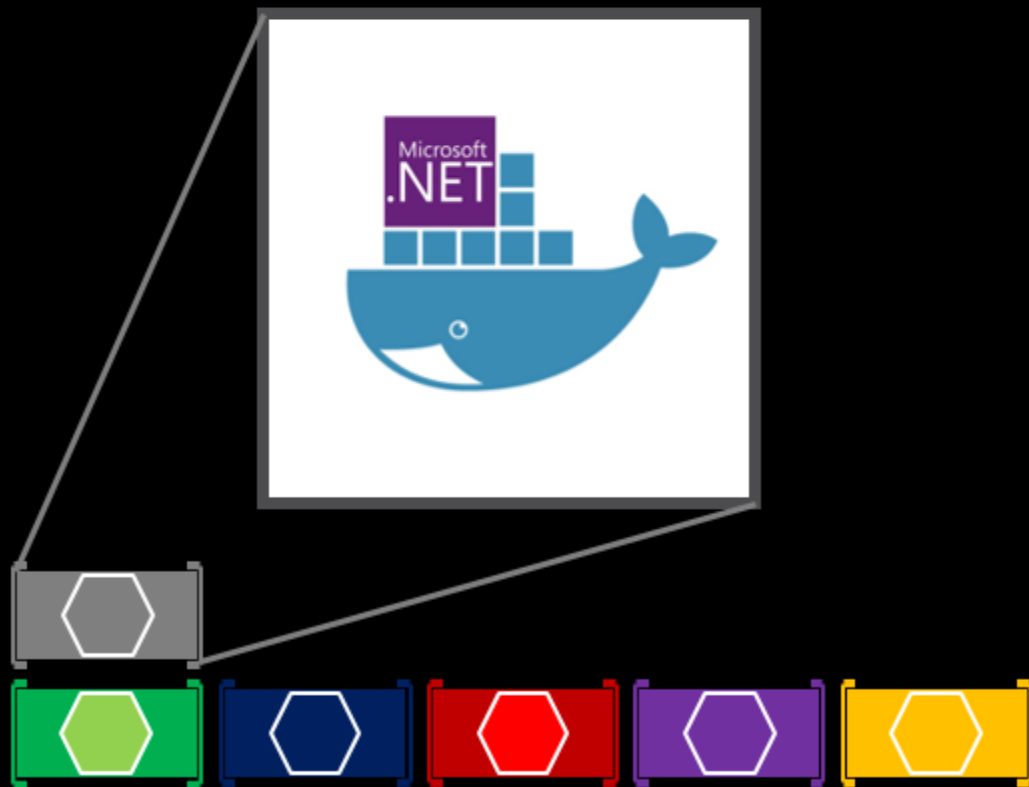# Architecting and Developing Containerized and Microservice based .NET Applications

Cesar de la Torre
Microsoft Corp.

Author:

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

Participants and reviewers (TBD):

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

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

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

Michael Friis, Product Manager, Docker Inc

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

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

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

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

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

Contents

Summary

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

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

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

Purpose

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

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

Containerized Docker Application Lifecycle with Microsoft Platform and Tools
https://aka.ms/dockerlifecycleebook

Who should use this guide

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

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

How you can use this guide

tbd

Introduction to containers and Docker

What are containers?

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

Real-life shipping containers are used transport goods by ship, train, or truck; they look the same on the outside regardless of the goods being transported inside them. Software containers are similar – they are simply a standard unit of software that behaves the same on the outside regardless of what code and dependencies are included on the inside. This enables developers and IT Professionals to transport them across environments with little or no modifications to the implementation, regardless of different configurations for each environment.

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

When running 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 Docker containers.

Inconsistent environment setups can create problems when deploying applications. By running an app or service inside a container, you avoid most of the issues associated with inconsistent environments.

Another important benefit when using containers is the ability to quickly instance any container. For example, 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.

In short , the main benefits provided by containers are isolation, portability, agility, scalability and control across the whole application lifecycle workflow. The most important benefit is the isolation provided between Dev and Ops.

What is Docker?

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

Docker is becoming the standard unit of deployment and is emerging as the de-facto standard implementation for containers as it is being adopted by most software platform and cloud vendors (Microsoft Azure, Amazon AWS, Google, etc.).
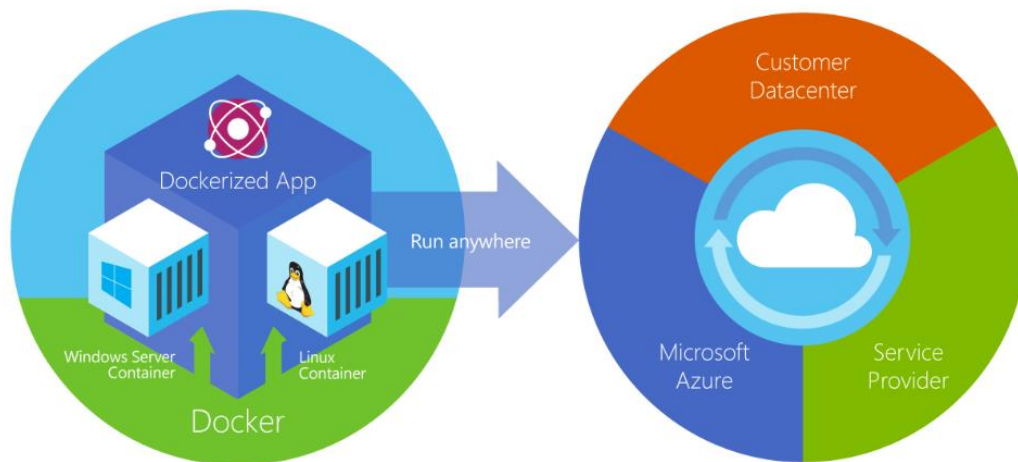
Figure 2-1. Docker deploys containers at all layers of the hybrid cloud

Docker containers can run natively on Linux and Windows. You can use MacOS as a development environment alternative to edit code or run the Docker CLI, but (at the time of this writing) containers do not run directly on MacOS. When targeting Linux containers, you will need a Linux host (typically a Linux VM) to run Linux containers on both Windows and MacOS development machines.

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

Related to Windows Containers, there are two types or runtimes:

Windows Server Containers – provide application isolation through process and namespace isolation technology. A Windows Server container shares a kernel with the container host and all containers running on the host.

Hyper-V Containers – expands on the isolation provided by Windows Server Containers by running each container in a highly optimized virtual machine. In this configuration, the kernel of the container host is not shared with the Hyper-V Containers, providing better isolation.

Summary

Comparing Docker containers with virtual machines

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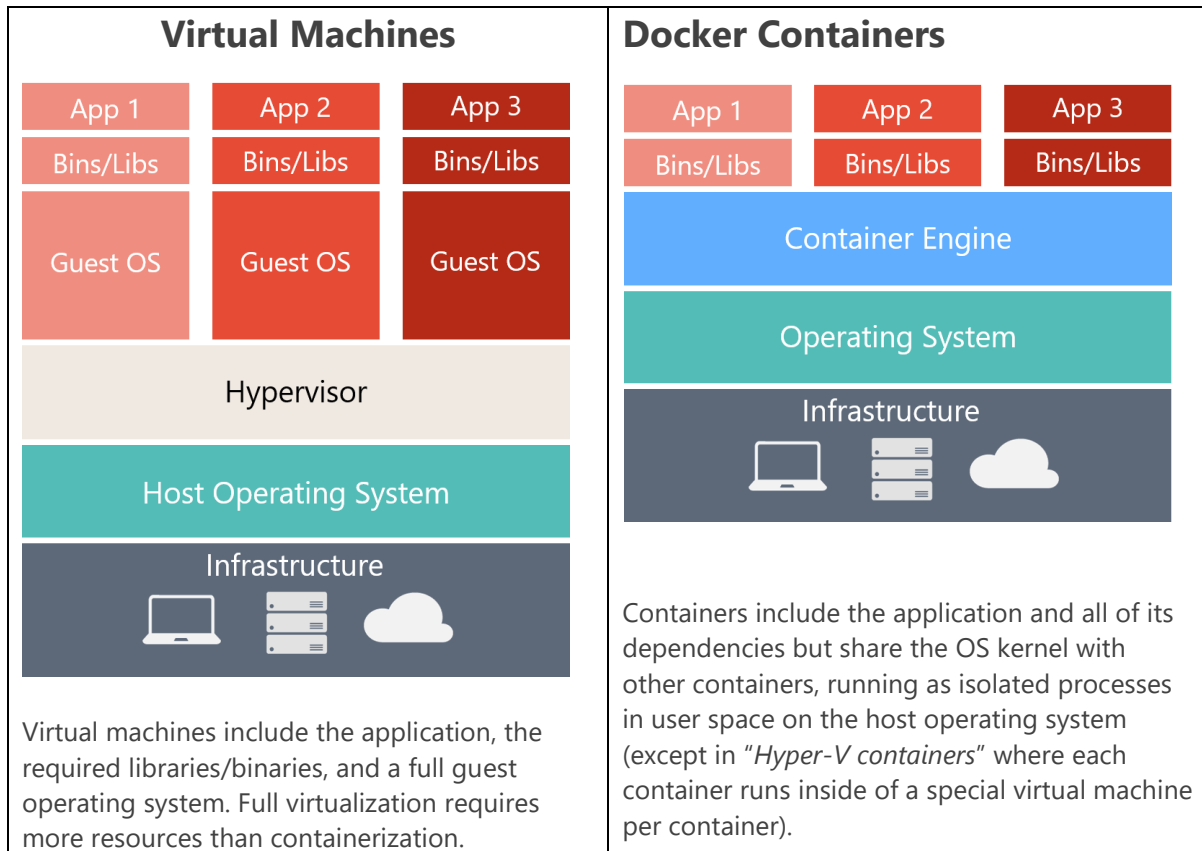
Figure 2-2. Comparison of traditional virtual machines to Docker containers

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

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

Docker also 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 do it on all the deployment targets (Dev/QA/Staging/Production, etc.).

What is Container as a Service?

Container as a Service (CaaS) is an IT managed and secured application environment of infrastructure and content provided as a service (elastic and pay as you go, like the basic cloud principles), with no upfront infrastructure design, implementation and investment per project, where 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 containers. Images (per version) are immutable.

In regards to host OS related updates, it usually is 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/

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

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

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

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

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

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

Registry: A Registry is a hosted service containing repositories of images which responds to the Registry API. The default registry (from Docker as an organization) can be accessed using a browser at Docker Hub or using the Docker search command. Therefore, a Registry usually contains many Repositories from multiple teams. 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 its 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 does everything that needs to be done to get it running. Docker-compose.yml files are used to build and run multi container applications, defining the build information as well the environment information for interconnecting the collection of containers.

Cluster: A Docker cluster pools together multiple Docker hosts and exposes them as a single virtual Docker host so it is able to scale up to many hosts very easily. Examples of Docker clusters can be created with Docker Swarm, Mesosphere DC/OS, Google Kubernetes and Azure Service Fabric. If using Docker Swarm, you typically 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 a much more. An orchestrator is responsible for running, distributing, scaling and healing workloads across a collection of nodes. Typically, Orchestrator products are the same products providing the cluster infrastructure like Mesosphere DC/OS, Kubernetes, Docker Swarm and Azure Service Fabric.

Basic Docker taxonomy: containers, images, and registries

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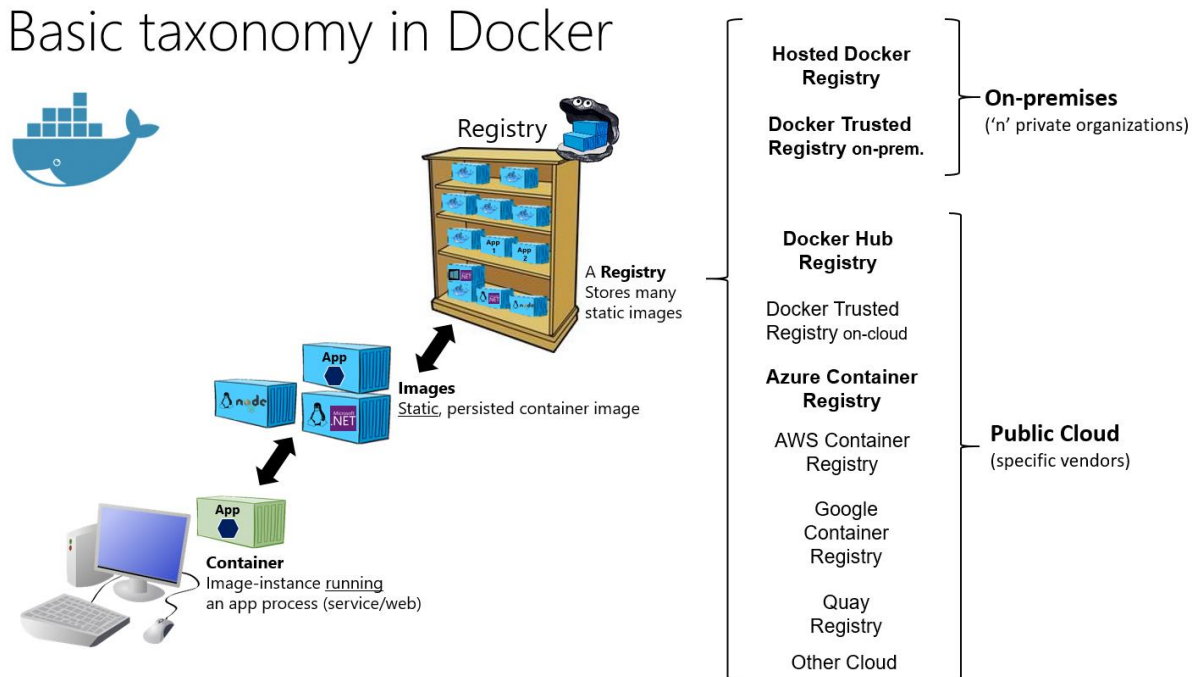
Figure 2-3. Taxonomy of Docker terms and concepts

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

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

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

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

You should use a private registry (an example of use of *Azure Container Registry*) if you want to:
- Tightly control where your images are being stored.
- Reduce network latency between the registry and the deployment nodes.
- Fully own your 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

Summary

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

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

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

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

- Your application currently uses .NET Framework and has strong dependencies on Windows
- You need to use Windows APIs not supported by .NET Core.
- You need to use third-party .NET libraries or NuGet packages not available for .NET Core.
- You need to use .NET technologies that are not available for .NET Core.

When to choose .NET Core for Docker containers

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

Cross-platform needs

Clearly, if your goal is to have an application (web/service) that 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. You could also avoid using a code editor and directly use the .NET Core command-line tools, available for all supported platforms.

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

Containers are commonly used in conjunction with a microservices architecture, although they can also be used to containerize web apps or services which follow any architectural pattern. You 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 architecture

.NET Core is the best candidate if you are embracing a microservices oriented system composed of multiple independent, dynamically scalable, stateful or stateless microservices. .NET Core is lightweight and its API surface can be minimized to the scope of the microservice. A microservices architecture also allows you to mix technologies across a service boundary, enabling a gradual 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 infrastructure platforms 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 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 performance and scalable systems

When your container-based system needs the best possible performance and scalability, .NET Core and ASP.NET Core are your best options. ASP.NET Core outperforms ASP.NET by a factor of 10, and it leads other popular industry technologies for microservices such as Java servlets, Go and node.js.

This is especially relevant for microservices architectures, where you could have hundreds of microservices/containers running. With ASP.NET Core can run your system with a much lower number of servers/VMs, ultimately saving costs in infrastructure and hosting.

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

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

When to choose .NET Framework for Docker containers

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

Current .NET Framework application directly migrated to a Docker container

You may want to use Docker containers for reasons other than targeting 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 frameworks to be ported to .NET Core - For a full list, take a look at CoreFX issues marked as port-to-core. Please note that this list doesn't represent a commitment from Microsoft to bring those components to .NET Core — they are simply capturing the desire from the community to do so. That being said, if you care about any of the components listed above, consider participating in the discussions on GitHub so that your voice can be heard. And if you think something is missing, please file a new issue in the CoreFX repository.

A need to use a platform/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 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	**.NET Core**	**.NET Core**
Monolithic deployment App	**.NET Core**	**.NET Framework** **.NET Core**
Best-in-class performance and scalability	**.NET Core**	**.NET Core**
Windows Server "brown-field" migration to containers	--	**.NET Framework**
Containers "green-field"	**.NET Core**	**.NET Core**
ASP.NET Core	**.NET Core**	**.NET Core** recommended .NET Framework is possible
ASP.NET 4 (MVC 5, Web API 2)	--	**.NET Framework**
SignalR services	**.NET Core** in upcoming releases	**.NET Framework** **.NET Core** in upcoming releases
WCF, WF and other traditional frameworks	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.

What OS to target with .NET containers
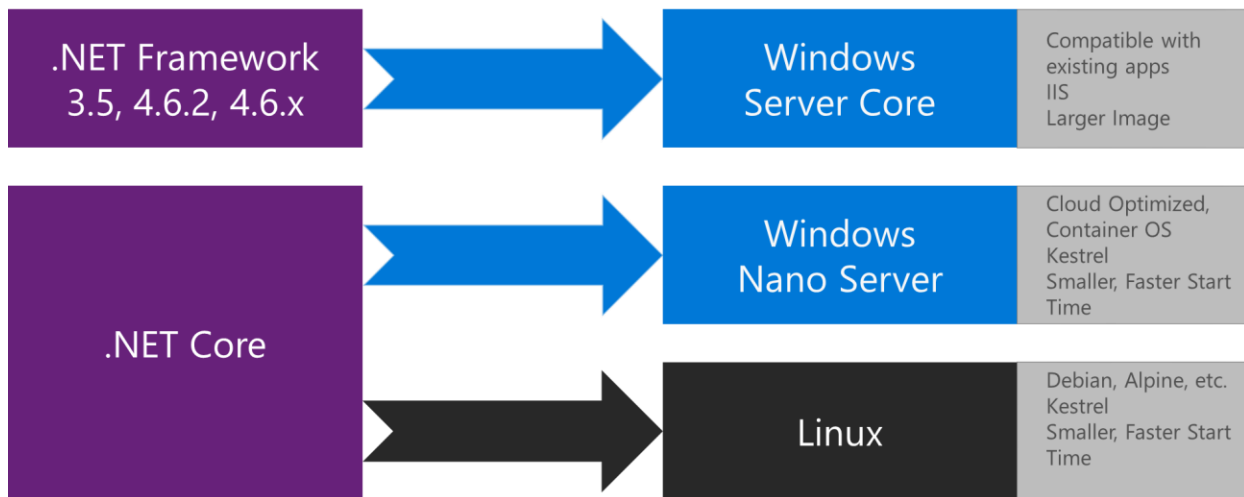
Figure X-X. OS to Target depending on .NET frameworks

However, you could also create your own Docker image from scratch in cases 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, because adding the ASP.NET Core stack in this smaller image would result in a bigger 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 Docker image optimizations per variant

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

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

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

Development: 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.

Build: When building, what's important is what's needed to compile your app. This includes the compiler and any other dependencies to optimize the binaries. This image isn't the image you deploy, rather it's an image you use to build the content you place into a production image. This image would be used in your continuous integration, or build environment. For instance, rather than installing all the dependencies directly on a build agent, the build agent would instance a build image to compile the application with all the dependencies required to build the app contained within the image. Your build agent only needs to know how to run this Docker image.

Production: What's important in production is how fast you can deploy and start your image. This image is small so it can quickly travel across the network from your Docker Registry to your Docker hosts. The contents are ready to run enabling the fastest time from Docker run to processing results. In the immutable Docker model, there's no need for dynamic compilation of code. The content you place in this image would be limited to the binaries and content needed to run the application. For example, the published output using dotnet publish contains the compiled 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**:1.1.0-sdk-msbuild**	.NET Core 1.1 with SDK included, on Linux

Introduction to Containers and Docker

Architecting containerized .NET applications with Docker and Azure

Vision

Architect and design scalable solutions with Docker in mind.

There are many great-fit use cases for containers, not just for microservices oriented architectures but also for regular services or web applications where 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. The scope of this document is not limited to microservices but to any Docker application lifecycle, therefore, it does not drill down deeply into microservices architecture because you can also use containers and Docker with regular Service Orientation, background tasks/jobs or even with monolithic application deployment approaches.

