service, the orchestrator could move service instances around the cluster, to try to maintain even resource utilization.

Note that if using Azure Service Fabric, it provides its own Health Monitoring model which is more advanced than simple Healthchecks.

Advanced monitoring visualization, analysis, and alerts

The final part of monitoring is visualizing the event stream, reporting on service performance, and alerting when an issue is detected. You can use different solutions for this aspect of monitoring.

You can use simple custom applications showing the state of your services, like the custom page shown using the explained ASP.NET Core Healthchecks, or you could use more advanced tools like *Azure Application Insights* and *Operations Management Suite* to alert based on the stream of events.

Finally, you can use *Microsoft Power BI* or a third-party solution like *Kibana* or *Splunk* to visualize the data if you were storing all the event streams.

References – Resiliency
ASP.NET Core Healthchecks https://github.com/aspnet/HealthChecks/ (Alpha)
Service Fabric health monitoring https://docs.microsoft.com/en-us/azure/service-fabric/service-fabric-health-introduction
Azure Application Insights https://azure.microsoft.com/en-us/services/application-insights/
Operations Management Suite https://www.microsoft.com/en-us/cloud-platform/operations-management-suite

Implementing Graceful Shutdowns

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

------------------------------- *Further implementation details* -------------------------------

REFERENCES:
https://github.com/aspnet/Hosting/blob/dev/src/Microsoft.AspNetCore.Hosting.Abstractions/IApplicationLifetime.cs

https://shazwazza.com/post/aspnet-core-application-shutdown-events/

Securing .NET microservices and web applications

Authentication

It's often necessary for resources and APIs exposed by a service to be made available to certain trusted users or clients, but not to others. The first step to making these sorts of API-level trust decisions is authentication. Authentication is the process of reliably ascertaining a user's identity or granted permissions.

In microservice scenarios, authentication is typically handled centrally. If using an API gateway, the API gateway is a good place to authenticate, as shown in figure X-X. Make sure, when using this approach, that the individual micro-services can't be reach directly (without the API gateweay) unless additional security is in place to authenticate messages as coming from the gateway or not.
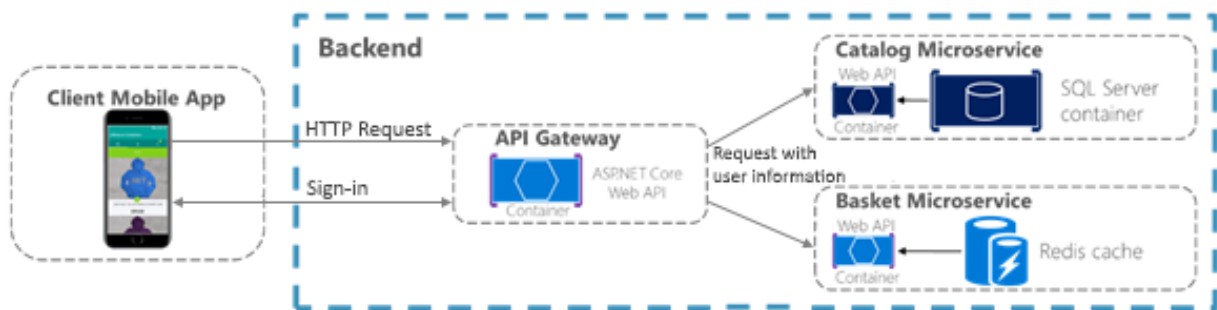
Figure X-XX. Centralized authentication with an API gateway

If different services are accessed directly, an authentication service like Azure Active Directory or a dedicated authentication micro-service acting as a *Security Token Service* (STS) can be used to authenticate users. Trust decisions are shared between services with security tokens or cookies (which can be shared between applications, if needed, in ASP.NET Core with data protection services). This pattern is illustrated in figure X-X, below.

Figure X-XX. Authentication by identity microservice; trust shared via authorization token

ASP.NET Core Identity

ASP.NET Core's primary mechanism for identifying an app's users is the ASP.NET Core Identity membership system. ASP.NET Core Identity handles storing user information (including sign-in information, roles, and claims) in a data store configured by the developer. Typically, the ASP.NET Core Identity data store will be a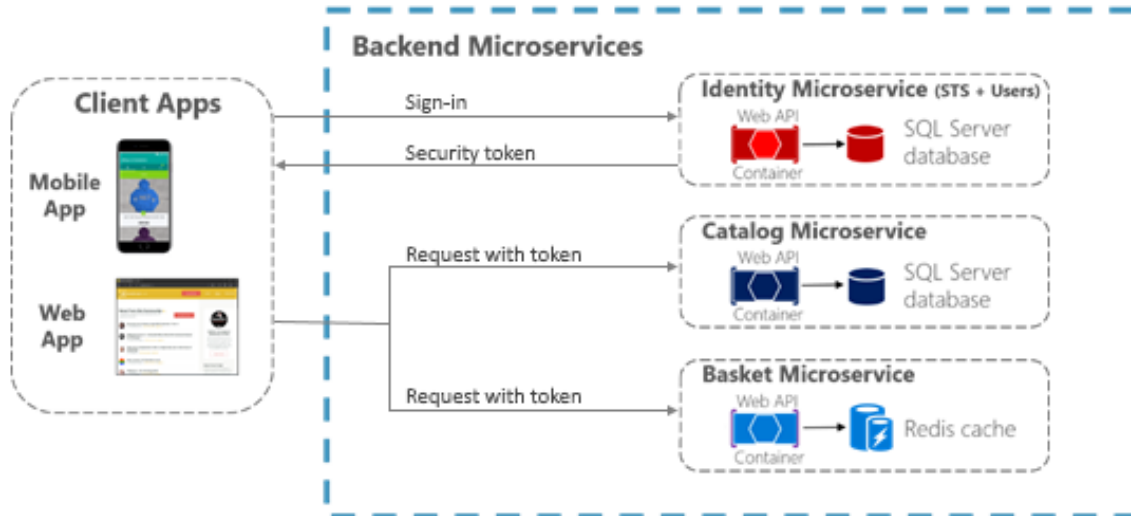n EntityFramework store (provided in the *Microsoft.AspNetCore.Identity.EntityFrameworkCore* package), though custom stores or other third-party packages can be used to store Identity information in Azure dtable storage, DocumentDB, or other locations.

This code (taken from the ASP.NET Core Web Application new project template with individual user account authentication selected) demonstrates configuring ASP.NET Core Identity (using EntityFramework.Core) in `Startup.ConfigureServices`:

```
services.AddDbContext<ApplicationDbContext>(options =>
    options.UseSqlServer(Configuration.GetConnectionString("DefaultConnection")));
services.AddIdentity<ApplicationUser, IdentityRole>()
    .AddEntityFrameworkStores<ApplicationDbContext>()
    .AddDefaultTokenProviders();
```

Once configured, ASP.NET Core Identity is enabled by calling `app.UseIdentity` in the service's `Startup.Configure` method.

Using ASP.NET Code Identity enables several useful scenarios:

- Local user information can be created and stored using Identity's UserManager type (`userManager.CreateAsync`).
- Users can be authenticated using the SignInManager type (`signInManager.SignInAsync` to sign in directly, or `signInManager.PasswordSignInAsync` to verify the user's password first).
- Signed-in users will have their user information and claims stored in a cookie for use in subsequent requests.

- Middleware is registered in the ASP.NET Core application's pipeline to read user information from cookies so that subsequent requests from a browser will include a signed-in user's identity and claims.

Beyond simple sign-in scenarios, ASP.NET Core Identity also supports two-factor authentication.

For authentication scenarios that make use of a local user data store and that persist identity between requests via cookies (as is appropriate for typical MVC web applications), ASP.NET Core Identity is a recommended solution.