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

Common container design principles

Container equals a process

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

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.
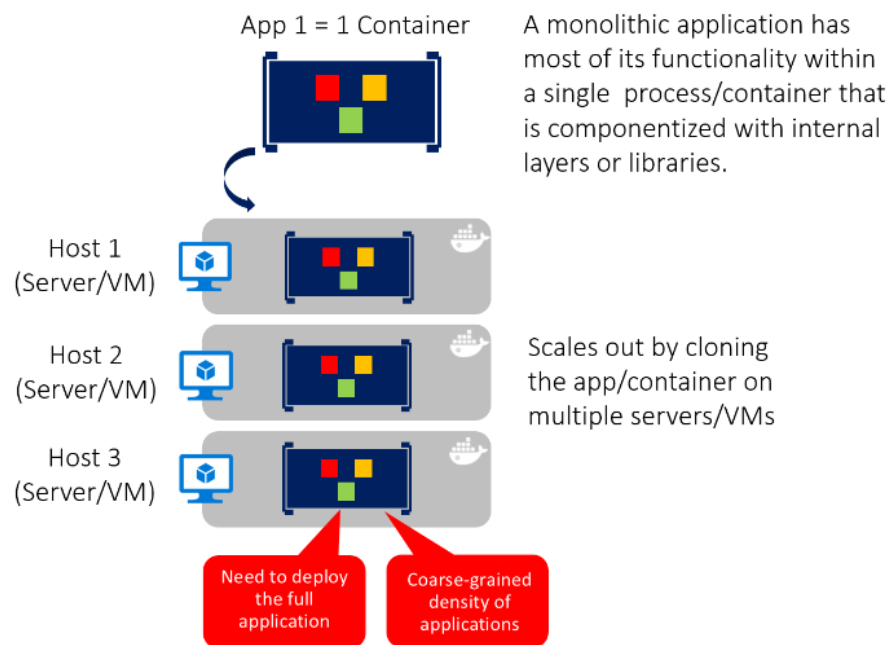
Figure X-X. Monolithic application architecture example

You can include multiple components/libraries or internal layers within each container, as illustrated in Figure X-X. But, following the container principal 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.

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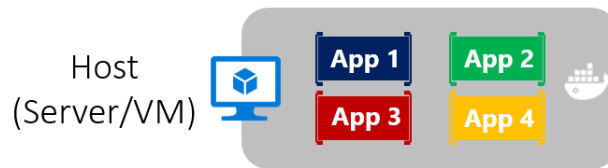
Figure X-X. Host running multiple apps/containers

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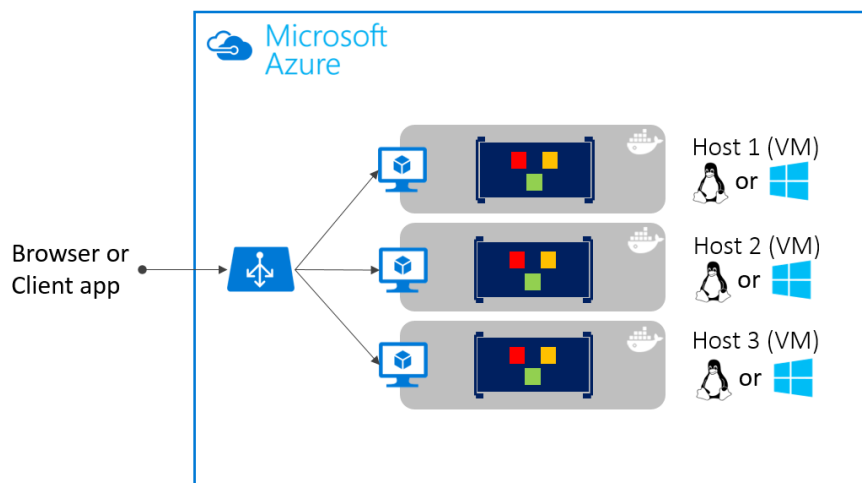
Figure 5-3. Multiple hosts scaling-out a single Docker application

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

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 are 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 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.
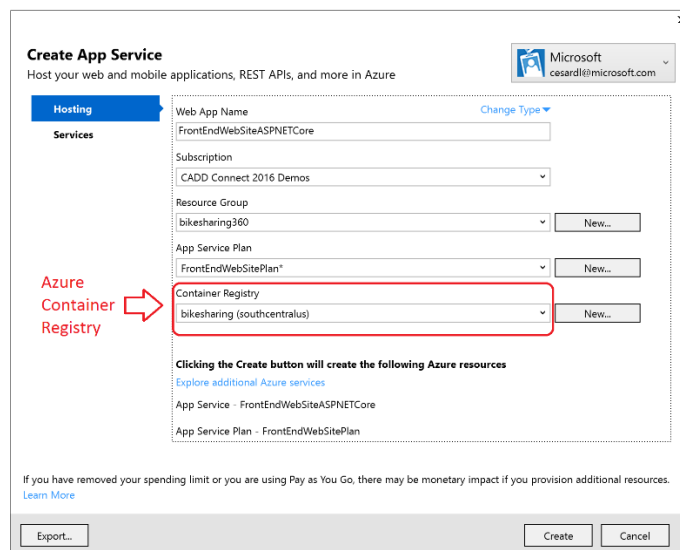
Figure X-X. Publishing a Container to Azure App Service from Visual Studio

Microsoft Platform and Tools for Docker

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

Containers are immutable; when compared to a VM, they don't disappear as a common occurrence. A VM may fail from dead processes, an overloaded CPU, or a full or failed disk. However, we expect the VM to always be available and RAID drives are commonplace to assure data is maintained despite drive failures.

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 copy memory will be durable. Containers, like processes, should be assumed to be duplicated or killed, or when managed with a container orchestrator, they may get moved.

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

To manage persistent data in Docker applications, there are common solutions:
- Data volumes which mount to the host as noted above.
- Data volume containers which provide shared storage across containers, using an external container that may cycle.
- Volume Plugins which mount volumes to remote locations, providing long term persistence.
- Remote data sources like SQL, NO-SQL databases or cache services like Redis.
- Azure Storage which provides geo distributable PaaS storage, providing the best of containers as long term persistence.

Data volumes are specially-designated directories within one or more containers that bypass the Union File System. 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.
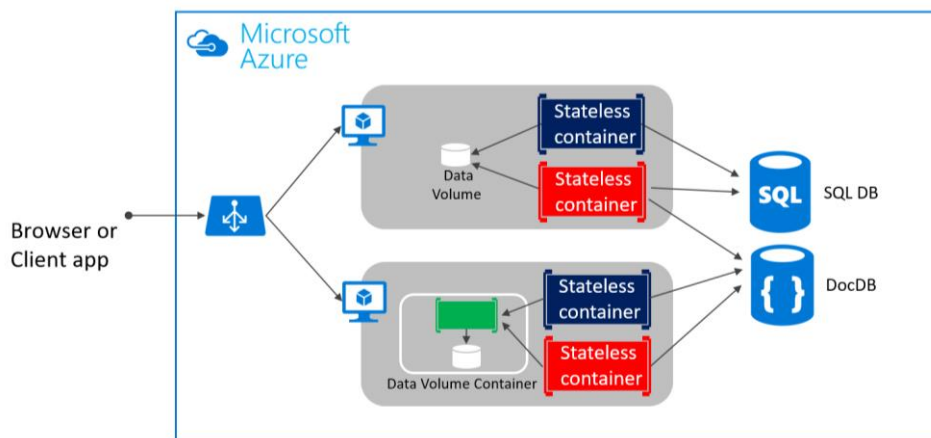
Figure 5-5. Data Volumes and external data sources for containers apps/containers

Due to the inability to manage data shared between containers that run on separate physical hosts, it is not recommended to use volumes for business data unless the Docker host is a fixed host/VM. 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 moved across multiple hosts.

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

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

Microsoft Platform and Tools for Docker

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 challenges if you are deploying based on single instances. This is where a Docker clustering software or orchestrator will help you out, as explained in later sections describe 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

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

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

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

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

Why microservices? In short, they provide 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 blocks that you must scale out at once, you can instead scale out specific microservices. That way, just the specific functional area that needs more processing power or network bandwidth to support demand can be scaled, rather than scaling out other areas of the application that really don't need it.

Architecting fine-grained microservice applications enables continuous integration and continuous development practices, and accelerates delivery of new functions into the application. Fine-grain

decomposition of applications also lets you run and test microservices in isolation, and to evolve microservices independently while maintaining rigorous contracts among them. As long as you don't break the contracts or interfaces, you can change any microservice implementation under the hood and add new functionality without breaking the other microservices that depend on it.

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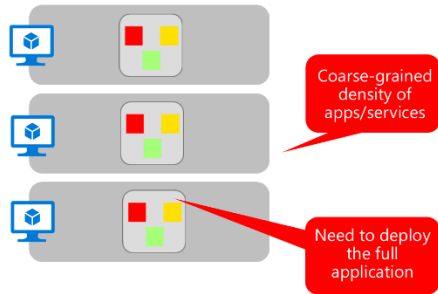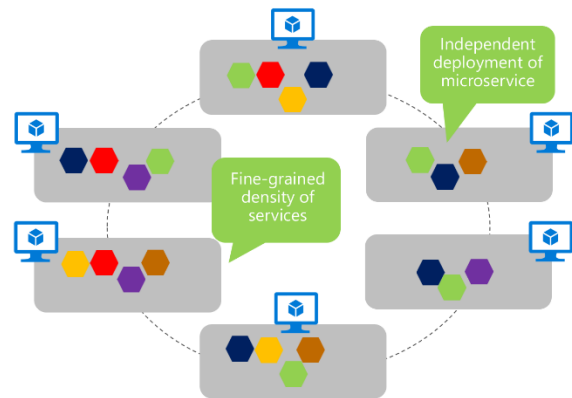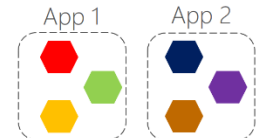
Figure X-X. Microservices approach compared to monolithic deployment approach

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

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

References – 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

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

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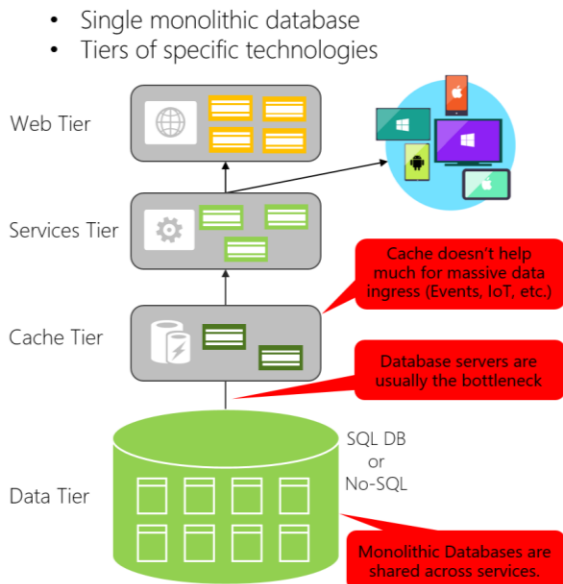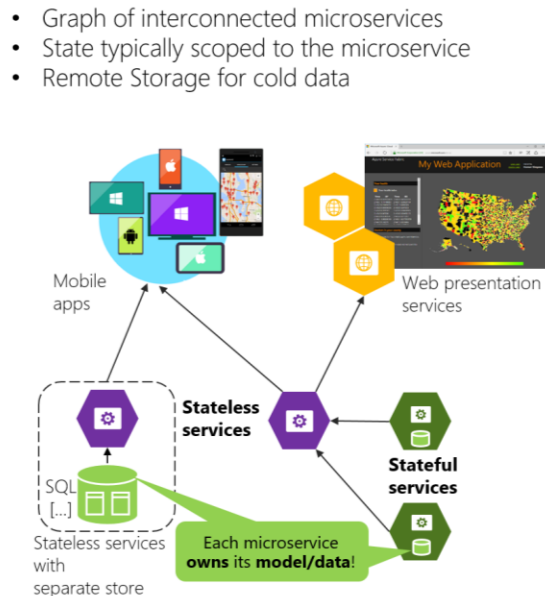
Figure X-X. Data Sovereignty Comparison: Microservices vs. Monolithic DB

The centralized database approach looks initially simpler and seems to enable re-use of entities in different subsystems to make everything consistent. But the reality is you end up with huge tables that serve many different subsystems and include attributes and columns that simply are not needed in most cases. It's like trying to use the same physical map for hiking a short trail, taking a day-long car trip, or learning geography.

A monolithic application with typically a single relational database has two important benefits. First, ACID transactions and second SQL language, but both across all the tables and data related to your app. 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 require coordinated updates to all the services and that would eliminate the microservice lifecycle autonomy.

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, and scalability. However, it does introduce some distributed data management challenges that will be explained in a later section.

Relationship between the Microservice pattern and the Bounded-Context pattern

The concept of microservice derives from the Bounded-Context pattern (BC) in Domain-Driven Design (DDD). DDD deals with large models by dividing them into multiple Bounded-Contexts and being explicit about their boundaries where each Bounded-Context must have its own model or database, in a similar way that a 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. 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 distributed protocols like HTTP or AMQP in order to access to the microservice. 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, but ultimately, both patterns are very much related.

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

References – Data Sovereignty per Microservice and Bounded-Context pattern
"Database per microservice" pattern: http://microservices.io/patterns/data/database-per-service.html   **Bounded-Context pattern:** http://martinfowler.com/bliki/BoundedContext.html   **The PolyglotPersistence approach:** http://martinfowler.com/bliki/PolyglotPersistence.html

Identifying domain-model boundaries per microservice

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

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

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

When dealing with a large application, its domain model will tend to fragment: a domain expert from the Catalog domain will think differently about 'inventory' than a logistics domain expert, for example. Or the user entity might be different in size and needed attributes when dealing with a CRM expert who wants to store every detail about the customer than 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 system.

Identifying a Domain-Model per Microservice/Bounded-Context
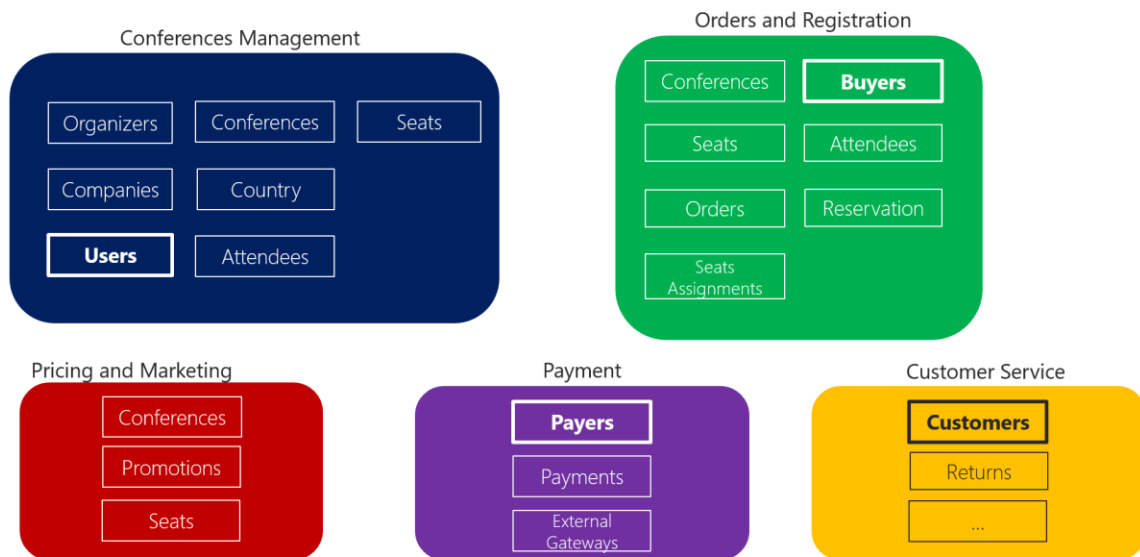
Figure X-X. Identifying Entities and Microservice's Model Boundaries

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 "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)
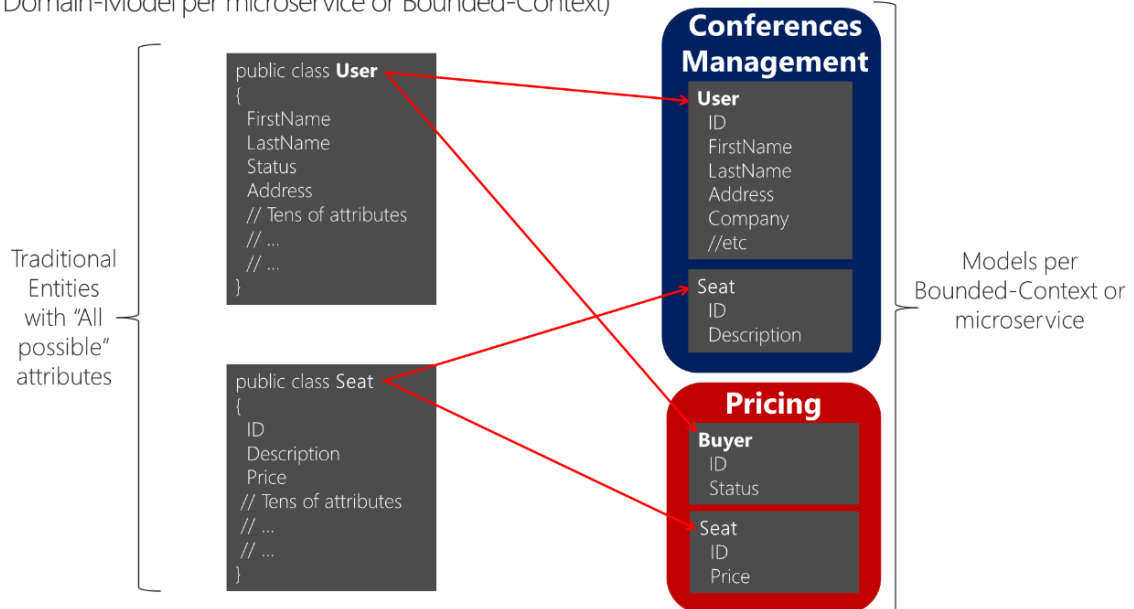
Figure <mark>X-X</mark>. Decomposing traditional data models into multiple domain-models

You can see how the "User" is present in the "Conferences Management" microservice's model, but it is also present in the form of a "Buyer", with alternate attributes, in the "Pricing" microservice's model because each microservice or Bounded-Context might not need all the data related to a "User" but just part of it, depending on the problem to solve or the context. For instance, for the pricing you don't need the "Address" or the "Passport number" of the user but just his "ID" 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".

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 (like "Two Phase Commit" transactions that you can do with SQL Server and the DTC) to check the available stock, create the order and update the available stock in the Products table.

In contrast, in a microservices architecture the Order and Product tables are private to their respective services, as shown in image <mark>X-X</mark>.
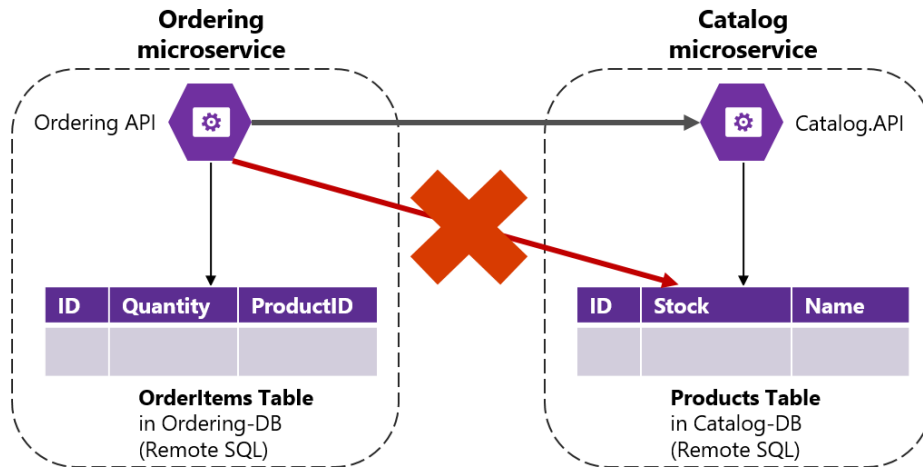
Cannot make this "red-arrow" direct update,
in a single ACID transaction,
in the microservices' world

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

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

As stated by the CAP theorem, you need to choose between availability and ACID-style consistency, and availability is usually the better choice for large and scalable systems like the ones that microservice-based architectures target. Moreover, ACID-style or Two-phase commit transactions are not just against microservices principles, but most NO-SQL databases (like Azure 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 to propagate changes across multiple microservices when certain attributes are redundant?".

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

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

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

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

B. **CQRS "Query-Table":** This solution is also known as the Materialized view pattern that pre-joins data owned by multiple microservices. For complex data aggregation from multiple tables and databases, comparable to a very complex join that you could do with a complex

SQL sentence involving multiple tables, that could be addressed with a CQRS approach by creating a de-normalized "Query-Table" in a different database used just for queries. That table will be designed per 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 requires these types of approaches and solutions.

C. **"Cold-Data" in central databases**: For complex reports and queries, a common approach is to export your "hot data" (transactional data from the microservices) into large databases only used for reporting. That central database can be a relational database like in SQL Server, Data Warehouse based like Azure SQL Data Warehouse or even based on Big Data solutions like Hadoop. Keep in mind that this centralized database would be used only for queries, but not for the original updates and transactions, as *your source of truth has to be in your microservices data*. The way you would synchronize data would be either by using Event-Driven Communication (covered in the next sections) or by using other database infrastructure import/export tools. If using Event-Driven communication, that integration would be like the way you propagate data to the mentioned CQRS "Query Database".