External Authentication

ASP.NET Core also supports using external authentication providers to log in users via OAuth 2.0 flows. This means that users can log in using existing authentication processes from external providers (such as Microsoft, Google, Facebook, or Twitter) and associate those identities with an ASP.NET Core Identity in your application.

This can be done by including the appropriate authentication middleware in your app's HTTP request processing pipeline. This middleware is responsible for handling requests to return URI routes from the authentication provider, capturing identity information, and making it available via the `SignInManager.GetExternalLoginInfo` method.

Common external authentication providers and their associated NuGet pacakges are shown in the table below. In all cases, the middleware is registered with a call to a registration method similar to `app.Use{ExternalProvider}Authentication` in startup.Configure. These registration methods take an options object containing application ID and secret information, as needed by the specific provider used. The external authentication providers require app registration (as explained in ASP.NET Core documentation) for security reasons and so that they can inform the user what application is requesting access to their identity.

Provider	Package
Microsoft	Microsoft.AspNetCore.Authentication.MicrosoftAccount
Google	Microsoft.AspNetCore.Authentication.Google
Facebook	Microsoft.AspNetCore.Authentication.Facebook
Twitter	Microsoft.AspNetCore.Authentication.Twitter

Once the necessary middleware is registered in Startup.Configure, users can be prompted to log in from a controller action by returning a ChallengeResponse with an AuthenticationProperties argument created from authentication provider's name and a redirect URL:

```
var properties = _signInManager.ConfigureExternalAuthenticationProperties(provider,
redirectUrl);
return Challenge(properties, provider);
```

The `redirectUrl` represents the URL that the external provider should redirect to once the user has authenticated. This should be an action that will sign in the user based on external identity information, as in this simplified sample code sample:

```
// Sign in the user with this external login provider if the user already has a login.
var result = await _signInManager.ExternalLoginSignInAsync(info.LoginProvider,
info.ProviderKey, isPersistent: false);
if (result.Succeeded)
{
    return RedirectToLocal(returnUrl);
}
else
{
    ApplicationUser newUser = new ApplicationUser
    {
        // The user object may be constructed with whatever specific claims are
        // returned by the external authentication provider used, or can
        // be created by gathering input from the user.
        UserName = info.Principal.FindFirstValue(ClaimTypes.Name),
        Email = info.Principal.FindFirstValue(ClaimTypes.Email)
    };

    var identityResult = await _userManager.CreateAsync(newUser);
    if (identityResult.Succeeded)
    {
        identityResult = await _userManager.AddLoginAsync(newUser, info);
        if (identityResult.Succeeded)
        {
            await _signInManager.SignInAsync(newUser, isPersistent: false);
        }

        return RedirectToLocal(returnUrl);
    }
}
```

Note that all of the code necessary to sign in with an external provider is present in the ASP.NET Core web application template project if "individual accounts" authentication is chosen at project creation time (as shown in figure X-X, below), so when starting from that template the only code you should need to add is the middleware registration for the specific external authentication providers you wish to use.
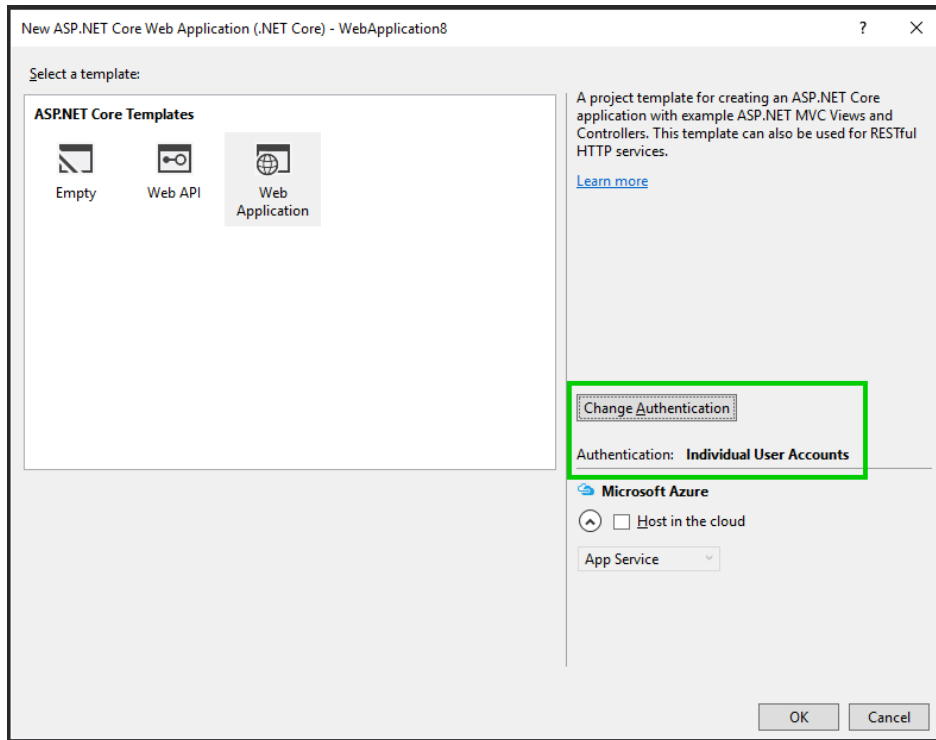
Other External Authentication Providers