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

References – Distributed Data
The CAP Theorem: https://en.wikipedia.org/wiki/CAP_theorem **Eventual Consistency:** https://en.wikipedia.org/wiki/Eventual_consistency **Data Consistency Primer:** https://msdn.microsoft.com/en-us/library/dn589800.aspx **CQRS (Command and Query Responsibility Segregation):** http://martinfowler.com/bliki/CQRS.html **Materialized View pattern:** https://msdn.microsoft.com/en-us/library/dn589782.aspx **ACID vs. BASE:** http://www.dataversity.net/acid-vs-base-the-shifting-ph-of-database-transaction-processing/ **Compensating Transaction pattern:** https://msdn.microsoft.com/en-us/library/dn589804.aspx

Microsoft Platform and Tools for Docker

Including the UI per microservice: Composite apps based on microservices

TBD Theory Discussion – This won't be a "selected approach" in our sample implementations

Stateless vs Stateful Microservices and advanced frameworks

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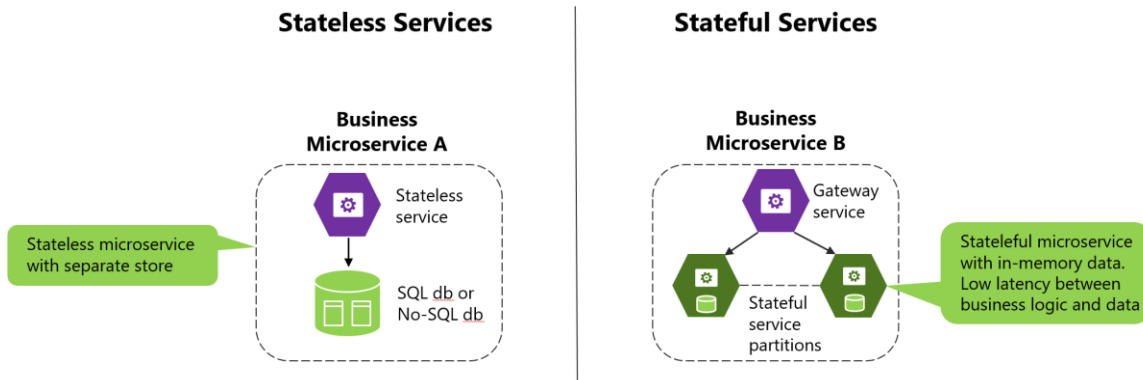
Figure X-XX. Stateless vs. Stateful services

Stateless is a perfectly valid approach and easier to implement than stateful microservices, as it is similar to traditional and well-known patterns. But stateless microservices impose latency between the process and data sources, while also presenting more moving pieces when 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 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.

Microsoft Platform and Tools for Docker

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

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

Direct Client-to-Microservice communication

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

Direct Client-To-Microservice communication
Architecture
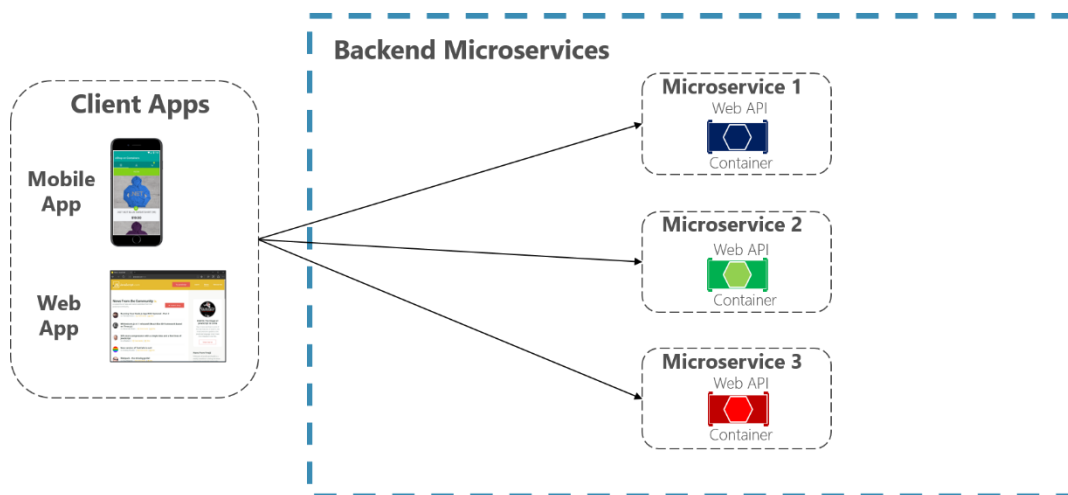
Using the Direct Client-To-Microservice communication architecture

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

This *"Direct Client-To-Microservice communication architecture"* 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.

- *How to allow clients to communicate with services that use non-Internet-friendly protocols? -* Protocols used on the server side (like AMQP or binary protocols) are not always well supported in clients, so requests will need to be translated.

- *How can you handle cross-cutting concerns such as authorization, load balancing, data transformations, and dynamic request dispatching*? – Implementing security and cross-cutting concerns on every microservice can be costly. A possible approach would be to have those services within the Docker host restricting access from the outside and implementing those cross-cutting concerns like security and authorization in a centralized place.

- *How to shape a façade especially made for mobile apps? -* API's are normally not designed around the needs of specific mobile platforms, so responses will need to be efficiently transformed, aggregated and compressed.

API Gateway

When designing and building large/complex microservice based applications, an 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 application's backend system. It is similar to the Facade pattern from object-oriented design, but in this case in a distributed system. The figure X-XX shows how an API Gateway can fit into a microservice-based architecture:
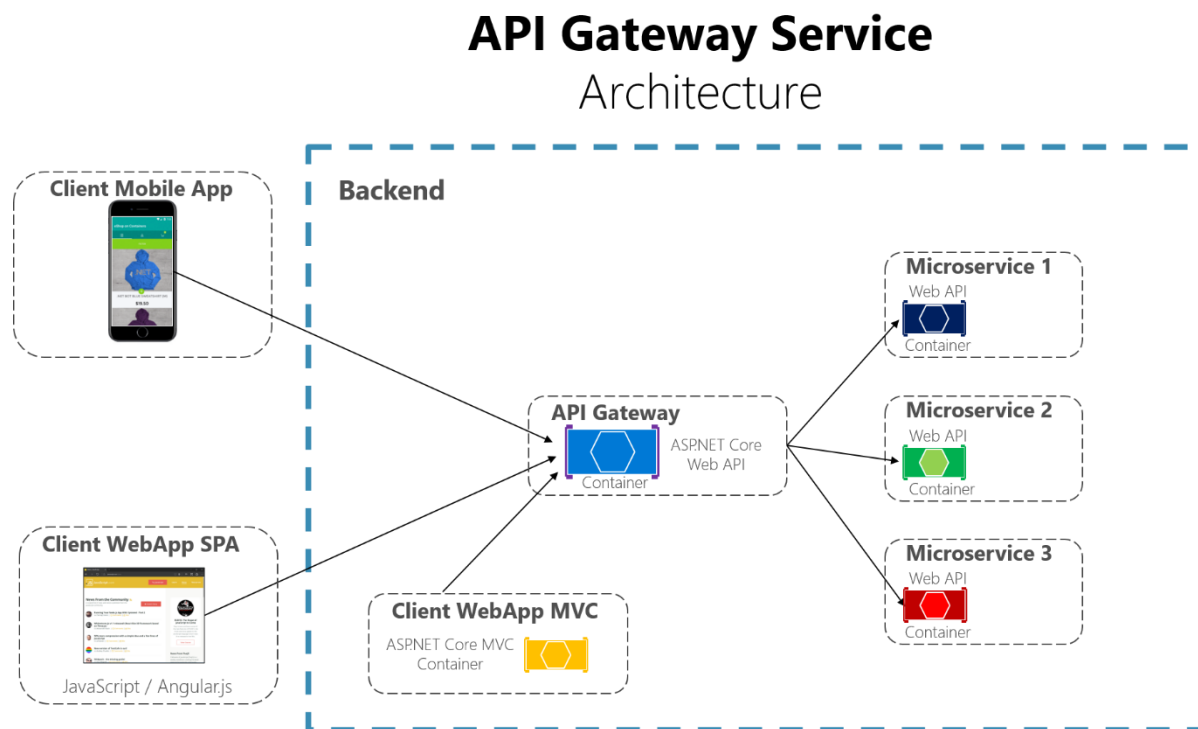
Figure X-XX. Using the API Gateway pattern in a microservice based architecture

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

Microsoft Platform and Tools for Docker

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 identify trends that might impact your business. Plus, you can have log request and response data for further online and offline analysis.

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

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

References – API Gateway and API Management
API Gateway pattern http://microservices.io/patterns/apigateway.html
Azure API Management https://azure.microsoft.com/en-us/services/api-management/

Communication between microservices

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

Communication Types

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

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

- *Synchronous* – The client waits for a response from the service. 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.
    - o An example of this communication is the "Command pattern".
- *One-to-many* – Each request is processed by multiple services or receivers. This type of communication needs to be asynchronous, as a single client synchronous execution typically can't get responses from multiple services.
    - o 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 <u>event-driven architecture</u> 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 microservices internally (within your Docker host or microservice cluster) you might also want to use binary format communication mechanisms, depending on the development platform you are using. Alternatively, you can use asynchronous, message-based communication mechanisms such as AMQP.

Additionally, there are also a variety of different message formats. Services can use human readable, text-based formats such as JSON or XML. Alternatively, you can use a binary format (which can be more efficient). 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 only for internal communication between your microservices, like when communicating between microservices bwithin your Docker host or microservice cluster (Docker orchestrators or Azure Service Fabric).

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

When using a request/response communication, a client sends a request to a service, then the service processes the request and sends back a response.

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

Request/Response Communication for Live Queries and Updates
HTTP and REST based Services
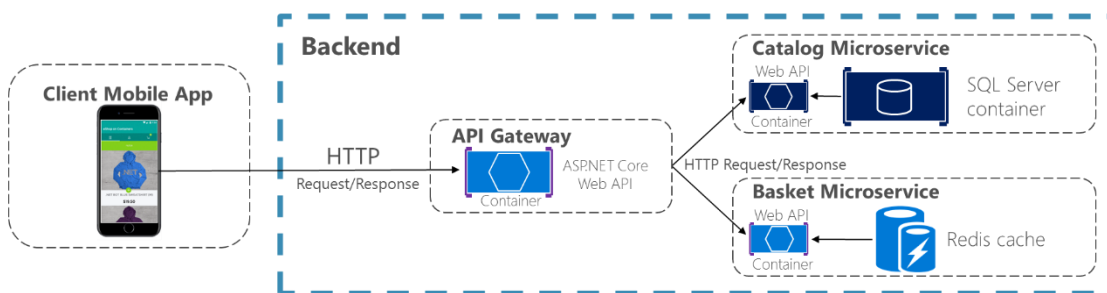
Figure <mark>X-XX</mark>. Using HTTP Request/Response communication (Sync or Async)

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 <u>ASP.NET Web API synchronously</u>. 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 <u>async/await</u>

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 Data-Driven 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 the communication is asynchronous, the client does not block waiting for a reply. Since it is a message-based communication, the client is assuming that the reply will not be received immediately.

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

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

One-to-One Asynchronous message communication

A point-to-point communication delivers a message to exactly one of the consumers that is reading from the channel, so it will be processed just once. A publish-subscribe channel delivers each message to all 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.

Asynchronous and Message based communication
One-to-One Async. communication
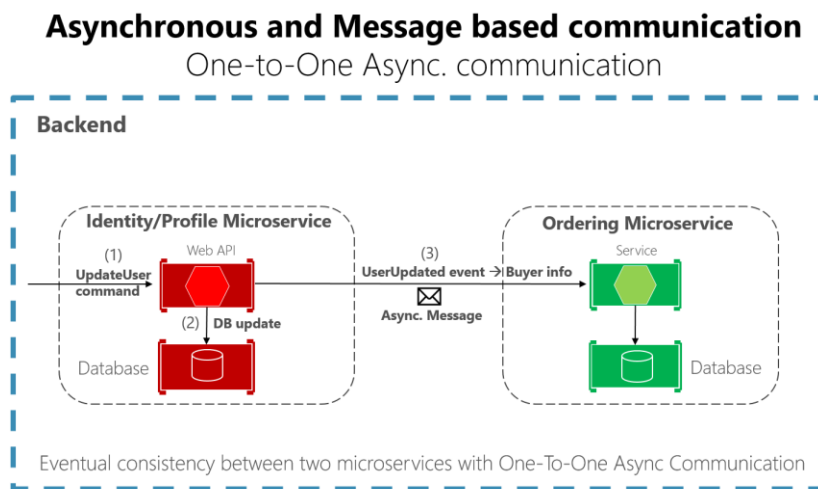
Figure X-XX. One-to-One async message communication

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

One-to-Many Asynchronous message communication

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

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 real-time and one-to-many communication with protocols such as [WebSockets](#) and higher level frameworks such as [ASP.NET SignalR](#).

Asynchronous Real-Time communication

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

Asynchronous Real Time communication
One-to-many communication
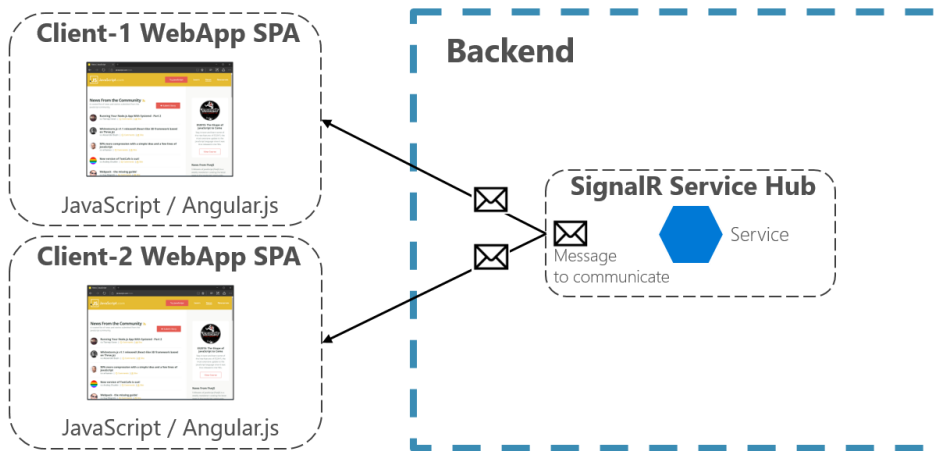
Figure <mark>X-XX</mark>. Asynchronous Real-Time communication

Asynchronous Event-Driven communication

When using this type of communication and architectural approach, a microservice publishes an event when something notable happens, such as when it updates a business entity. Other microservices subscribe to those events. When a microservice receives an event, it can update its own business entities, which might lead to more events being published.

As introduced in the previous section "Challenges and solutions for Distributed Data Management", 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.

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 <mark>X-XX</mark>. 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
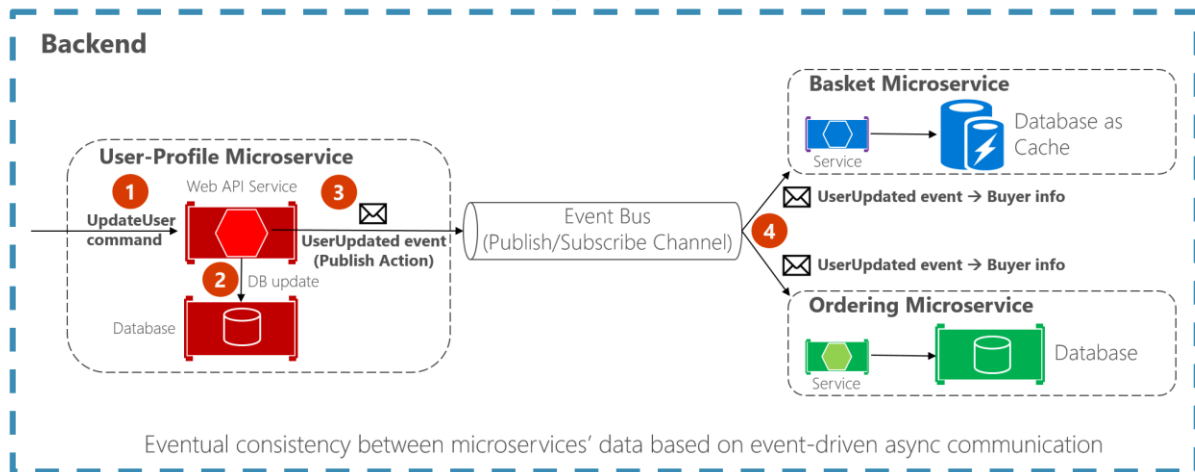
Figure <mark>X-XX</mark>. Event-Driven and async message communication

In regards to the protocol communication 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 could also be a choice, although less reliable. What you will probably need is some kind of abstraction level (like an Event-Bus interface) with its related implementation in classes with code using the API from a Service Bus like Azure Service Bus with Topics or RabbitMQ, so you can articulate the mentioned Publish/Subscribe system.

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

1. Using a transactional database table as a message queue that will be the base for an event-creator component that would create the event and publish it.
2. Using transaction log mining.
3. Using the event sourcing pattern.

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

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

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, doesn't need to recover. It also needs to then decide whether it can continue to move forward to the newer version or instead roll back to a previous version to maintain a consistent state. Questions such as whether enough machines are available to keep moving forward and how to recover previous versions of the microservice need to be considered. This requires the microservice to emit health information to be able to make these decisions.

Health Reports and Diagnostics in Microservices

It may seem obvious, and it is often overlooked, but a microservice must report its health and diagnostics. Otherwise, there is little insight from an operations perspective. Correlating diagnostic events across a set of independent services and dealing with machine clock skews to make sense of the event order is challenging. In the same way that you interact with a microservice over agreed-upon protocols and data formats, there emerges a need for standardization in how to log health and diagnostic events that ultimately end up in an event store for querying and viewing. In a microservices approach, it is key that different teams agree on a single logging format. There needs to be a consistent approach to viewing diagnostic events in the application.

Health is different from diagnostics. Health is about the microservice reporting its current state to take appropriate actions. A good example is working with upgrade and deployment mechanisms to maintain availability. Although a service may be currently unhealthy due to a process crash or machine reboot, the service might still be operational. The last thing you need is to make this worse by

performing an upgrade. The best approach is to do an investigation first or allow time for the microservice to recover. Health events from a microservice help us make informed decisions and, in effect, help create self-healing services.

When creating a microservice-based application you need to deal with complexity. Of course, a single microservice is simple to deal with, but tens or hundreds of types and thousands of instances of microservices is a complex problem to solve. 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.
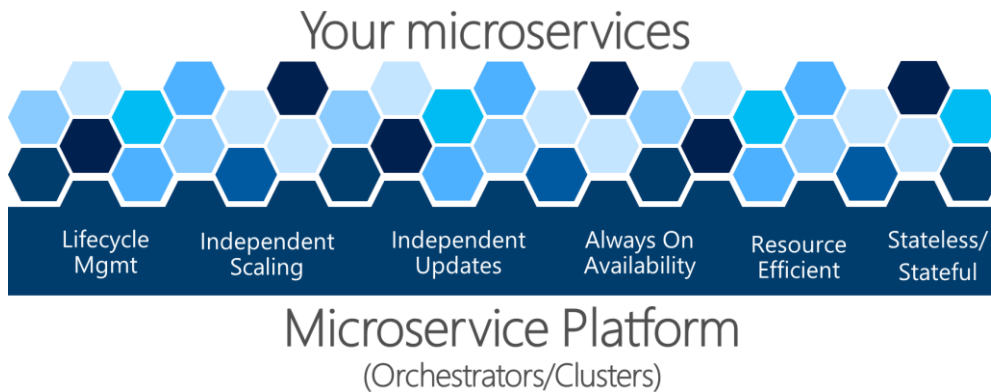
Figure X-XX. A Microservice Platform is fundamental for Microservice based applications

Those mentioned complex problems shown in figure XX-X are very hard to solve by yourself. However, development teams should focus on solving business problems and building custom applications with microservices approaches but not solving those complex infrastructure problems or the cost of any microservice-based application would be huge. 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.

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.

An architecture for composed and microservices approaches using containers would be like the diagram in Figure X-X.
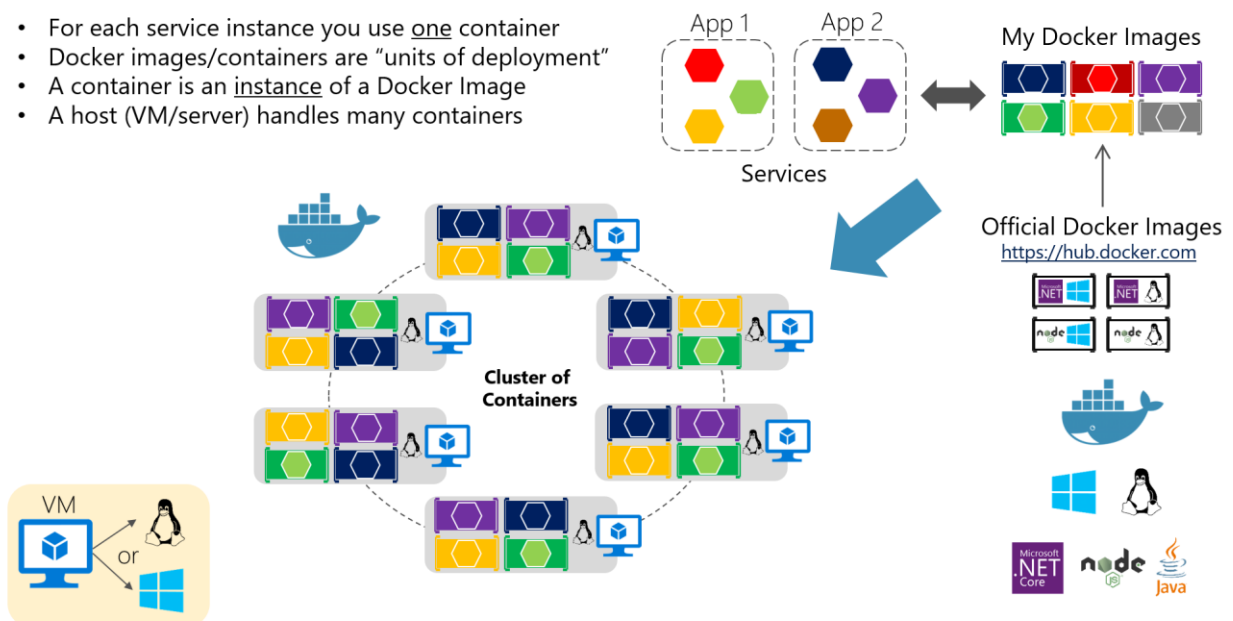
Figure X-X. Cluster of containers

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 container orchestrator Azure Service Fabric.