In addition to the external authentication providers listed previously, there are third-party packages available which provide middleware for using [many more](#) external authentication providers. It is also possible, of course, to create your own external authentication middleware.

Authenticating with Bearer Tokens

Authenticating with ASP.NET Core Identity (or Identity plus external authentication providers) works well for many web application scenarios in which storing user information in a cookie is appropriate. In other scenarios, though, cookies are not a natural means of persisting and transmitting data.

For example, an ASP.NET Core web API which exposes RESTful endpoints that may be accessed by single page applications (SPAs), native clients, or even other web APIs will typically want to use bearer token authentication instead since these scenarios don't work with cookies, but can easily retrieve a bearer token and include it in the authorization header of future requests.

To enable token authentication, ASP.NET Core supports several options for using [OAuth 2.0](#) and [OpenID Connect](#).

Authenticating Against an OpenID Connect or OAuth 2.0 Identity Provider

If user information is stored in Azure Active Directory or another identity solution that supports OpenID Connect or OAuth 2.0, the *Microsoft.AspNetCore.Authentication.OpenIdConnect* package can be used to authenticate using the OpenID Connect workflow.

To [authenticate against Azure Active Directory](#), for example, an ASP.NET Core web application can use middleware from that package as follows:

```
// Configure the OWIN pipeline to use OpenID Connect auth.
app.UseOpenIdConnectAuthentication(new OpenIdConnectOptions
{
    ClientId = Configuration["AzureAD:ClientId"],
    Authority = String.Format(Configuration["AzureAd:AadInstance"],
Configuration["AzureAd:Tenant"]),
    ResponseType = OpenIdConnectResponseType.IdToken,
    PostLogoutRedirectUri = Configuration["AzureAd:PostLogoutRedirectUri"]
});
```

Note that the configuration values are Azure Active Directory values which are created when your application is registered as an Azure AD client. If necessary, a single client ID can be shared between multiple micro-services in an application which may all need to authenticate users signed on by Azure Active Directory.

When using this workflow, the ASP.NET Core Identity middleware is not needed as all user information storage and authentication is handled by Azure Active Directory.

Issuing Security Tokens from an ASP.NET Core Service

If, rather than using an external identity provider, you prefer to issue security tokens for local ASP.NET Core Identity users, there are a couple good third-party libraries that can help.