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

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

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

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

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

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

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

Google Kubernetes	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.

Microsoft Platform and Tools for Docker

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.
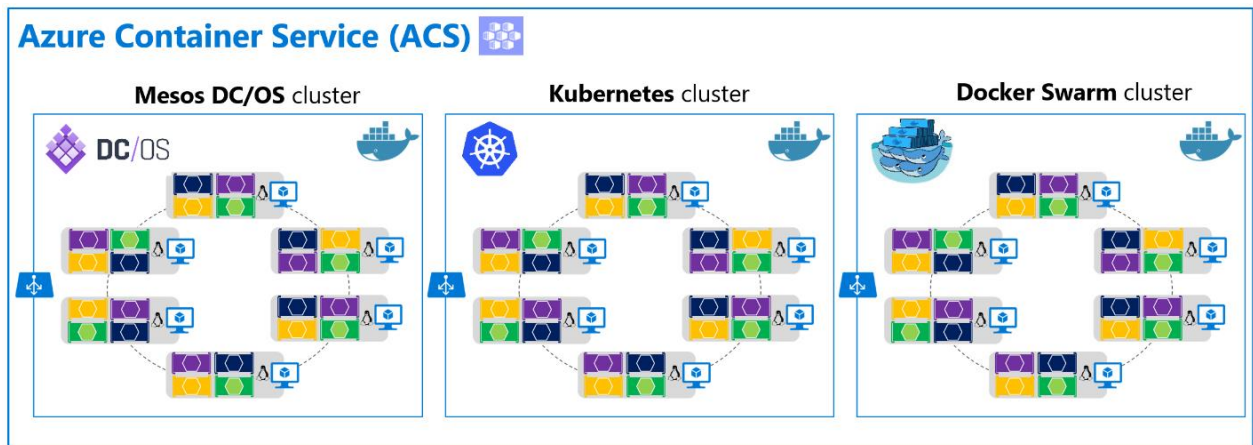
Figure 5-8. Clustering choices in ACS

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.

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.
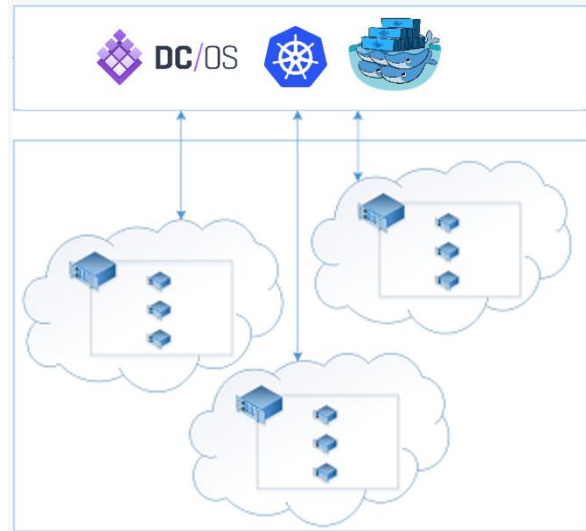
Figure 5-9. Orchestrators in ACS

Getting started with Azure Container Service

To begin using Azure Container Service, you deploy an Azure Container Service cluster 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/

Microsoft Platform and Tools for Docker

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, 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 built-in high availability.

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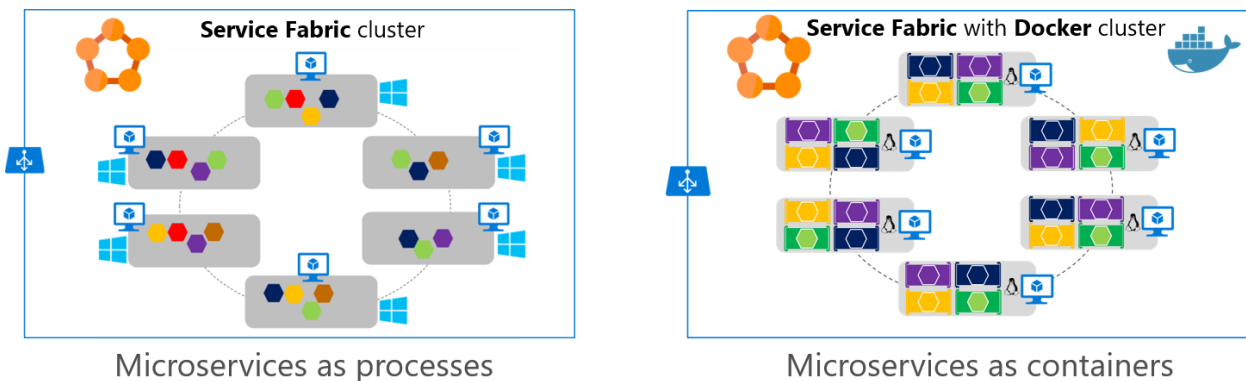
Figure X-X. Deploying microservices as processes or as containers in Azure Service Fabric

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

Development process for Docker based applications

Vision

Develop containerized .NET applications the way you like, 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
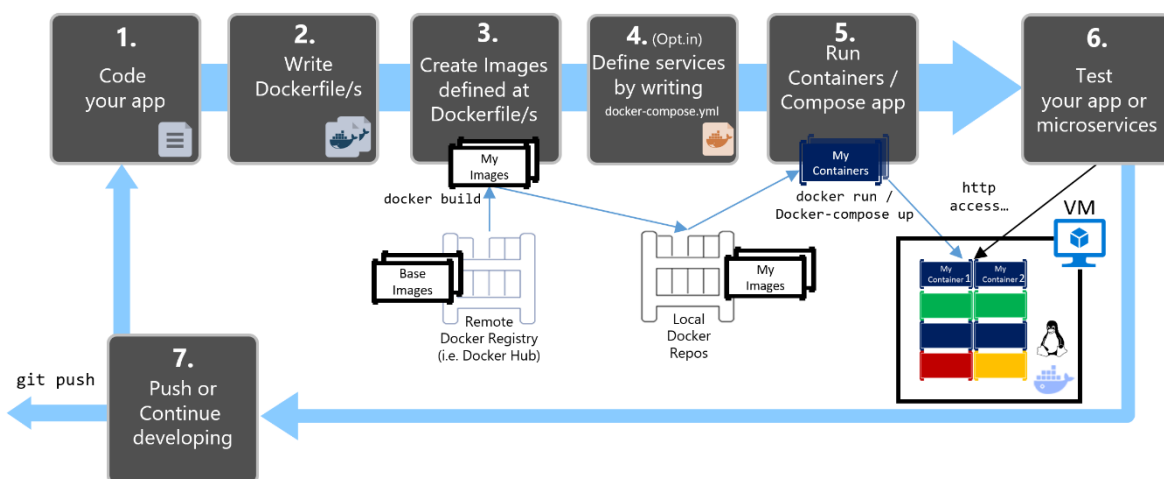
Figure <mark>X-XX</mark>. Step-by-step workflow developing Docker containerized apps

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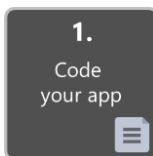
Step 1. Start coding and create your initial app/service baseline

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 <mark>x-x</mark>.
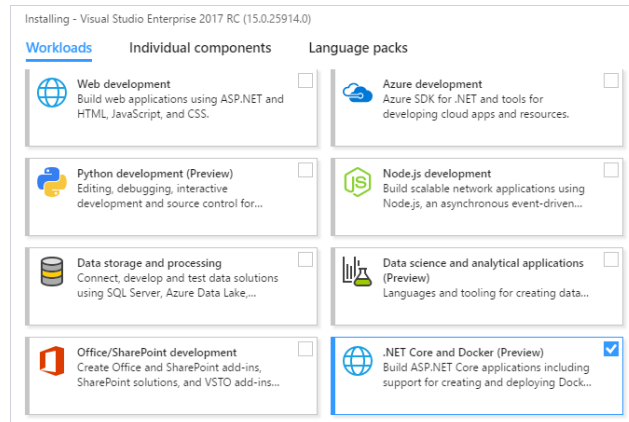
Figure X-X. Selecting the Docker and .NET Core workload

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

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

Visual Studio Tools for Docker:

http://aka.ms/vstoolsfordocker

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

Working with .NET and Visual Studio

You can start coding your app in .NET (usually 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 usually placed within the root folder of your app/service and contains the required commands so Docker knows how to setup up and run your app/service. You can 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.
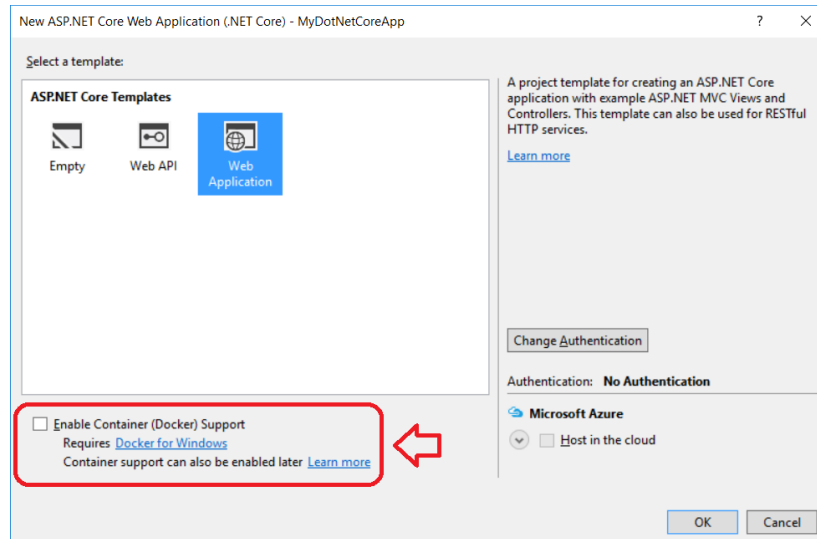
Figure X-X. Enabling Docker Support when creating a new project

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 you app is a multi-container application, as shown in figure X-X.
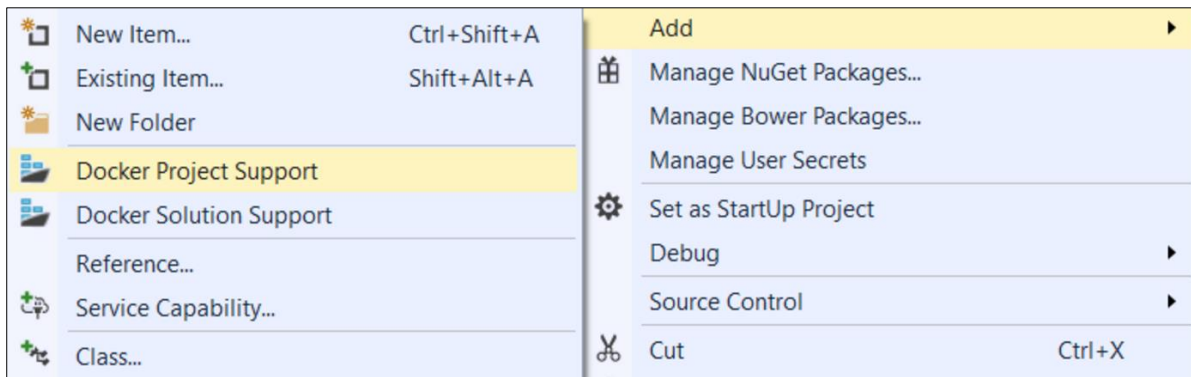
Figure X-XX. Enabling Docker support in a Visual Studio 2017 project

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

Microsoft Platform and Tools for Docker

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:

```
Dockerfile   ✛ ✕
    1 ⊟FROM microsoft/aspnetcore:1.1.0
    2  ARG source
    3  WORKDIR /app
    4  EXPOSE 80
    5  COPY ${source:-bin/Release/PublishOutput} .
    6  ENTRYPOINT ["dotnet", "MySingleContainerApp.dll"]
```
Figure X-XX. *Sample Dockerfile for a .NET Core container*

In this case, it is using the version 1.1.0 of the official ASP.NET Core Docker image for Linux named "`microsoft/aspnetcore:1.1.0`". For further details, 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.

3.
Create Images
defined at
Dockerfile/s
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.

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. You can also use the `docker-compose up --build` command for applications composed of multiple containers and services.

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 Image*

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.

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 using*

Creating Docker Images with Visual Studio

When you are using Visual Studio and a project with Docker support, you don't explicitly create an image, it will be created for you 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=l
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*

Microsoft Platform and Tools for Docker

In this case, the docker-compose.yml file 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:

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

We will re-visit 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 is not just adding a dockerfile file to your project, it is also adding a service section in your solutions docker-compose.yml file (or creating the file if it didn't exist). It is an easy way to start composing your multiple-container solution, and you can then open the docker-compose.yml file and update it with additional features.
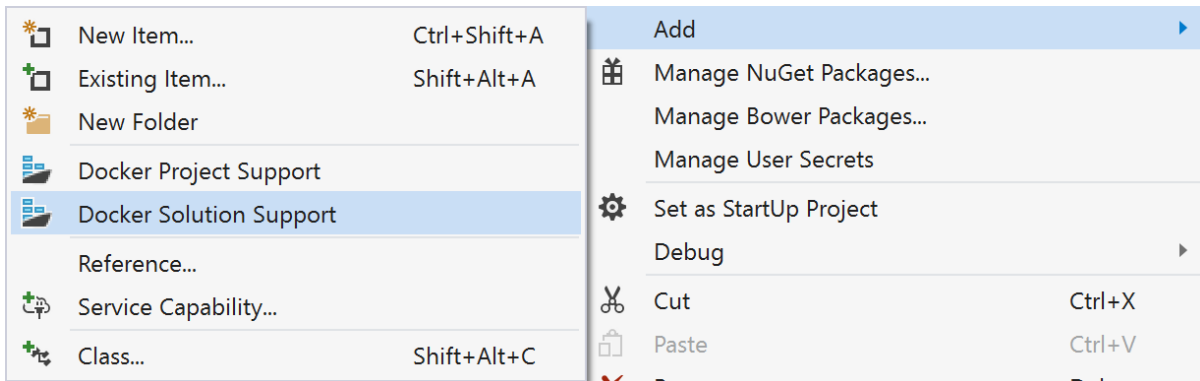
Figure X-XX. Enabling Docker Solution support in a Visual Studio 2017

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, like the docker-compose.yml example shown previously.

 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
```

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

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

Note that for this 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.

Running a single container with Visual Studio

When using Visual Studio 2015 (with Docker tools installed) or Visual Studio 2017, it is even simpler. You just need to press F5 or select the **Docker Play** button on the tool bar. Under the covers, Visual Studio will create the Docker image, deploy and run it in your Docker host.
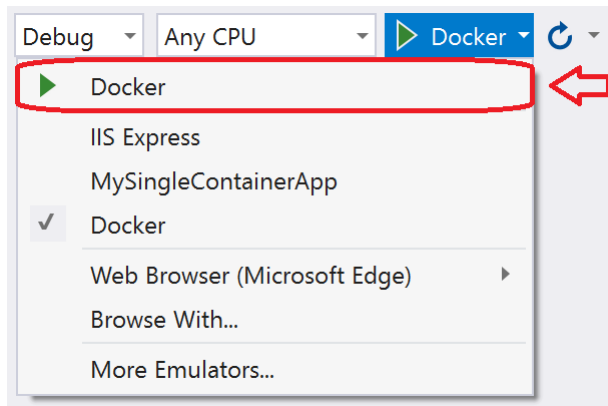
Figure X-XX. Running a Docker container using Visual Studio

Option B. Running a multi-container application

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

Figure X-XX. VM with Docker containers

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.

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

Figure 5-21. Example results when running the "docker-compose up" command

After running `docker-compose up`, the application and its related containers deployed into your Docker Host, as illustrated in the VM representation in Figure 5-20.

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 5-26, in Visual Studio 2017 there is an additional **Docker: Debug Solution** command. 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
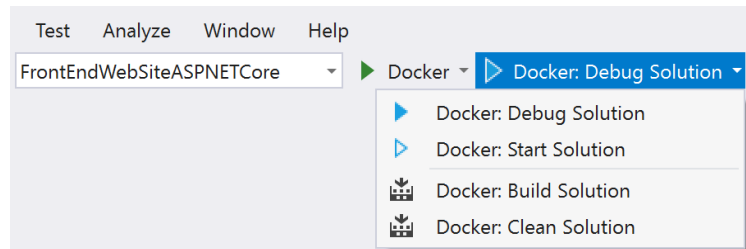
Figure X-XX. Running multi-container apps in Visual Studio 2017

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.

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 CD VM)

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

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

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

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

Open a browser on the Docker host and navigate to that site, and you should see your app/service running.
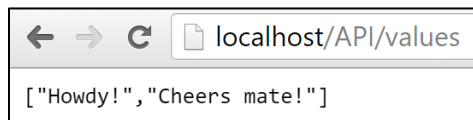
Figure 5-22. Example of testing your Docker application locally using localhost

Note that it is using the port 80 but internally it was being redirected to the port 5000, because that's how it was deployed with 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/

Microsoft Platform and Tools for Docker

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.

VS development workflow for Docker apps
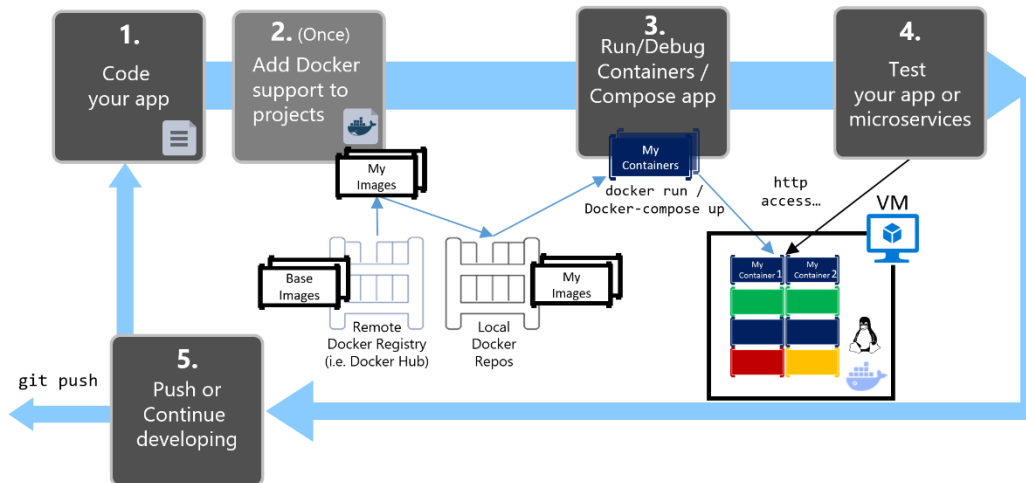
Figure X-XX. Simplified workflow when developing with Visual Studio

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 applications for Linux or Windows containers

Vision

tbd

(Short section – "Easy to get started with Docker" choice – Regular Docker containers on Linux or Windows Server Nano – Simple case with for example a monolithic ASP.NET Core MVC application)

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

TBD

tbd

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 NO-SQL).
- Application integration logic - messaging layer, possible service buses, etc.

The application will have requisites of high scalability, but probably, certain sub-systems will 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 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.
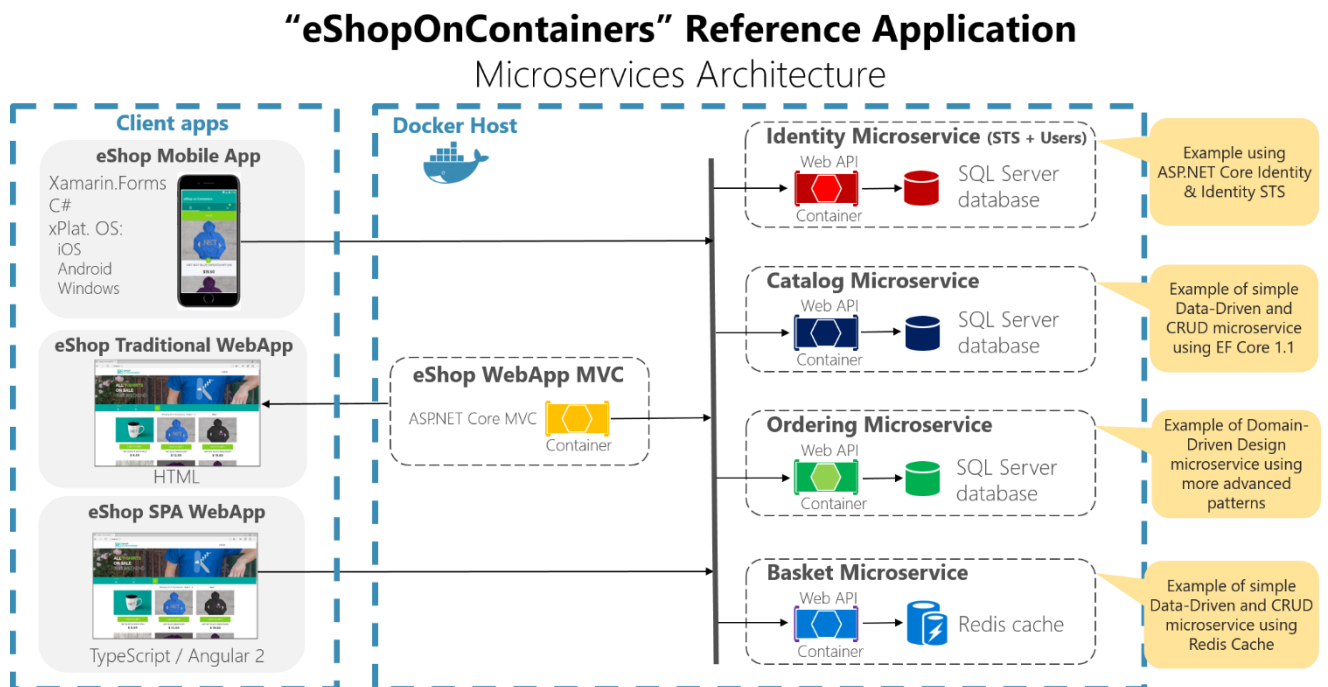
Figure X-XX. eShopOnContainers reference app – Using Direct Client-to-Microservice Communication

Hosting environment: In the architecture diagram shown you see several containers deployed within a single Docker Host. That would be the case when deploying to a single Docker Host with 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 – Initially using Direct Client-to-Microservice Communication.

The application will be deployed as a set of microservices in the form of containers, and client apps can communicate with those containers, as well as communicate between microservices/containers. 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.

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, and speeds up deployments.
  - 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.

    o 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 not using a microservice oriented infrastructure (like an orchestrator or scheduler). This 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.

External Architecture per application

Internal Architecture per microservice

Backend Microservices

Client Apps
- Mobile App
- Web App

Microservice 1
Web API
Container

Microservice 2
Web API
Container

Microservice 3
Web API
Container

- Direct Client-to-Microservice
- API Gateway
- Inter service communication
- Pub/Sub and Event-Driven

- Simple Data-Driven/CRUD design versus Advanced design patterns, DDD, etc.
- Single or multiple libraries
- Dependency Injection, IoC and S.O.L.I.D

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 overengineered, and in a similar way, complex microservices with a lot of ever-changing business logic shouldn't be implemented as CRUD components or you will end up with low quality spaghetti code.

Creating a simple data-driven/CRUD microservice

Designing a simple data-driven/CRUD microservice

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

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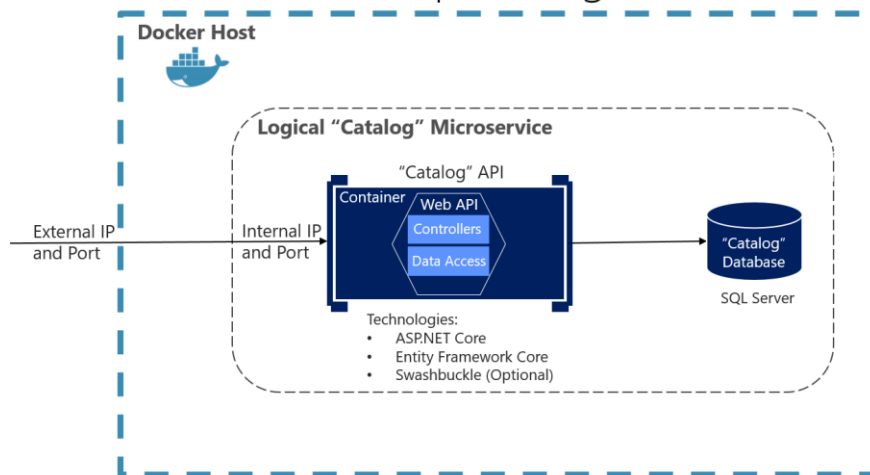
Figure X-XX. Simple data-driven/CRUD microservice design diagram

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 HA and HS. For example, you might choose DocumentDB when using a NO-SQL 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.
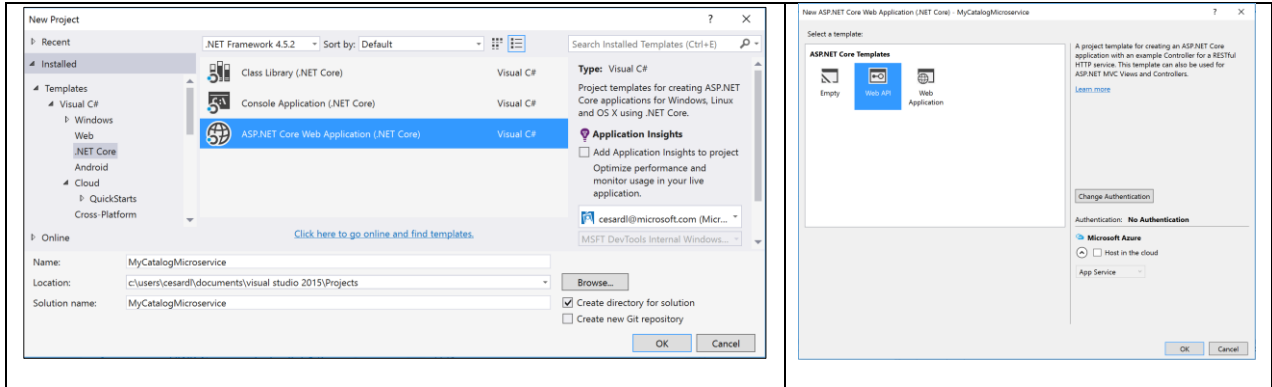
Figure X-XX. Creating an ASP.NET Core Web API project in VS 2015

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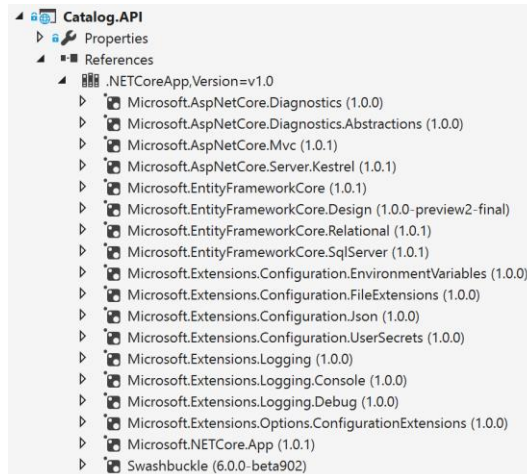
Figure X-XX. Dependencies in a simple CRUD Web API microservice

Implementing CRUD Web API services with Entity Framework Core

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

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 Class) entity class.

Figure X-XX. Sample POCO Entity class: CatalogItem

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.

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.

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

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

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

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

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

        return Ok(model);
    }
}
```

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

Saving data

Data is created, deleted, and modified in the database using instances of your entity classes. See [Saving Data](#) to learn more. You can add code 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(blog);
_context.SaveChanges();
```

Figure <mark>X-XX</mark>. Saving Data

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.
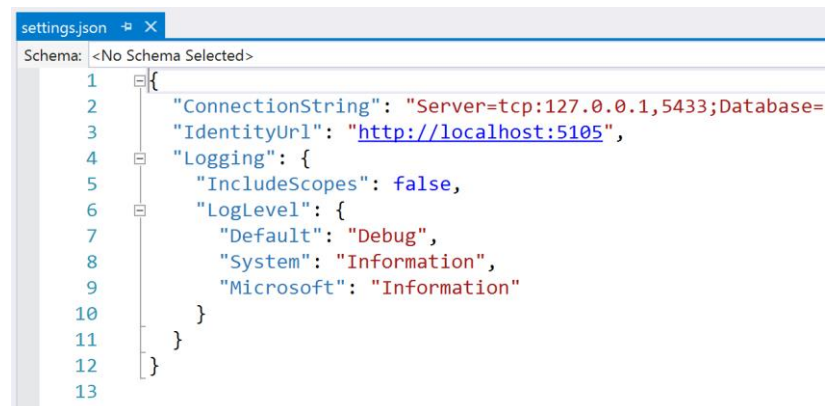
```
0 references | Carlos Cañizares Estévez, 5 days ago | 4 authors, 6 changes
public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IConfiguration>(Configuration);

    services.AddDbContext<CatalogContext>(c =>
    {
        c.UseSqlServer(Configuration["ConnectionString"]);
        c.ConfigureWarnings(wb =>
        {
            //By default, in this application, we don't want to have client evaluations
            wb.Log(RelationalEventId.QueryClientEvaluationWarning);
        });
    });
}
```

Figure <mark>X-XX</mark>. 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.

```
settings.json  ⊞ ×
Schema: <No Schema Selected>
    1  ⊟{
    2        "ConnectionString": "Server=tcp:127.0.0.1,5433;Database=
    3        "IdentityUrl": "http://localhost:5105",
    4  ⊟    "Logging": {
    5          "IncludeScopes": false,
    6  ⊟      "LogLevel": {
    7            "Default": "Debug",
    8            "System": "Information",
    9            "Microsoft": "Information"
   10          }
   11        }
   12  ⌐}
   13
```

Figure <mark>X-XX</mark>. Docker and environment variables for connection strings

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.

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

    ports:
      - "5101:5101"
```

The docker-compose.yml files at the solution level are not 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.

Application Configuration in ASP.NET Core services

TBD

Working with multiple environments

TBD

References – Securing .NET Applications
Configuration in ASP.NET Core https://docs.microsoft.com/en-us/aspnet/core/fundamentals/configuration **Working with multiple environments: Dev, Production, Staging.** https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments

RESTful web API Design and Implementation

<mark>TBD</mark>

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

<mark>TBD</mark>

References – API Versioning
TBD 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 APIs

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 **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 to 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 <mark>X-XX</mark> below:
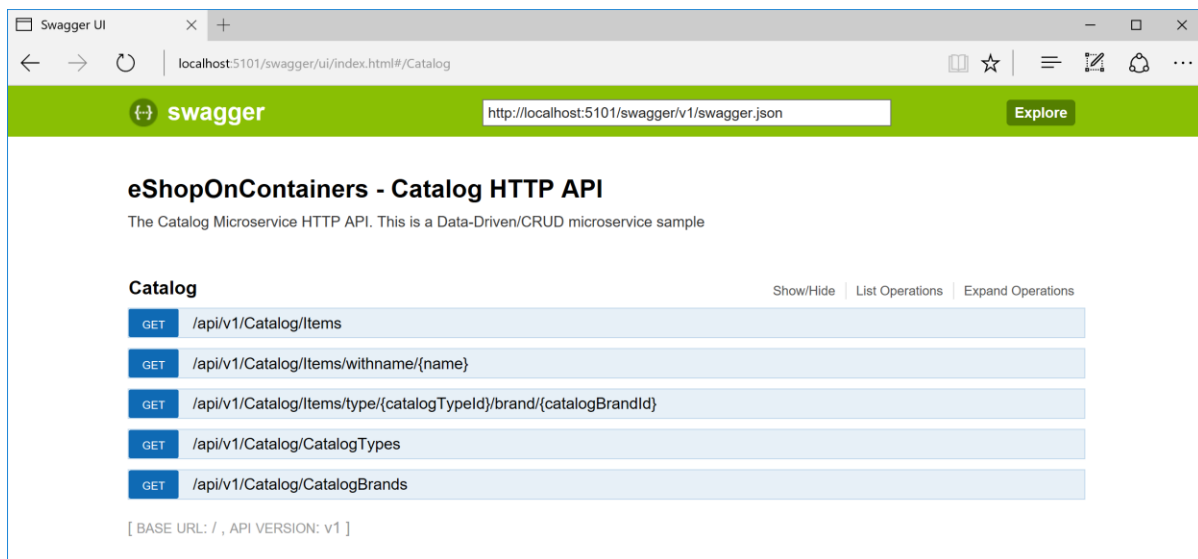
Figure <mark>X-XX</mark>. Swashbuckle UI based on Swagger metadata – eShop Catalog microservice example

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 be from Swagger-based tools, including client proxy classes code generators that can target many platforms. For example, using swagger-codegen, which allows code generation of API client libraries, server stubs and documentation automatically.

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

Once you have installed those Nuget packages 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.
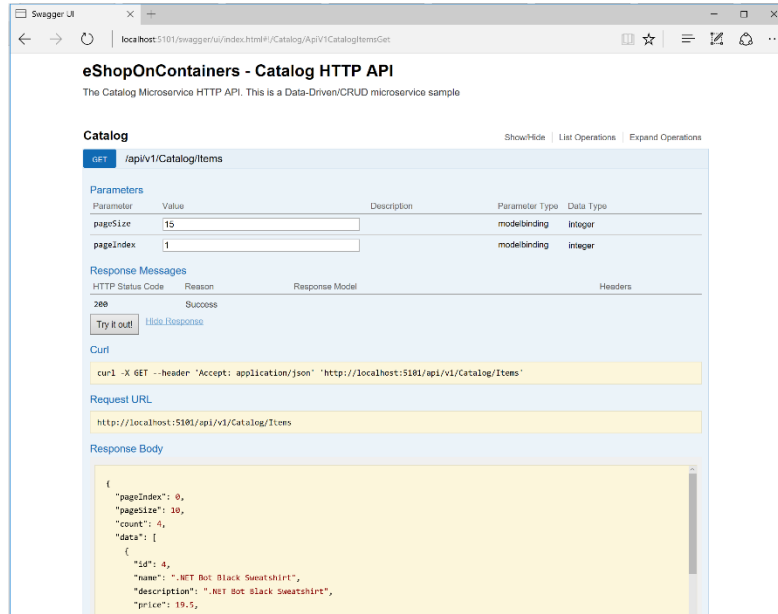
Figure X-XX. Swashbuckle UI testing the Catalog/Items API method

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.
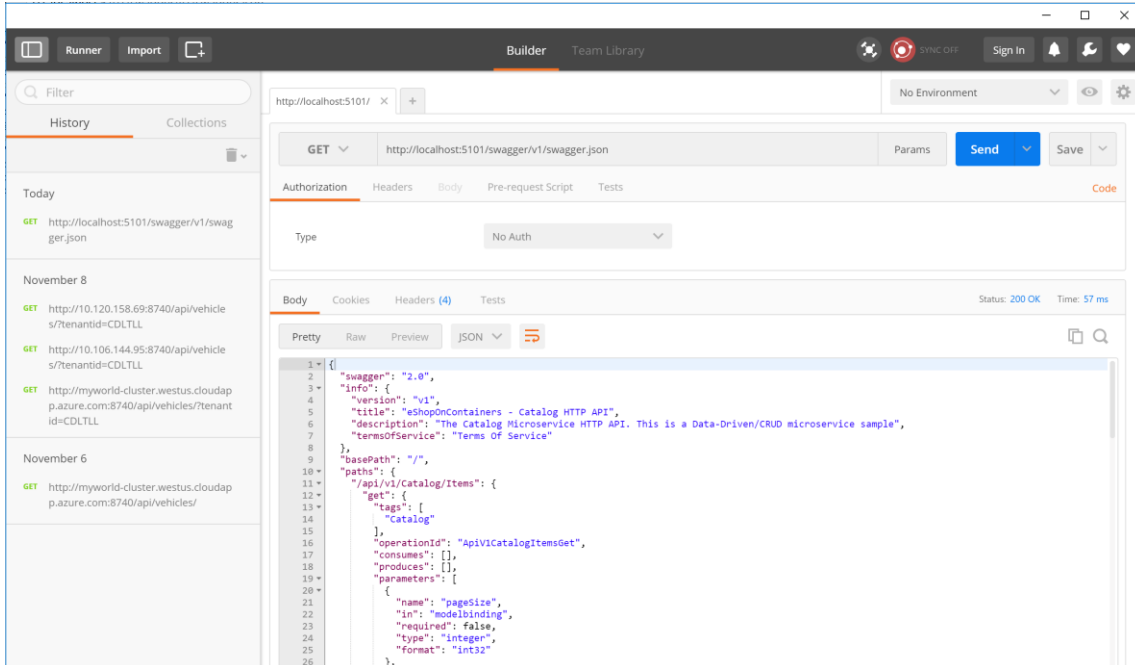
Figure X-XX. Swagger JSON metadata

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

Architecting and developing Docker applications

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

References – Swagger and Swashbuckle
ASP.NET Web API Help Pages using Swagger https://docs.microsoft.com/en-us/aspnet/core/tutorials/web-api-help-pages-using-swagger

Creating microservices based on Domain-Driven Design (DDD) and Command and 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 earler 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.

DDD vs. DDD patterns

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 Eric Evan's book (Domain-Driven Design), 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

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 logical two 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
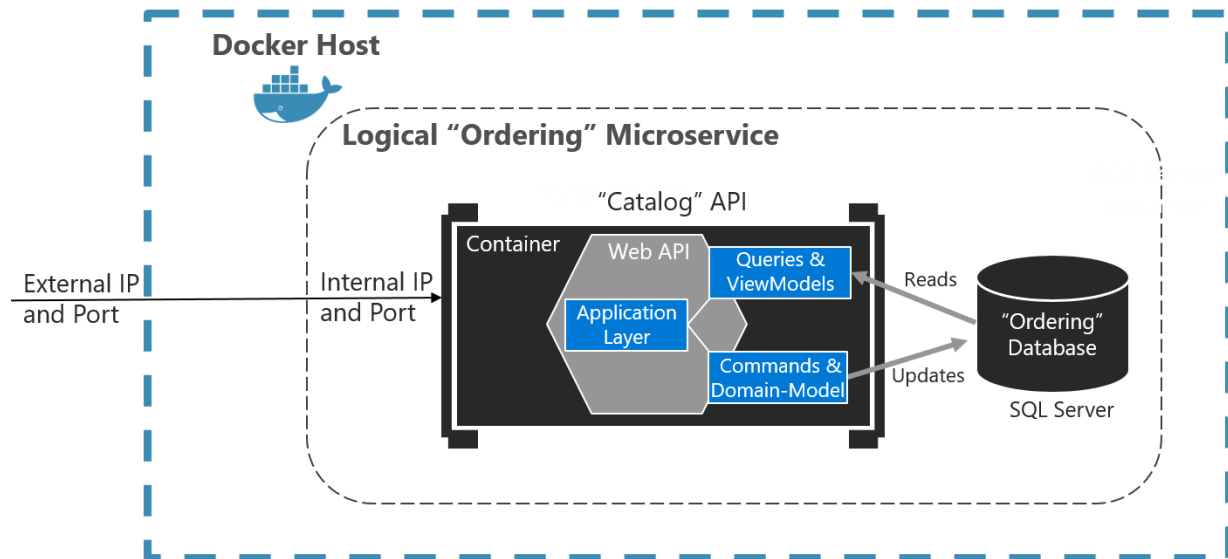
Figure X-XX. Simplified CQRS and DDD based microservice design

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, is about methods within the same object being either queries or commands - returning state or mutating state but not both. Even a single Repository pattern object could be compliance with CQS according with that definition. CQS could be It can be considered as a principle.

[CQRS (Command and Query Responsibility Segregation)](#) was introduced by Greg Young and also promoted by Udi Dahan and other advocates. It is based on the CQS principle, although it is more 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 your it in the long term with quality, but honestly, they usually just matter for the transactional/updates area which can be triggered by

commands. If that is the case, why should you limit yourself and use the same constraints, limitations and even unnecessary complexity when still using those patterns for the queries if that can turn to worse performance and lack of flexibility in your queries?

For example, when using Aggregates for your model plus Entity Framework Core for your infrastructure, if you 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 convolute operations that you shouldn't need to do for a query. Not taking into account that when using Entity Framework Core, you might not get the best performance possible for your queries for many reasons, compared to plain SQL data access as when using a Micro ORM.

This is why, as shown in image X-XX, 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.

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 bad. It will be a large failure if you try to use CQRS and DDD patterns everywhere because many subsystems, bounded-contexts or microservices are simpler and can be implemented in an easier way as simple CRUD services or any other approach depending on what you need to create.

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

References – 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

http://cqrs.nu/Faq/command-query-responsibility-segregation

Event-Sourcing (ES)

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

Implementing the Reads/Queries in a CQRS microservice

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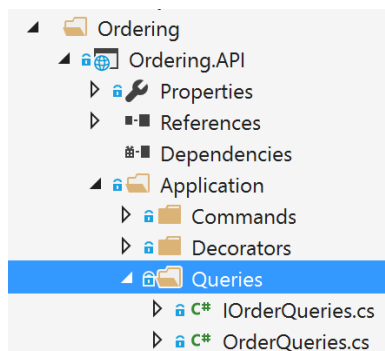
Figure X-XX. Queries in the Ordering microservice from eShopOnContainers

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 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 eShopOnContainers 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 eShopOnContainers 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.

Dapper by Sam Saffron, Marc Gravell, Nick Craver   ✅ v1.50.2
A high performance Micro-ORM
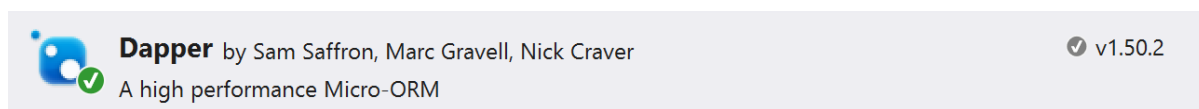
You will also need to add a `using` statement so your code has access to Dapper's extension methods.