IdentityServer4 is an OpenID Connect provider for ASP.NET Core that integrates easily with ASP.NET Core Identity to enable issuing security tokens from an ASP.NET Core service. The IdentityServer4 documentation has in-depth instructions for using the library, but the basic steps to using IdentityServer4 to issue tokens are:

1. Call app.UseIdentityServer() in Startup.Configure to add IdentityServer4 to the application's HTTP request processing pipeline (so that it can serve requests to OpenID Connect and OAuth2 endpoints like /connect/token).
2. Configure IdentityServer4 in Startup.ConfigureServices with a call to services.AddIdentityServer. Further configuration is needed with subsequent calls to setup the following IdentityServer4 concepts:
   a. Credentials used for signing
   b. Identity and API resources that users may request access to
      i. API resources represent some protected data or functionality which a user might gain access to with an access token. An example of an API resource would be a web API (or set of APIs) that require authorization to call.
      ii. Identity resources represent information (claims) which are given to a client to identify a user. This could include their name, email address, or other claims.
   c. Clients that will be connecting to request tokens
   d. ASP.NET Core Identity (or an alternative user store)

When specifying clients and resources for IdentityServer4 to use, you can just pass an IEnumerable<T> of the appropriate type to methods which take in-memory client or resource stores or, for more complex scenarios, provide client or resource provider types via dependency injection.

A sample configuration of IdentityServer4 using in-memory resources and clients provided by a custom IClientStore might look like this:

```
// Add IdentityServer services
```

```
services.AddSingleton<IClientStore, CustomClientStore>();

services.AddIdentityServer()
    .AddSigningCredential("CN=sts")
    .AddInMemoryApiResources(MyApiResourceProvider.GetAllResources())
    .AddAspNetIdentity<ApplicationUser>();
```

Consuming Security Tokens

Authenticating against an OpenID Connect endpoint or issuing your own security tokens covers some scenarios, but what about a service that simply needs to limit access to users with valid security tokens (provided by a different service)?

Authentication middleware which handles JWT tokens is available in the *Microsoft.AspNetCore.Authentication.JwtBearer* package.

Simple use of the middleware might look like this (make sure this call precedes calls to ASP.NET Core's MVC middleware (app.UseMvc)):

```
app.UseJwtBearerAuthentication(new JwtBearerOptions()
{
    Audience = "http://localhost:5001/",
    Authority = "http://localhost:5000/",
    AutomaticAuthenticate = true
});
```

The parameters in this usage are:

- **Audience** represents the intended recipient of the incoming token or the resource that the token grants access to. If the value specified in this parameter doesn't match the aud parameter in the token, the token will be rejected because it was meant to be used for accessing a different resource.
- **Authority** is the address of the token-issuing authentication server. The JWT bearer authentication middleware will use this URI to find and retrieve the public key that can be used to validate the token's signature. It will also confirm that the iss parameter in the token matches this URI.
- **AutomaticAuthenticate** is a boolean value indicating whether the user defined by the token should be automatically logged in.
- **RequireHttpsMetadata** is not used in the code snippet above, but is useful for testing purposes. In real-world deployments, JWT bearer tokens should always be passed only over HTTPS.

With this middleware in place, JWT tokens will automatically be extracted from authorization headers, deserialized, validated (using the audience and authority parameters specified), and stored as user information to be referenced later by MVC actions or authorization filters/middleware.

The JwtBearerAuthentication middleware can also support more advanced scenarios (such as using a local certificate to validate a token if the authority is not available) by specifying a TokenValidationParameters object in the JwtBearerOptions object.

Authorization

After authentication, ASP.NET Core web APIs often need to authorize access. This process allows a service to make APIs available to some authenticated users, but not to all.

Authorization can be done based on users' roles or based on custom policy (which may include inspecting claims or any other heuristics).

Restricting access to an ASP.NET Core MVC route is as easy as applying an `[Authorize]` attribute to the action method (or to the controller's class if all the controller's actions require authorization), as shown in the sample code below.

```
public class AccountController : Controller
{
    public ActionResult Login()
    {
    }

    [Authorize]
    public ActionResult Logout()
    {
    }
}
```

By default, adding an `[Authorize]` attribute will limit access to authenticated users. To further restrict an API to be available for only specific users, the attribute can be expanded to specify required roles or policies that users must satisfy.

Role-Based Authorization

ASP.NET Core Identity has a built-in concept of roles. In addition to users, ASP.NET Core Identity stores information about different roles used by the application and keeps track of which users are assigned to which roles. These assignments may be changed programmatically with the `RoleManager` type (which adjusts roles in persisted storage) and `UserManager` type (which can assign/un-assign users from roles).

When authenticating with JWT bearer tokens, ASP.NET Core's JWT bearer authentication middleware will populate a user's roles based on 'role' claims found in the token.

To limit access to an MVC action (or controller) to users in specific roles, just include a 'Roles' parameter in the authorize header:

```
[Authorize(Roles = "Administrator, PowerUser")]
public class ControlPanelController : Controller
{
    public ActionResult SetTime()
    {
    }

    [Authorize(Roles = "Administrator")]
    public ActionResult ShutDown()
    {
    }
}
```

In the code snippet above, only users in the Administrator or PowerUser roles can access APIs in the `ControlPanel` controller (like the `SetTime` action). The `ShutDown` API is further restricted to allow access only to users in the Administrator role.

To require a user be in multiple roles, use multiple Authorize attributes, as shown in figure <mark>X.X</mark> below.
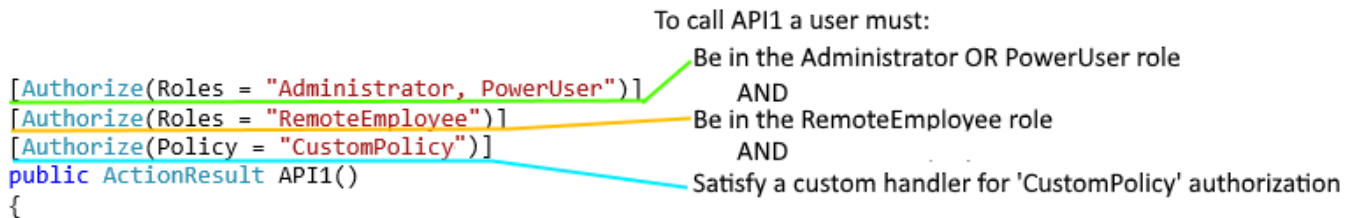
Policy-Based Authorization

Custom authorization rules can also be written using <u>authorization policies</u>. An overview is given here, but more detail is available in the online <u>ASP.NET Authorization Workshop</u>.

Custom authorization policies are registered in the `Startup.ConfigureServices` method using the `service.AddAuthorization` method. This method takes an action method that configures an `AuthorizationOptions` argument.

```
services.AddAuthorization(options =>
{
    options.AddPolicy("AdministratorsOnly", policy => policy.RequireRole("Administrator"));
    options.AddPolicy("EmployeesOnly", policy => policy.RequireClaim("EmployeeNumber"));
    options.AddPolicy("Over21", policy => policy.Requirements.Add(new
MinimumAgeRequirement(21)));
});
```

As shown above, policies can be associated with different types of requirements. After registering these policies, they can be applied to an action or controller by passing the policy's name as the `Policy` argument of the Authorize attribute (for example: `[Authorize(Policy=EmployeesOnly)]`). Note that policies may have multiple requirements, not just one as in the above samples.

The first `AddPolicy` call above is just an alternative way of authorizing by role. If `[Authorize(Policy=AdministratorsOnly)]` is applied to an API, then only users in the Administrator role will be able to access it.

The second `AddPolicy` call demonstrates an easy way of requiring that a particular claim be present for the user. The `RequireClaim` method also optionally takes expected values for the claim. If values are specified, then the requirement is only met if the user both has a claim of the correct type and with one of the specified values. When using the JWT bearer authentication middleware, all JWT properties will be available as user claims.

The most interesting policy shown here is the last one because it uses a custom authorization requirement. By using custom authorization requirements, developers can have a great deal of control over how authorization is performed. For this to work, the developer must implement a couple types:

- A requirement type deriving from `IAuthorizationRequirement` which contains fields specifying the details of the requirement (an age field, for example, for our sample MinimumAgeRequirement type).