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.

```csharp
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 steps/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.
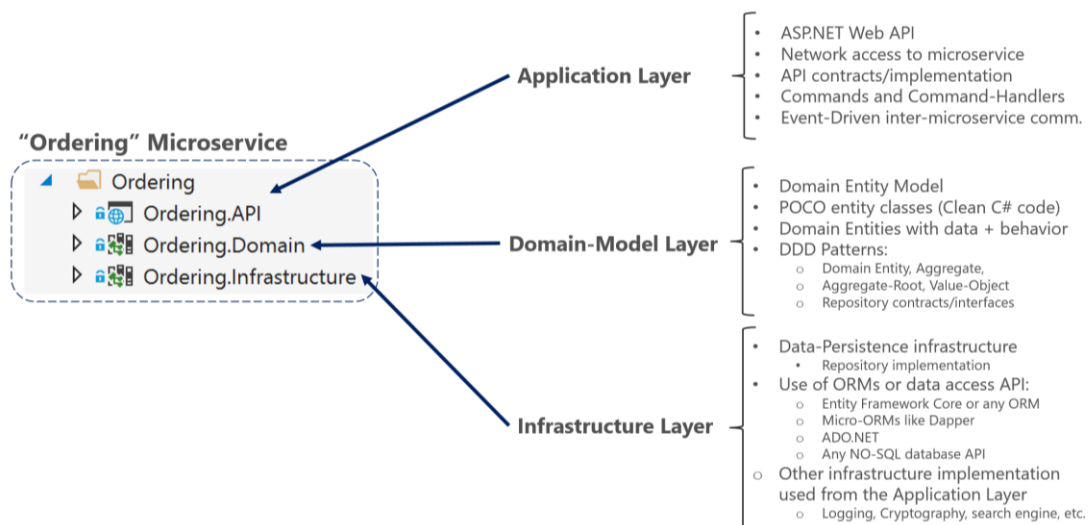
Figure X-XX. DDD Layers in the Ordering microservice from eShopOnContainers

Architecting and developing Docker applications

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 <mark>x-xx</mark> below about the Ordering.Domain layer library which only has dependencies with the .NET Core libraries.
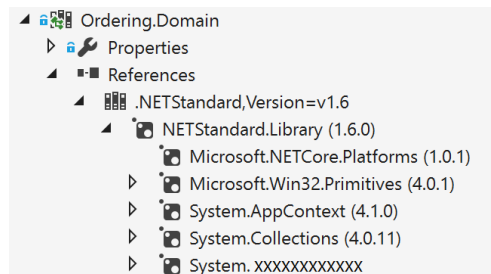
Figure <mark>X-XX</mark>. Layers implemented as libraries allow a better control of dependencies

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

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

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

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

Luckily, 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 NO-SQL 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 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 CQS approach, commands accepted by the microservice, and even the event-driven communication between microservices. However, the ASP.NET Web API must not contain business rules or domain knowledge (especially domain rules in regards to transactions or updates), which should be owned by the Domain Model class library.

The Application Layer (in this case an ASP.NET Web API project) must only coordinate tasks and must not hold 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
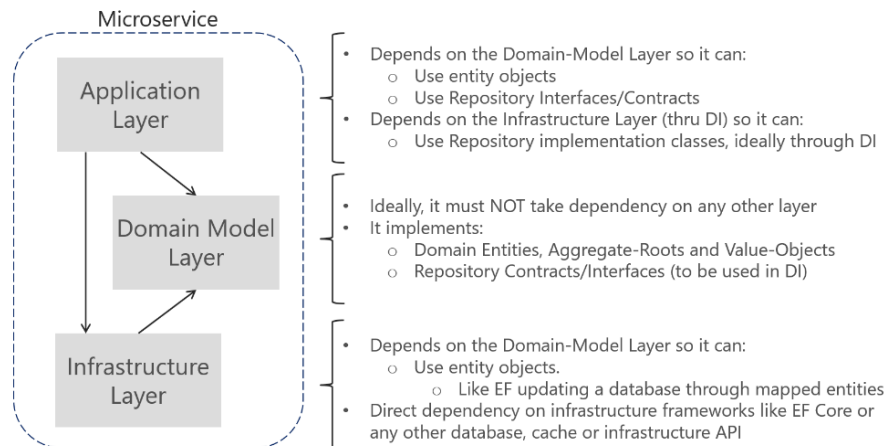
Figure X-XX. Dependencies between Layers in DDD

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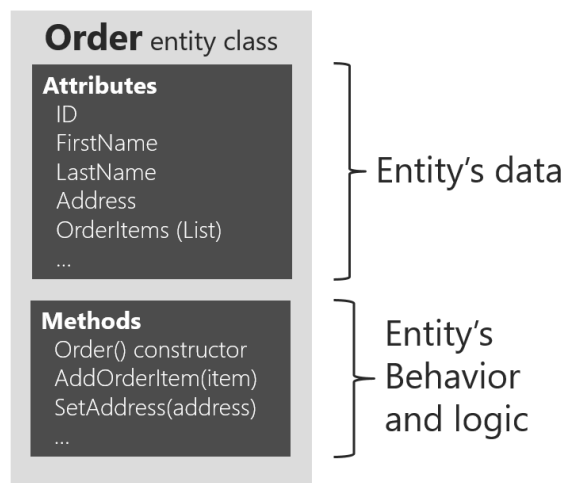
Figure X-XX. Example of Domain Entity Design implementing data plus behavior

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

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 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.

Aggregate pattern
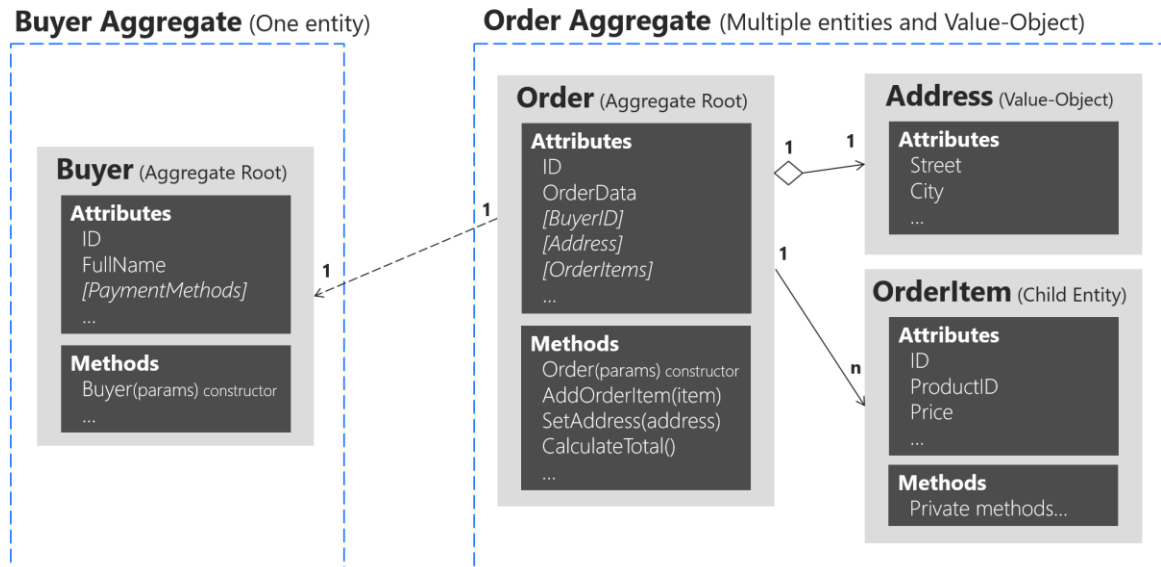
Figure X-XX. Aggregate pattern examples

Note that the Buyer aggregate could have additional child entities depending on your Domain, as it has in the sample Ordering microservice from *eShopOnContainers* reference application. The figure X-XX is just a case supposing that it could have a single entity, as an example of aggregate holding only an aggregate-root.

Identifying and working with aggregates requires research and experience. Below 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

Implementing a microservice's Domain Model with .NET Core and Entity Framework 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.
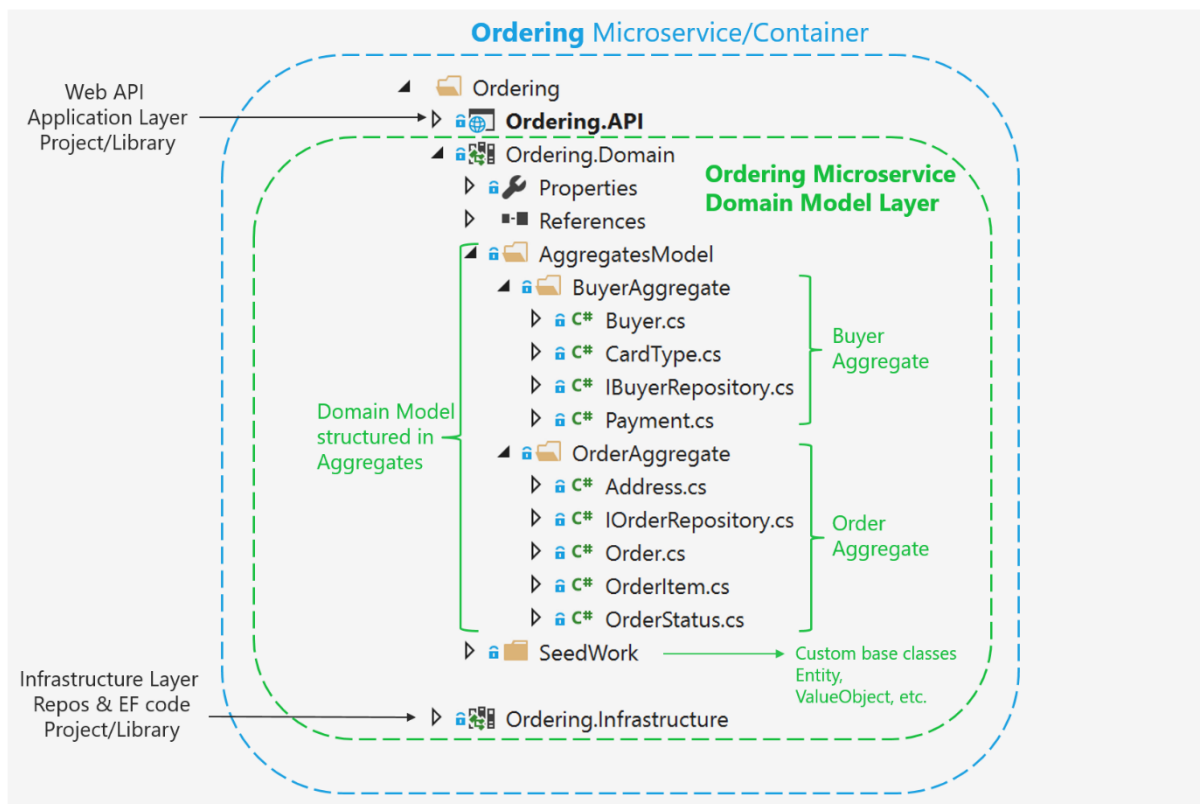
Figure X-XX. Domain Model structure for the Ordering microservice

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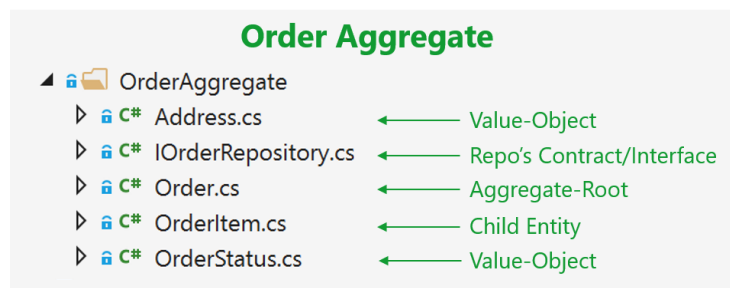
Figure X-XX. The "Order" aggregate in the VS solution

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 custom, so it is your code, not infrastructure code from any ORM like EF.

```
Entity Framework Core 1.0

    public class Order : Entity, IAggregateRoot //Entity is a custom base class with the Id
    {
        public int BuyerId { get; private set; }
        public DateTime OrderDate { get; private set; }
        public int StatusId { get; private set; }
        public ICollection<OrderItem> OrderItems { get; private set; }
        public Address ShippingAddress { get; private set; }
        public int PaymentId { get; private set; }

        protected Order() { } //Needed only by EF Core 1.0

        public Order(int buyerId, int paymentId)
        {
```

```
            BuyerId = buyerId;
            PaymentId = paymentId;
            StatusId = OrderStatus.InProcess.Id;
            OrderDate = DateTime.UtcNow;
            OrderItems = new List<OrderItem>();
        }
        public void AddOrderItem(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 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 class:

Wrong according to DDD

//My code in CommandHandlers or Web API controllers

//... Code with validations and business logic ...

OrderItem myNewOrderItem = new OrderItem(orderId, productId, productName, pictureUrl, unitPrice, discount, units);

myOrder.OrderItems.Add(myNewOrderItem);

//...

In this 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.

For that approach, you would have needed to mark the `OrderItems` collection with a public setter in its property definition. That is forbidden in DDD. Entities must not have public setters in any entity's property.

As you can see in the code implementing the Order Aggregate-Root, all setters should be private, so any operation against the entity's data or its child entities will need to be performed through methods in the Aggregate-Root class. 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

//My code in CommandHandlers or WebAPI controllers, only related to application stuff

// NO code here related to OrderItem validations/logic

myOrder.AddOrderItem(productId, productName, pictureUrl, unitPrice, discount, units**);**

// The code related to OrderItem params validations or domain rules will be within AddOrderItem()

//...

The important point here is that most of the validations 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. This is extremely useful when properties only must have a get accessor. Previously, with properties get and set accessors were required and the only choice was to make the setters private.

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

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
{
    private bool _someOrderInternalState;

    private int _buyerId;
    public int BuyerId => _buyerId;

    private DateTime _orderDate;
    public DateTime OrderDate => _orderDate;

    private int _statusId;
    public int StatusId => _statusId;

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

    private Address _shippingAddress;
    public Address ShippingAddress => _shippingAddress;

    private int _paymentId;
    public int PaymentId => _paymentId;

    protected Order() { }

    public Order(int buyerId, int paymentId)
    {
        _buyerId = buyerId;
        _paymentId = paymentId;
        _statusId = OrderStatus.InProcess.Id;
        _orderDate = DateTime.UtcNow;
        _orderItems = new List<OrderItem>();
    }
    public void AddOrderItem(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
    //...
```

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

When using EF 1.0, within the DbContext, you need to map the properties that you defined with only get accessors to the actual field in the database. 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 X-XX 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.
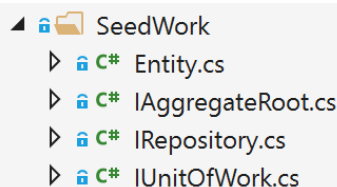
Figure X-XX. A sample Domain Model "Seedwork" with base classes and interfaces/contracts

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

The custom Entity base class

The following code is an example of an Entity base class where you can place code that can be used the same way by any Domain Entity, such as the entity Id, equality operators, etc.:

```
Entity Framework Core 1.0

    public abstract class Entity
    {
        int? _requestedHashCode;
        int _Id;

        public virtual  int Id
        {
            get
            {
                return _Id;
            }
            protected set
            {
                _Id = value;
            }
        }
        public bool IsTransient()
        {
            return this.Id == default(Int32);
        }

        public override bool Equals(object obj)
        {
            if (obj == null || !(obj is Entity))
                return false;

            if (Object.ReferenceEquals(this, obj))
                return true;

            Entity item = (Entity)obj;

            if (item.IsTransient() || this.IsTransient())
                return false;
            else
                return item.Id == this.Id;
        }
        public override int GetHashCode()
        {
            if (!IsTransient())
            {
                if (!_requestedHashCode.HasValue)
                    _requestedHashCode = this.Id.GetHashCode() ^ 31;
                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 particular implementation.

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

References – Repository Contracts
Separated Interface pattern (By Martin Fowler)
http://www.martinfowler.com/eaaCatalog/separatedInterface.html

Implementing Value Objects

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

References – Value-Objects
Implementing a Value Object with EF Core 1.1
http://tbd http://geeks.ms/unai/2017/01/29/shadow-properties-otros-ejemplos-de-uso/ (Remove/Spanish)

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 like using the state pattern or a special forms of Value-Objects (VO). A VO lets you implement and vary the logic related to the same state in the same class. This also increases cohesion and lessens class coupling.

Value Objects and State Pattern advantages are:

- Easier to extend with new states by adding a new object. (Open/Close Principle)

- Easier to assure that all signals are treated by the states, since the base class should define the signals as abstract functions.

- Easier to extend a particular states' behavior by deriving from the state. The state pattern should put a particular state's behavior in one object.

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

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

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/
DRY principle
https://en.wikipedia.org/wiki/Don't_repeat_yourself
Implementing Enumeration classes in .NET
https://lostechies.com/jimmybogard/2008/08/12/enumeration-classes/

Designing Validations in the Domain Model Layer

From the DDD perspective, validation rules can be viewed as invariants. One of the central responsibilities of an aggregate is enforcement of invariants across state changes for all the entities within that aggregate.

Domain Entities should be "always-valid entities". There is a certain number of invariants for an object that should always be true (as an example that an OrderItem object always has to have a quantity or a name). From that point of view, invariant enforcement is the responsibility of the domain entity itself (especially of the Aggregate-Root) and therefore an entity shouldn't even be able to exist without being valid. Invariant rules are simply expressed as contracts and exceptions or notifications are thrown when these are violated.

The reasoning behind this is that a lot of bugs comes from the fact that objects are in a state they should never have been. This is a good and practical explanation from *Greg Young*:

"Let's propose we now have a SendUserCreationEmailService that takes a UserProfile … how can we rationalize in that service that Name is not null? Do we check it again? Or more likely … you just don't bother to check and "hope for the best" you hope that someone bothered to validate it before sending it to you. Of course, using TDD one of the first tests we should be writing is that if I send a customer with a null name that it should raise an error. But once we start writing these kinds of tests over and over again we realize … 'wait if we never allowed name to become null we wouldn't have all of these tests'…".

Implementing Validations in the Domain Model Layer

Validations are usually implemented in the Domain entities constructors and within methods that can update the entity. There are multiple ways to implement those variations like sentences of control verifying data and raising exceptions if the validation cannot succeed or more advanced patterns like using the Specification pattern for validations and the Notification pattern to be able to return a collection of errors instead of returning an exception as soon as something was not validated.

Validating conditions and returning exceptions

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

```
public void SetAddress(Address address)
{
    if (address == null)
```

```
        {
            throw new ArgumentNullException(nameof(address));
        }
        ShippingAddress = address;
    }
```

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

Using Validation attributes in the model based on Data Annotations

Another approach is to use validation attributes (Data Annotations). Validation attributes are a way to configure model validation so it's similar conceptually to validation on fields in database tables. This includes constraints such as assigning data types or required fields. Other types of validation include applying patterns to data to enforce business rules, such as a credit card, phone number, or email address. Validation attributes make enforcing these requirements very easy to use.

However, this approach might be too intrusive in a Domain-Driven Design Model as it takes a dependency on ModelState.IsValid() from Microsoft.AspNetCore.Mvc.ModelState which you have to call from your MVC controllers. The model validation occurs prior to each controller action being invoked, and it is the controller method's responsibility to inspect ModelState.IsValid and react appropriately. So, use it or not depending on how coupled you'd like your model to be with that infrastructure:

```
using System.ComponentModel.DataAnnotations;
//Other usings
public class Product : Entity  //Entity is a custom base class which has the Id
{
    [Required]
    [StringLength(100)]
    public string Title { get; private set; }

    [Required]
    [Range(0, 999.99)]
    public decimal Price { get; private set; }

    [Required]
    [VintageProduct(1970)]
    [DataType(DataType.Date)]
    public DateTime ReleaseDate { get; private set; }

    [Required]
    [StringLength(1000)]
    public string Description { get; private set; }

    //Constructor…

    //Additional methods for Entity's logic and constructor…
}
```

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

Validating Entities by implementing the Specification pattern and the Notification pattern

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

It is worth to mention that you can also use only one of those patterns, like validating manually with sentences of control but using the Notification pattern to be able to stack and return a list of validation errors.

Dealing with deferred validation in the domain

There are various approaches to deal with deferred validations in the domain such as the mentioned Specification pattern and the Deferred Validation approach described by Ward Cunningham in his Checks pattern language. If you have the Implementing Domain-Driven Design book by Vaughn Vernon, you can also read from pages 208-215.

References – Validations in the Domain Model
Model Validation in ASP.NET Core https://docs.microsoft.com/en-us/aspnet/core/mvc/models/validation **Adding Validation in ASP.NET Core** https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation **Using the Notification Pattern to replace throwing exceptions with notification in validations** https://martinfowler.com/articles/replaceThrowWithNotification.html **Specification and Notification Patterns** https://www.codeproject.com/Tips/790758/Specification-and-Notification-Patterns **Validation in Domain-Driven Design (DDD)** http://gorodinski.com/blog/2012/05/19/validation-in-domain-driven-design-ddd/ **Domain Model Validation** http://colinjack.blogspot.com/2008/03/domain-model-validation.html **Validation in a DDD world** https://lostechies.com/jimmybogard/2009/02/15/validation-in-a-ddd-world/

Client side validation (Validation in the Presentation Layers)

Even when the "source of truth" is the Domain Model and ultimately you have to have validation at the Domain Model level, validation can be both at the domain model level (server side) and the client side.

Client side validation is a great convenience for users. It saves time they would otherwise spend waiting for a round trip to the server that might return validation errors. In business terms, even a few fractions of seconds multiplied hundreds of times each day adds up to be a lot of time, expense, and frustration. Straightforward and immediate validation enables users to work more efficiently and produce better quality input and output.

Also, just like the view model and the domain model are different, view model validation and domain validation might be similar but serve a different purpose. And if you're still concerned about being DRY ("Don't Repeat Yourself" principle), consider that in this case code reuse might also mean "coupling" and in enterprise applications it is more important not to couple the server side to the client side more than to follow the DRY principle everywhere.

You could also validate you commands or input DTOs in the server side, especially if your system doesn't have any client UI application, like if you are building just a public API. But if you have any

client application, from a UX perspective, it is good to be proactive and not allow the user to type in stuff that makes no sense.

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

The implementation of client side validation depends on what kind of client application you are building, as it will be different if you are validating data in a web MVC web application with most of the code in .NET, or a SPA web app with that validation being coded in JavaScript or TypeScript, or a mobile app coded with Xamarin and C#.

Below there are a few references of validations depending on each type of client apps and technologies.

References – Validation in the Client side (Presentation Layer apps)
Validation in Xamarin mobile apps
https://developer.xamarin.com/recipes/ios/standard_controls/text_field/validate_input/
https://developer.xamarin.com/samples/xamarin-forms/XAML/ValidationCallback/
Validation in ASP.NET Core apps
https://docs.microsoft.com/en-us/aspnet/core/tutorials/first-mvc-app/validation
Validation in SPA web apps (Angular 2 / TypeScript / Javascript)
https://scotch.io/tutorials/angular-2-form-validation
https://angular.io/docs/ts/latest/cookbook/form-validation.html
http://breeze.github.io/doc-js/validation.html

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

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

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

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

It is easier to duplicate validation logic than to keep it consistent across application layers and in many cases having redundant validation in the client side is good as you can be proactive.

Domain Events

Domain events are similar to messaging-style events, with one important difference. With true messaging, queuing and a service bus, a message is fired and handled asynchronously and is very useful for integrating multiple bounded-contexts, microservices or even different applications. However, with domain events, you want to raise an event within the same domain operations you are actually running at that same moment. You 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 (which is the case when using static and synchronous domain events).

Independently of the chosen implementation (static/synchronous events vs recorded events and before committing the transaction, dispatching those domain events at that point), the domain events and their "side effects" (actions triggered afterwards managed by event-handlers) should occur immediately, in-proc and as part of the same logical transaction.

Domain Events vs. Integration Events

Semantically, domain and integration events are the same thing, plain notifications about something that just happened. However, their implementation might 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 whole operation effectively never happened.

In addition, Integration events have to be based on asynchronous communication between multiple microservices (other bounded-contexts) or even external systems/applications. Thus, under the Event-Bus interface it needs some infrastructure that allows inter-process and distributed communication between potentially remote services. It can be based on a commercial service bus, queues, shared database used as a mailbox, ASP.NET SignalR Hubs (SignalR wouldn't assure the communication to happen, though, but could be okay for development/test environments), or any other distributed and ideally push based messaging system.

Implementing Domain Events

A domain event is just a simple POCO that represents an interesting occurrence in the domain. For example, the XXXX event defined in the code below.

TBD – POCO Event definition and base interface, etc.

TBD - Drill down on Domain Events IMPLEMENTATION – ...

References – Validation in the Client side (Presentation Layer apps)
A Better Domain Events Pattern
https://lostechies.com/jimmybogard/2014/05/13/a-better-domain-events-pattern/
Strengthening your domain: Domain Events
https://lostechies.com/jimmybogard/2010/04/08/strengthening-your-domain-domain-events/
Domain Events Pattern Example
http://www.tonytruong.net/domain-events-pattern-example/
Domain Events – Take 2
http://udidahan.com/2008/08/25/domain-events-take-2/
Domain Events – Salvation
http://udidahan.com/2009/06/14/domain-events-salvation/
How to create fully encapsulated Domain Models

http://udidahan.com/2008/02/29/how-to-create-fully-encapsulated-domain-models/

A Pattern for Sharing Data Across Domain-Driven Design Bounded Contexts, Part 2 (Integration Events)

https://msdn.microsoft.com/en-us/magazine/dn857357.aspx

Designing the Infrastructure-Persistence Layer

The data persistence components provide access to the data hosted within the boundaries of our microservice (i.e. your related microservice's database). Therefore, it has the actual implementation of components like "Repositories" and "Unit of Work" patterns that provide such functionality to access the data hosted within the boundaries of our microservice.

The Repository pattern

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

"Repository" is one of the well documented ways of working with a data source. Martin Fowler in his PoEAA book describes a repository as follows:

"*A repository performs the tasks of an intermediary between the domain model layers and data mapping, acting in a similar way to a set of domain objects in memory. Client objects declaratively build queries and send them to the repositories for answers. Conceptually, a repository encapsulates a set of objects stored in the database and operations that can be performed on them, providing a way that is closer to the persistence layer. Repositories, also, support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping*".

Define one Repository per Aggregate

Hence, per each aggregate (or per each Aggregate-Root as they are 1:1) you should create one Repository class that allows you to populate data in-memory coming from the database in the form of the Domain Entities and also allows you to persist updated data in those entities of the aggregate back into the database.

If you are using the CQS/CQRS architectural pattern, then most of the public methods you will have in a Repository will be just to create/update/delete the database from your Domain Model, but you won't need to have any method for queries in such a Repository.

It is important to re-emphasize that Repositories should only be defined per each Aggregate-Root. Following the goals of the aggregate-root to maintain transactional consistency between all the objects within an aggregate you should never create one Repository per each table in the database but per each aggregate-root.

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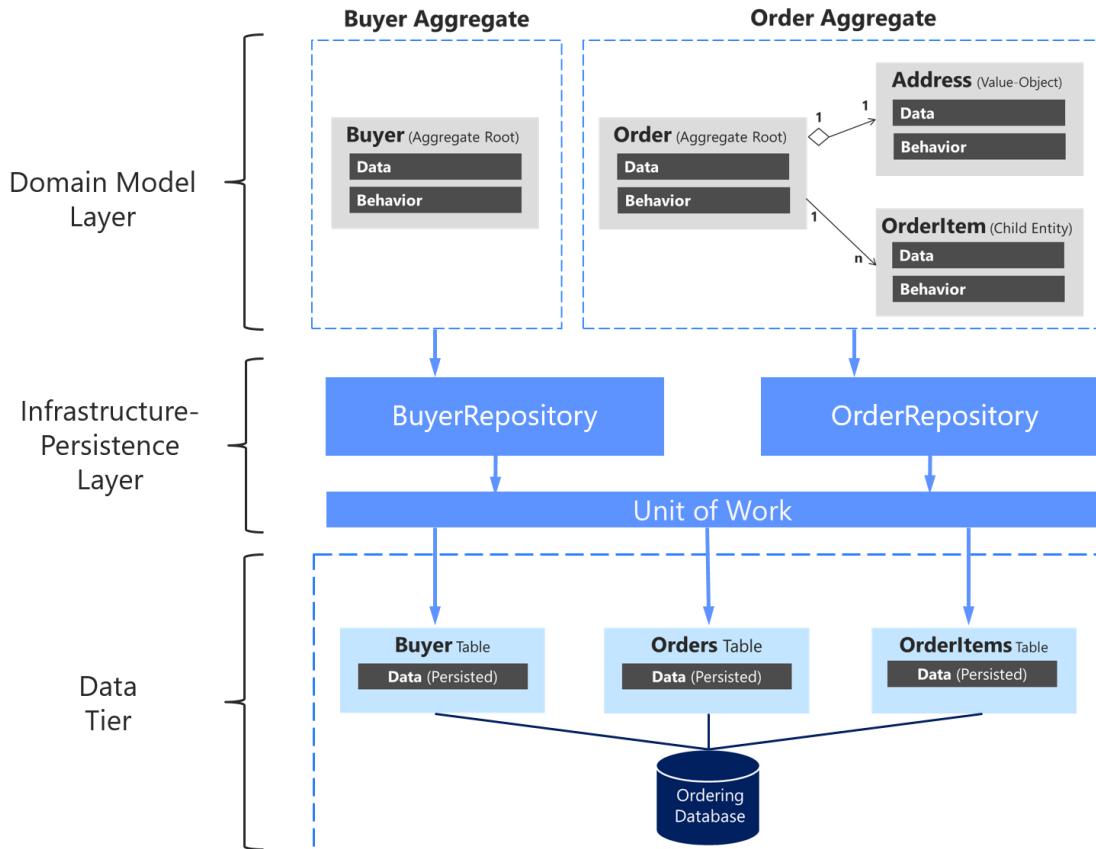
Figure X-XX. Relationship between Repositories, Aggregates and Database Tables

The Repository pattern makes it easier to test your application logic

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

As introduced in a previous section, it is recommended to define and place the Repository interfaces in the domain layer so the application layer (like your Web API microservice) doesn't directly depend on the Infrastructure layer where you have implemented the actual Repository classes. By doing so and using Dependency Injection in the controllers of your Web API you could implement mock Repositories that would return fake hard-coded data instead of accessing the database. That decoupled approach allows you to create and run unit tests that can test just the logic of your application without needing any connectivity to the database.

Connection to databases can fail and most of all, running hundreds of tests against a database is a bad thing because first, it might take a lot of time because of the large number of tests, and second, the database's records might change and impact on the results of your tests so they might not be consistent. Testing against the database is not a Unit Tests but an Integration Test. You should have many Unit Tests running fast but fewer Integration Tests against the databases.

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

It is important to differentiate between a Repository and the legacy "Data Access" object (aka. DAL). A Data Access object directly performs data access and persistence operations against the storage. However, a repository "marks" the data with the operations you want to do in the memory of a Unit of

Work object (like in EF when using the DbContext), but these updates will not be performed immediately.

A Unit of Work is referred to as a single transaction that involves multiple operations of insert/update/delete. To say it in simple words, it means that for a specific user action (say registration on a website), all the transactions like insert/update/delete are done in one single transaction, rather than doing multiple database transactions in a more chatty way.

These multiple persistence operations will be performed at a later time in a single action when your code from the Application layer commands it. That decision about "Applying changes" in memory into the real database storage is usually based on the Unit of Work pattern (In Entity Framework the Unit of Work is implemented as the DBContext).

In many cases, this pattern or way of applying operations against the storage can increase the application performance and reduce the possibility of inconsistencies. Also, it reduces transaction blocking in the database tables because all the intended operations will be committed as part of one transaction which will be more efficiently run in comparison to many isolated operations against the database. Therefore, the selected ORM will be able to optimize the execution against the database (e.g., grouping several update actions) as opposed to many small separate executions.

References – Infrastructure and Persistence patterns
The Repository pattern http://martinfowler.com/eaaCatalog/repository.html https://msdn.microsoft.com/en-us/library/ff649690.aspx http://deviq.com/repository-pattern/ Repository pattern. By Eric Evans in his DDD book. **The Unit of Work pattern** http://martinfowler.com/eaaCatalog/unitOfWork.html **Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application** https://www.asp.net/mvc/overview/older-versions/getting-started-with-ef-5-using-mvc-4/implementing-the-repository-and-unit-of-work-patterns-in-an-asp-net-mvc-application

Implementing the Infrastructure-Persistence Layer with Entity Framework Core

When using relational databases like SQL Server, Oracle, PostgreSQL, etc., a recommended approach when using .NET is to implement this persistence layer based on Entity Framework which supports LINQ and provides strong typed objects for your model and a simplified persistence into your database.

Entity Framework has a long history as part of the .NET Framework. However, when using .NET Core you should use Entity Framework Core which is cross-platform (runs on Windows or Linux in the same way than .NET core does) plus it is a completely "reset" in Entity Framework, meaning that EF Core was completely re-written and created as a much lighter framework in regards footprint with very 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 EF Core intros and explanations are already available in Microsoft's documentation, this present 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, something important available in EF is, as introduced in the Domain Layer section, the capability of using POCO Domain Entities also known as POCO Code-First entities in EF jargon. That way, your Domain Model classes are "Persistence Ignorant" as the Persistence Ignorance and the Infrastructure Ignorance principles state.

In addition to that, there are some new possibilities since EF Core 1.1 like being able to have plain fields in your entities instead of properties with public/private setters, so if you want any entity field not to be accessible from the outside, you just create the attribute/field, no need to use private setters if you prefer this cleaner way.

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 means any developer using the parent entity class can add or remove items from its property collections. According to 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.

So, you can use a private collection while exposing a read-only IEnumerable, as shown in the code below.

```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 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).

EF Core provides a way to map the domain model to the physical database without "contaminating" your domain model so it is kept as pure .NET POCO code, because that "mapping action" can be implemented in your persistence layer. Precisely in that mapping action, you need to configure the fields to database mapping, in OnModelCreating, as shown below in the code in bold which 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 (method AddOrderItem()) for adding new items to the order, so behavior and data are tied and will be consistent throughout the entire 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, like the following class:

```
//usings…
namespace Microsoft.eShopOnContainers.Services.Ordering.Infrastructure.Repositories
{
    public class BuyerRepository : IBuyerRepository          Repository contract implemented
    {                                                         in the Domain Laver
        private readonly OrderingContext _context;

        public IUnitOfWork UnitOfWork
        {
            get
            {
                return _context;
            }
        }
}

        public BuyerRepository(OrderingContext context)
        {
            if (context == null)
            {                                                 The EF DbContext comes in the constructor
                throw new ArgumentNullException(              through Dependency Injection and is shared
                            nameof(context));                 between multiple Repositories within the same
            }                                                 HTTP request/scope thanks to its by default
                                                              lifetime (ServiceLifetime.Scoped) that can also
            _context = context;                               be explicitly set at services.AddDbContext<>
        }

        public Buyer Add(Buyer buyer)
        {
            return _context.Buyers                            Adds a Buyer entity to the
                .Add(buyer)                                   UnitOfWork (DbContext)
                .Entity;
        }

        public async Task<Buyer> FindAsync(string BuyerIdentityGuid)
        {
            var buyer = await _context.Buyers
                .Include(b => b.Payments)                     Optional query method
                .Where(b => b.FullName == BuyerIdentityGuid)
                .SingleOrDefaultAsync();

            return buyer;
        }
    }
```

Methods to implement in a Repository (Updates/Transactions vs. Queries)

In regards the persistence methods that should be implemented within each Repository class, you should usually place methods that are updating the state of entities contained by its related Aggregate (Remember the 1:1 relationship between an Aggregate and its related Repository), taking into account that an Aggregate-Root entity object might come with embedded child entities within the EF graph, like a Buyer having multiple PaymentMethods related as child entities.

Since the selected approach for the Ordering microservice in eShopOnContainers application is also based on CQS/CQRS, most of the queries are not implemented in custom repositories so developers have freedom to create the queries and joins they need for the presentation layer without the

restrictions coming from the Aggregates, custom Repositories per aggregate and DDD in general. Therefore, most of the custom repositories suggested by this guidance might only have update/transactional methods but not query methods, unless you need any specific query for the transactional operations, like it is the case of the BuyerRepository which also 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 these queries if needed by validations or 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, like when using it from an ASP.NET Core MVC controller. That is the way you can create the simples code like in the CRUD Catalog microservice in eShopOnContainers. So, in cases where you just want to have the simples code possible, you might want to directly use it.

However, implementing custom Repositories has several benefits when implementing more complex microservices or application. 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 each 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.
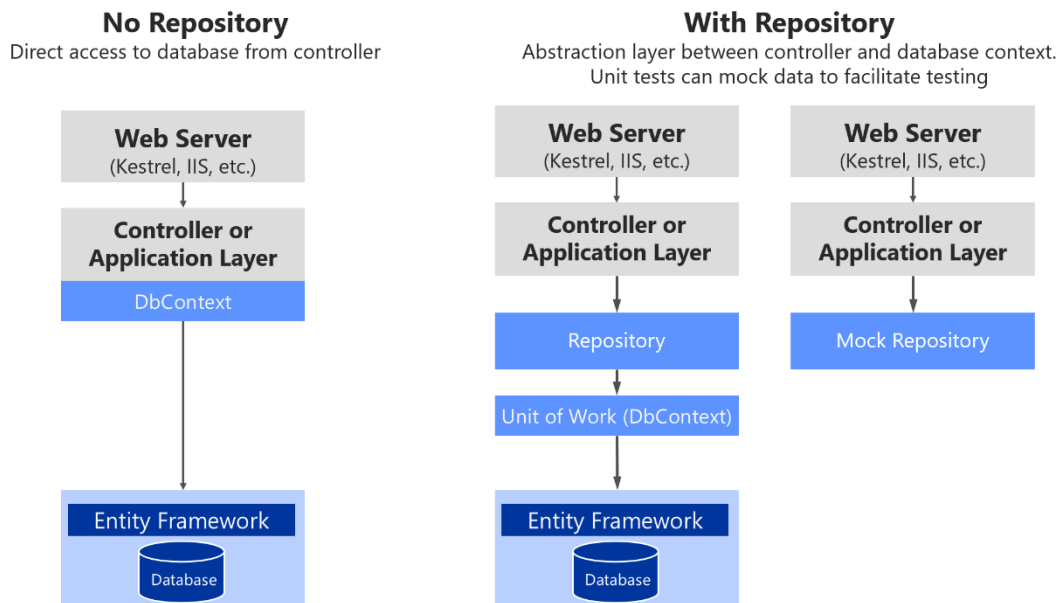
Figure X-XX. Using custom Repositories vs. plain DbContext

Architecting and developing Docker applications

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 have a better code easier to be tested with pure unit tests that are not impacted by the data tier state. If you were testing while accessing the real database through entity Framework, that wouldn't be unit tests but integration tests which are a lot slower.

If you were using DbContext directly, the only choice you have to run unit test-"ish" would be by using In-memory SQL Server with predictable data for unit tests. But 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 is 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 in the case when the operation being executed has to deal with multiple aggregates or simply because you are using multiple repository instances.

In order to do that, and as shown in the code below, the instance of the DbContext object has to be as ServiceLifetime.Scoped , which is in any case the default lifetime when registering your DbContext with services.AddDbContext<> into your IoC container, from the ConfigureServices() method of your Startup.cs at 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
            );
}
```

Usually, the DbContext instantiation mode should not be configured as ServiceLifetime.Transient neither as ServiceLifetime.Singleton.

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

Table Mapping

Table mapping identifies which table data should be queried from and saved to in the database.

You saw how your domain entities (like Product or Order) can be used to generate a related database schema. In EF, most of it is based on the concept of "Conventions". Those conventions are topics like *"What is going to be the name of a table?"*, or *"What property is going to be the primary key?"*, etc. and usually they are based on convention names (like *"a property ending with the prefix 'Id' will be the primary key"*).

By convention, each entity will be setup 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 included 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 either using 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 because you are contaminating your model with data annotations related to the infrastructure database. On the other hand, Fluent API is a pretty 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(), but it is advisable to partition that code and have multiple sub-methods, one per entity, like shown in the code above.

The Hi/Lo pattern in EF Core

An interesting configuration in that code is that it is using a Hilo as the key generation strategy based on the *Hi/Lo pattern*. EF Core supports "HiLo" out of the box with the *ForSqlServerUseSequenceHiLo* method.

The *Hi/Lo pattern* describe a mechanism for generating safe-ids on the client side rather than 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 generator in other DBMS.
- It generates human readable identifier unlike to GUID techniques.

Mapping Fields instead of Properties

With this 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 could not have any property related, neither for setter or getter. That field could be calculated within the order's business logic and used from the order's methods, too, so it needs not to be a property. However, it needs to be persisted in the database. So, 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 actually see when it is that mapping performed with the *PropertyAccessMode.Field* configuration highlighted in the previous code.

There are many other Fluent API configuration that you can research available at the reference link, below.

Shadow Properties and Value-Objects

Shadow properties are properties that do not exist in your entity class. The value and state of these properties is 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 moment of this writing , Entity Framework 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 Is 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

No-SQL databases as your persistence infrastructure

When using No-SQL databases for your infrastructure data tier you wouldn't usually be using an ORM like Entity Framework Core but directly the API provided by the chosen No-SQL engine like Azure Document DB, MongoDB, Cassandra, RavenDB, CouchDB, Azure Storage Tables, etc.

However, if using a No- SQL database, especially when using Document-oriented databases like Azure Document DB, CouchDB and RavenDb, the way you design your model with DDD Aggregates would be similar in regards 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 whole document (serialized in JSON or any other format).*

The difference would be the way you persist that model. But this is why when implementing a Domain Model, you want to have a model based on POCO entity classes, agnostic to the infrastructure persistence, so potentially you could move to a different persistence infrastructure. Although that is not 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 No-Sql databases the entities will be a lot more de-normalized, so it is not a simple table mapping at all. Your domain model might have a few impacts, after all.