- A handler implementing `AuthorizationHandler<T>` where T is the type of `IAuthorizationRequirement` that the handler can satisfy. The handler must implement the `HandleRequirementAsync(AuthorizationHandlerContext context, T requirement)` method which checks whether a given context (which contains information about the user) satisfies the requirement.
    - o   If the user meets the requirement, a call to `context.Succeed(requirement)` will indicate that the user is authorized.
    - o   If there are multiple ways that a user might satisfy an authorization requirement, multiple handlers can be created.

Note that in addition to registering custom policy requirements with `AddPolicy` calls, custom requirement handlers also need to be registered via dependency injection (`services.AddTransient<IAuthorizationHandler, MinimumAgeHandler>()`).

An example of a custom authorization requirement and handler for checking a user's age (based on a DateOfBirth claim) is available in the ASP.NET Core authorization documentation.

References – Securing .NET Applications
ASP.NET Core Authentication
https://docs.microsoft.com/en-us/aspnet/core/security/authentication/identity
ASP.NET Core Authorization
https://docs.microsoft.com/en-us/aspnet/core/security/authorization/introduction

Safe storage of app secrets during development

To connect with protected resources and other services, ASP.NET Core applications will typically need to use connection strings, passwords, or other credentials containing sensitive information. These sensitive pieces of information are called 'secrets'. It is a best practice to not include secrets in source code and certainly not to store secrets in source control. Instead, ASP.NET Core's configuration model should be used to read the secrets from more secure locations.

Be sure to use separate secrets for accessing development and staging resources from those used for accessing production resources (as different individuals will need access to those different sets of secrets).

To store secrets used during development, common approaches are to either store secrets as environment variables or with ASP.NET Core's Secret Manager tool. For more secure storage in production environments, micro-services can store secrets in an Azure Key Vault.

Secrets from Environment Variables

One easy way to keep secrets out of source code is for developers to set string-based secrets as environment variables on their development machines. When using environment variables to store secrets with hierarchical names (those nested in configuration sections), include the full hierarchy of the secret's name in the environment variable name, delimited with colons (:).

For example, setting an environment variable `Logging:LogLevel:Default = Debug` would be equivalent to a configuration value read from the following JSON file:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug"
    }
  }
}
```

To access these values from environment variables, the application just needs to call `AddEnvironmentVariables` on its `ConfigurationBuilder` when constructing an `IConfigurationRoot` object.

Note that environment variables are generally stored as plain text, so if the machine or process with the environment variables are compromised, the environment variables' values will be visible.

Secrets using the Secret Manager

ASP.NET Core's [Secret Manager](#) tool provides another method of keeping secrets out of source code.

To use the Secret Manager tool, include a tools reference (`<DotNetCliToolReference>`) to the *Microsoft.Extensions.SecretManager.Tools* package in your project file. Once that reference is present and has been restored, '`dotnet user-secrets`' can be used to set the value of secrets from the command line. These secrets will be stored in a JSON file in the user's profile directory (details vary by OS), away from source code.

Note that secrets set by the SecretManager are organized by the UserSecretsId of the project using the secrets, so be sure to set the UserSecretsId property in your project file. The string used is unimportant, as long as it's unique.

```
<PropertyGroup>
  <UserSecretsId>UniqueIdentifyingString</UserSecretsId>
</PropertyGroup>
```

Using secrets stored with the Secret Manager in an application is similar to using secrets stored as environment variables: just call `AddUserSecrets<T>` on the `ConfigurationBuilder` to include secrets for the application in its configuration. The generic parameter T should be a type from the assembly the UserSecretesId was applied to (usually using `<Startup>` is fine).

Using Azure Key Vault to protect secrets in production time

Because secrets stored as environment variables or by the Secret Manager are still stored locally (and unencrypted) on the machine, a more secure option for storing secrets is [Azure Key Vault](#). Azure Key Vault provides a secure, central location for storing keys and secrets.

The *Microsoft.Extensions.Configuration.AzureKeyVault* package allows an ASP.NET Core application to read configuration information from Azure Key Vault. To start using secrets from an Azure Key Vault, you will need to follow these steps:

1. Register your application as an Azure AD application (access to Key Vaults is managed by Azure AD). This can be done through the Azure management portal or, if you would like your application to authenticate with a certificate (instead of a password/client secret), you can use the New-AzureRmADApplication PowerShell cmdlet.
   a. If you use the *New-AzureRmADApplication* cmdlet to register the application and wish to authenticate with a certificate, you should pass the raw cert data as a base 64 string as the CertValue parameter. The certificate registered with Key Vault only needs to contain your public key (your application will use the private key).
2. Give the registered application access to the key vault by creating a new service principal. This can be done with PowerShell as follows:
   a. `$sp = New-AzureRmADServicePrincipal -ApplicationId "<Application ID guid>"`
   b. `Set-AzureRmKeyVaultAccessPolicy -VaultName "<VaultName>" - ServicePrincipalName $sp.ServicePrincipalNames[0] -PermissionsToSecrets all -ResourceGroupName "<KeyVault Resource Group>"`
3. Include the key vault as a configuration source in your application by calling the `IConfigurationBuilder.AddAzureKeyVault` extension method when creating an `IConfigurationRoot`.
   a. Note that calling `AddAzureKeyVault` will require the application ID that was registered and given access to the key vault in steps (1) and (2).

Currently, .NET Standard (and also .NET Core) supports getting configuration information from an Azure Key Vault using a client ID and client secret. .NET Framework applications may alternatively use an overload of `IConfigurationBuilder.AddAzureKeyVault` that takes an X509 certificate in place of the client secret. Work is in-progress to make that overload available on .NET Standard/.NET Core. Until the `AddAzureKeyVault` overload accepting a certificate is available, ASP.NET Core application can access an Azure Key Vault with certificate-based authentication by explicitly creating a `KeyVaultClient`, as shown here:

```
// Configure Key Vault client
var kvClient = new KeyVaultClient(new KeyVaultClient.AuthenticationCallback(async
(authority, resource, scope) =>
{
    var cert = // Get certificate from local store/file/key vault etc., as needed
    // From the Microsoft.IdentityModel.Clients.ActiveDirectory pacakge
    var authContext = new AuthenticationContext(authority, TokenCache.DefaultShared);
    var result = await authContext.AcquireTokenAsync(resource,
            // From the Microsoft.Rest.ClientRuntime.Azure.Authentication pacakge
            new ClientAssertionCertificate("<Application ID>", cert));
    return result.AccessToken;
}));

// Get configuration values from Key Vault
var builder = new ConfigurationBuilder()
    .SetBasePath(env.ContentRootPath)
    // Other configuration providers go here.
    .AddAzureKeyVault("<KeyValueUri>", kvClient, new DefaultKeyVaultSecretManager());
```

Note that the call to `AddAzureKeyVault` comes at the end of configuration provider registration in this sample. It is a best practice to register Azure Key Vault as the last configuration provider so that it has

an opportunity to override configuration values from previous providers and so that no configuration values from other sources override those from the key vault.

<table>
<tr><td>References – Securing .NET Applications</td></tr>
<tr><td>Using Azure Key Vault to protect application secrets
https://docs.microsoft.com/en-us/azure/guidance/guidance-multitenant-identity-keyvault
Safe storage of app secrets during development
https://docs.microsoft.com/en-us/aspnet/core/security/app-secrets
Configuring data protection
https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/overview
Key management and lifetime
https://docs.microsoft.com/en-us/aspnet/core/security/data-protection/configuration/default-settings#data-protection-default-settings

https://github.com/aspnet/Configuration/tree/dev/src/Microsoft.Extensions.Configuration.DockerSecrets</td></tr>
</table>

Conclusions

Key takeaways

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