However, if you were modelling your Domain Model based on Aggregates, moving to No-Sql and document oriented databases might be a lot easier because you already defined the aggregate's boundaries which are pretty similar to serialized documents in document-oriented databases.

For instance, the following JSON code will be a sample implementation of an Order Aggregate, similar to the order aggregate we implemented in eShopOnContainers, but using EF 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 and to be used by, for instance, the Azure Document DB SDK, it would be pretty similar to the C# POCO classes used with EF Core. The difference will be the way to used them from the application and infrastructure layers, like in the following code.

```
//C# example of an Order Aggregate being persisted with DocumentDB API

// *** Domain Model Code ***
// Aggregate: Create an Order object with its child entities and/or value-objects.
// Then, use AggregateRoot's methods to add the nested objects so invariants and
// logic is consistent across the nested properties (Value-Objects and entities).
// This can be saved as JSON as is without converting into rows/columns.

Order orderAggregate = new Order
{
    Id = "2017001",
    OrderDate = new DateTime(2005, 7, 1),
    BuyerId = "1234567",
    PurchaseOrderNumber = "PO18009186470"
}


Address address = new Address
{
    Street = "100 One Microsoft Way",
    City = "Redmond",
    State = "WA",
```

```csharp
    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 could be pretty similar to the way you are using it in your Domain Model Layer even 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 No-SQL db, implemented in the infrastructure and persistence layer, then is where the code and API will dramatically change internally.

References – No-SQL Databases
Azure Document DB https://docs.microsoft.com/en-us/azure/documentdb/documentdb-modeling-data
DDD Aggregate storage https://vaughnvernon.co/?p=942
Event storage https://github.com/NEventStore/NEventStore

Designing the microservice's Application Layer and Web API

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 like when developing a microservice with DDD patterns. However, you should also use DI and apply the SOLID principles even when not developing with DDD approaches or patterns.

SOLID is an acronym that groups five fundamental principles:
- Single Responsibility Principle
- Open/close principle
- Liskov substitution principle
- Inversion Segregation principle
- Dependency Inversion principle

SOLID 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 decoupled 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 in order to perform its actions are provided/injected to 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 Invesion of Controlo (IoC) containers. ASP.NET Core provides a simple built-in IoC container, but you can also plug your favorite IoC container, like Autofac, Ninject, etc.

By following the SOLID 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 constructor it will be easy to detect by just taking a look to the number of parameters of your constructor. If there are too many dependencies, this is generally a "bad smell", a sign 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 SOLID 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 on these topics.

In case you are not familiar with SOLID and DI, please read the information from the links below at the references table.

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, unless 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.

As introduced, 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.

Usually, you'd want to inject dependencies that implement infrastructure objects. The most typical dependency to inject are the already introduced 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 IBuyerRepository _buyerRepository;
    private readonly IOrderRepository _orderRepository;

    public CreateOrderCommandHandler(IBuyerRepository buyerRepository,
                                     IOrderRepository orderRepository)
    {
        if (buyerRepository == null)
        {
            throw new ArgumentNullException(nameof(buyerRepository));
        }

        if (orderRepository == null)
        {
            throw new ArgumentNullException(nameof(orderRepository));
        }

        _buyerRepository = buyerRepository;
        _orderRepository = orderRepository;
    }

    public async Task<bool> Handle(CreateOrderCommand message)
    {
        //
        // ... Additional code
        //

        // Create the Order AggregateRoot
        // Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate
```

```
            var order = new Order(buyer.Id, payment.Id,
                            new Address(message.Street,
                                    message.City, message.State,
                                    message.Country, message.ZipCode));

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                            item.Discount, item.PictureUrl, item.Units);
        }

        //Persist the Order through the Repository
        _orderRepository.Add(order);

        var result = await _orderRepository.UnitOfWork
                                        .SaveChangesAsync();

        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

The other side of the coin that you need to know is 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 simple built-in IoC container provided by ASP.NET Core (like the simple Catalog microservice from eShopOncontainers), you register the types in the ConfigureServices() method from the MVC Startup.cs

```
// 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.AddTransient<IEmailSender, AuthMessageSender>();

    services.AddTransient<IMyCustomRepository, MyCustomSQLServerRepository>();

}
```

For instance, the last line of code means that when any of my constructors has a dependency on IMyCustomRepository (interface or abstraction), the IoC container will inject an instance of the MyCustomSQLServerRepository implementation class.

Using Autofac as IoC container

However, you can also use additional IoC containers and plug them to the ASP.NET Core pipeline, like in the Ordering microservice which is using Autofac. When using Autofac you usually register the types in thorugh "modules" which allow you to split the registration types in multiple files depending on where your types are, as you could have the application types distributed across multiple class libraries.

For instance, this is the application module for one class library with the implemented custom types.

```csharp
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();
    }
}
```

As you can see 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. Simplifying to the most important cases, when a request is made for a dependency, the IoC container can return a single instance per LifetimeScope (called in ASP.NET Core as "scoped"), a new instance per dependency (named in ASP.NET Core as "transient"), or a single instance shared across every object using the IoC container (named in ASP.NET Core as "singleton").

For additional information about DI, lifetime scopes and usage in ASP.NET Core, read the following references.

References – ASP.NET Core DI and Autofac
Using Dependency Injection in ASP.NET Core and .NET Core https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection **Autofac** http://docs.autofac.org/en/latest/getting-started/index.html http://docs.autofac.org/en/latest/lifetime/instance-scope.html **Comparing lifetime scopes between ASP.NET Core built-in container and Autofac** https://blogs.msdn.microsoft.com/cesardelatorre/2017/01/26/comparing-asp-net-core-ioc-service-life-times-and-autofac-ioc-instance-scopes/

Implementing the Command and Command-Handlers patterns

In the DI through constructor example shown in the previous section, the IoC container was injecting Repositories through a constructor, but where? - In a very simple Web API (like the Catalog microservice from eShopOnContainers), you would be doing that at the MVC Controllers level, at a Controller constructor. However, in the previous example it is done at a CommandHandler level, so let's explain what a Command 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 (already explained using in this approach simplified queries with Dapper Micro ORM) and the Commands as starting point for the transactions/writes.

Remember, CQRS is not an architecture, it's a pattern which you can use just in some microservices of your application architecture or in all of them. But you decide if you implement CQRS per bounded-context or microservice not as the top-level architecture for your whole application.
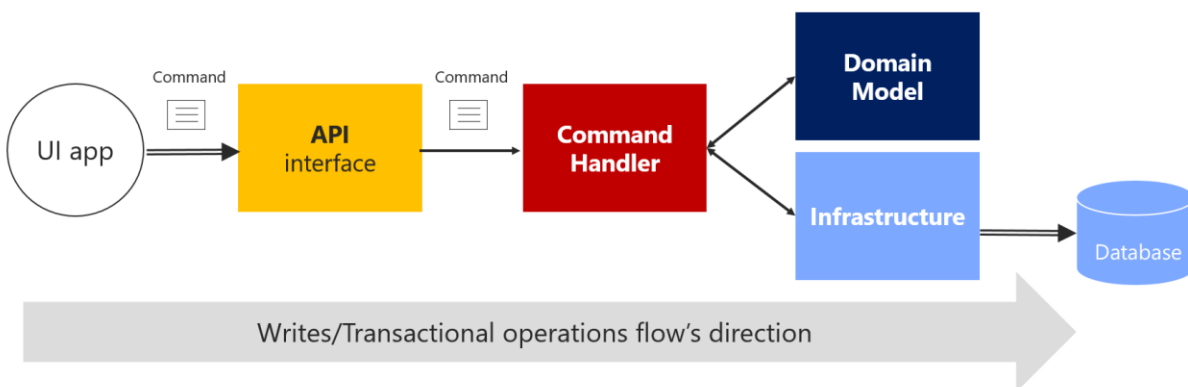
As shown in the high-level diagram below, the

Figure X-XX. High level "Writes side" in a CQRS pattern

The Command

What is a command? – Client apps request changes/transactions to the domain by sending commands. They are named with a verb in the imperative mood plus 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.

Another very important characteristic of a command is that *a command must be processed just once* and it has to be idempotent. 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 as systems interested on that event.

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.

Then, how do you implement a Command? – It is really simple, a Command is implemented with a class with data fields or collections containing 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.

Here's for example a simplified CreateOrderCommand used in the Ordering microservice from eShopOnContainers.

```csharp
public class CreateOrderCommand
    :IAsyncRequest<bool>
{
    private readonly List<OrderItemDTO> _orderItems;

    public string City { get; set; }
    public string Street { get; set; }
    public string State { get; set; }
    public string Country { get; set; }
    public string ZipCode { get; set; }
    public string CardNumber { get; set; }
    public string CardHolderName { get; set; }
    public DateTime CardExpiration { get; set; }
    public string CardSecurityNumber { get; set; }
    public int CardTypeId { get; set; }

    public string BuyerIdentityGuid { get; set; }
    public IEnumerable<OrderItemDTO> OrderItems => _orderItems;

    public CreateOrderCommand()
    {
        _orderItems = new List<OrderItemDTO>();
    }
}
```

Basically, that Command class (similar to a DTO) will contain 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 data for reading, and no behavior*. The Command's name indicates the

purpose. In many languages like C#, they are represented as classes, but they are not true classes in the real OO sense.

As an arguable 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 is not having any setters, or other methods which change internal state. This immutability and "no setters" is however an improvement of the Command's design, but it is not critical.

An example is the "Create an order" command. In this case, the Command class might be similar in regards data to the Order you want to create, but probably you don't need the same attributes. For instance, the CreateOrderCommand still doesn't have any Oder Id because it hasn't been created just yet.

Many other Command classes can be very simple, like having just a few fields about some state that needs to be changed. For instance, 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.

```
public class UpdateOrderStatusCommand
    :IAsyncRequest<bool>
{
    public string Status { get; set; }
    public string OrderId { get; set; }
    public string BuyerIdentityGuid { get; set; }
}
```

The Command-Handler class

Okay, the Command class was pretty obvious. But where do you actually use that command object and provide the needed data to the Domain objects? – In any Web API controller? an Application Layer Service?

Well, 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 important point here is that all the 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 means to get the domain model out of the persistent store plus telling to 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 it and just push the behavior down to the domain objects (aggregate-root's and child entity's methods) as needed by refactoring it.

As an example of a Command-Handler class, you can see the again the CreateOrderCommandHandler class from the Ordering microservice in eShopOnContainers that you previously saw when tacklying about DI. In this case you can see highlighted the actual Handle() method and the operations with the Domain model objects/aggregates.

```
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
    private readonly IBuyerRepository _buyerRepository;
    private readonly IOrderRepository _orderRepository;

    public CreateOrderCommandHandler(IBuyerRepository buyerRepository,
                                     IOrderRepository orderRepository)
    {
        if (buyerRepository == null)
        {
            throw new ArgumentNullException(nameof(buyerRepository));
        }

        if (orderRepository == null)
        {
            throw new ArgumentNullException(nameof(orderRepository));
        }

        _buyerRepository = buyerRepository;
        _orderRepository = orderRepository;
    }
    public async Task<bool> Handle(CreateOrderCommand message)
    {
        //
        // ... Additional code
        //

        // Create the Order AggregateRoot
        // Add child entities and value-objects through the Order Aggregate-Root
        // methods and constructor so validations, invariants and business logic
        // make sure that consistency is preserved across the whole aggregate

        var order = new Order(buyer.Id, payment.Id,
                              new Address(message.Street,
                                          message.City, message.State,
                                          message.Country, message.ZipCode));

        foreach (var item in message.OrderItems)
        {
            order.AddOrderItem(item.ProductId, item.ProductName, item.UnitPrice,
                               item.Discount, item.PictureUrl, item.Units);
        }

        //Persist the Order through the Aggregate's Repository
        _orderRepository.Add(order);

        var result = await _orderRepository.UnitOfWork
                                                .SaveChangesAsync();

        return result > 0;
    }
}
```

This is the common sequence of steps a command handler might follow:

- Validate the command's data incoming.
- 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/use a Command-Handler from? – 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 some kind of 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.
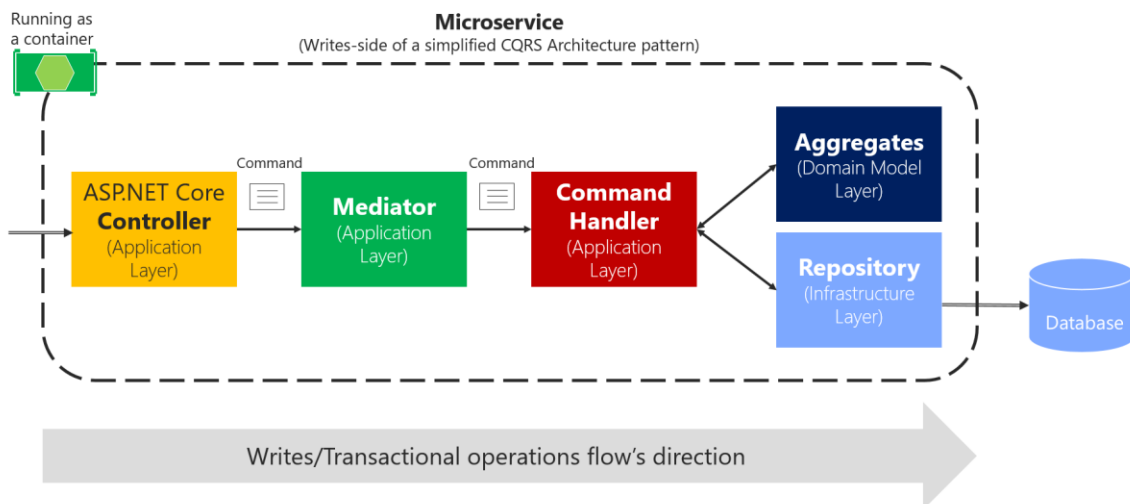

Figure X-XX. Using the Mediator pattern in CQRS

Architecting and developing Docker applications

The reason why using a mediator pattern makes sense is because in enterprise applications the processing requests can get more and more complicated and in those cases, you will want to be able to add an open number of cross-cutting concerns like logging, validations, transactions, audit, security, etc. 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 very centralized and transparent way by just applying "decorators". See the decorator pattern.

Decorators are like a déjà vu of Aspect Oriented Programming – AOP – but 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 calls interception. Both typical AOP approaches are "like magic" and when dealing with serious issues or bugs it 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
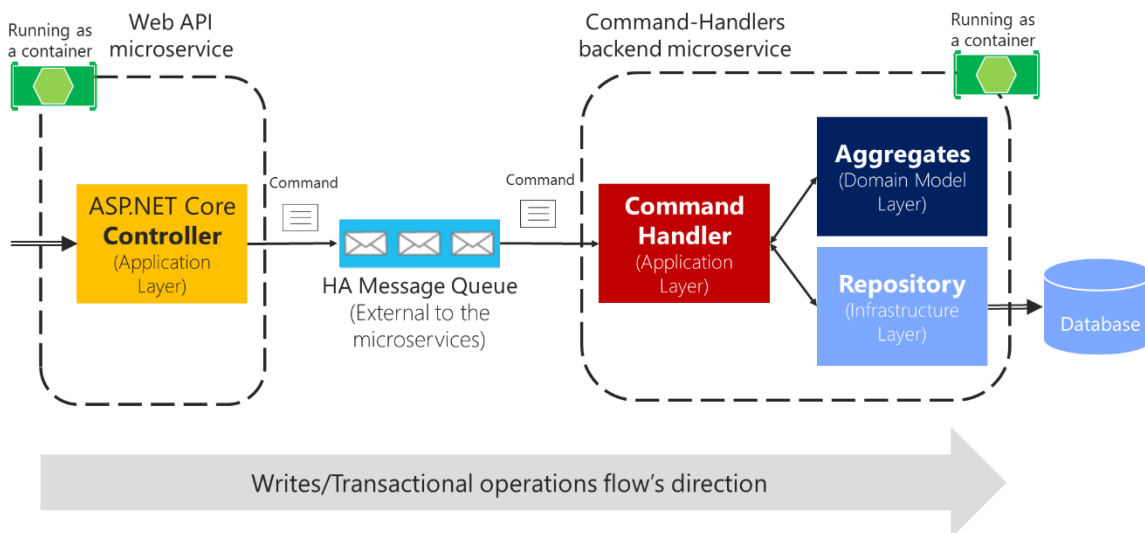
Using message queues to accept the commands can complicate further your command's pipeline as you will probably need to split the pipeline in two processes connected through the external message queue, but it should be used if you need to have a better resiliency when submitting the command messages plus having better scalability and even performance because of the asynchronous messaging you can implement. Take into account that in this case the controller just posts the

command message into the queue and returns back. Then, the command-handlers will be processing the messages at its own pace. That is a great benefit typical from queues, as the message queue can act as a buffer in cases when hyper scalability is needed for ingress data, like stocks or any other scenario with a high volume of ingress data.

However, and precisely because of the asynchronism you implement with message queues, you will need to figure out how to respond to the client application about the success or failure of the command's process because, make no mistake about it, as a general rule you should not use "fire and forget commands". Every business application will need to know if a command was processed successfully or at least validated and accepted. It shouldn't be a pure "fire a forget" command.

Thus, being able to respond to the client after validating a command message that was submitted to an asynchronous queue involves additional complexity to your system compared to an in-process command process that can return the operation's result after running the transaction. Using queues you might need to return the result of the command process maybe through other "operation result messages" which will need additional components and custom communication in your system.

In any case, this should be a decision based on your application's or microservice's business and quality of service requirements.

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 as a way to separate cross-cutting concerns.

About the implementation in .NET Core, there are multiple open source libraries available implementing the mediator pattern, but the chosen library used in this guidance is the open source library called *MediatR* (built by Jimmy Bogard) which is a small, simple but neat 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 it 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 of all, 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 keeps a very clean and lean Web API controller. And within the controller's methods, the code is also pretty simple, almost just one line sending a Command to the mediator object.

```csharp
[Route("new")]
[HttpPost]
public async Task<IActionResult> CreateOrder([FromBody]CreateOrderCommand
                                                    createOrderCommand)
{
    var result = await _mediator.SendAsync(createOrderCommand);
    if (result)
    {
        return Ok();
    }
    return BadRequest();

}
```

In order for Mediator 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 code below is how you can 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 related each Command with its Command-Handler because that relationship is stated in the CommandHandler class, like in the following example.

```csharp
public class CreateOrderCommandHandler
    : IAsyncRequestHandler<CreateOrderCommand, bool>
{
```

So, this is the piece that closes the loop and correlates Commands with CommandHandlers. The handler is just a simple class, but it inherits from RequestHandler<T> and MediatR makes sure it gets invoked with the correct payload.

Applying cross-cutting concerns when processing commands with the Mediator and Decorator patterns

There's one more thing, the capability of being able to apply cross-cutting concerns to the mediator pipeline. In the Autofac registration module code you can also see at the end of that code how it is registering a decorator type, specifically, a custom "Log Decorator".

That LogDecorator class can be implemented as the following simple code which is simply logging info about the command handler being executed and whether it was successful or not.

```csharp
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($"Succedded 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, check 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

Implementing event based communication between microservices: Integration Events

Defining an Event Bus interface

TBD

Multiple implementations of an Event Bus

TBD

Implementing a simple Event Bus with a SignalR Hub service

TBD

Implementing an Event Bus with Azure Service Bus

TBD

Intro to an Event Bus implementation with NServiceBus

TBD

Intro to Event Bus implementation with RabbitMQ

TBD

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

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

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

In this document, the docker-compose.yml file was introduced in the section "*Step 4. Define your services in docker-compose.yml when building a multi-container Docker app with multiple services*", however, there are a few additional interesting definitions and details that is worth to dig into.

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

```
version: '2'
services:
  webmvc:
    image: eshop/webmvc
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
    ports:
      - "5100:80"
    depends_on:
      - catalog.api
      - identity.data
      - basket.api
  webspa:
    image: eshop/webspa
    environment:
      - CatalogUrl=http://catalog.api
      - OrderingUrl=http://ordering.api
    ports:
      - "5104:80"
    depends_on:
      - catalog.api
      - identity.data
      - basket.api
  catalog.api:
    image: eshop/catalog.api
    environment:
      - ConnectionString=Server=catalog.data;Initial Catalog=CatalogData;User Id=sa;Password=your@password
    expose:
      - "80"
    ports:
      - "5101:80"
    depends_on:
      - catalog.data
  catalog.data:
    image: microsoft/mssql-server-linux
    environment:
      - SA_PASSWORD=Pass@word
      - ACCEPT_EULA=Y
    ports:
      - "5434:1433"
  ordering.api:
    image: eshop/ordering.api
    environment:
      - ConnectionString=Server=ordering.data;Database=Microsoft.eShopOnContainers.Services.OrderingDb;User
Id=sa;Password=your@password
    ports:
      - "5102:80"
# (Go to Production): For secured/final deployment, remove Ports mapping and
# leave just the internal expose section
#    expose:
#      - "80"
    extra_hosts:
      - "CESARDLBOOKVHD:10.0.75.1"
    depends_on:
      - ordering.data
  ordering.data:
    image: eshop/ordering.data.sqlserver.linux
    ports:
      - "5432:1433"
```

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

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

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

A simple Web Service API container

The catalog.api container-microservice has a simple and straightforward definition:

```
    catalog.api:
      image: eshop/catalog.api
      environment:
        - ConnectionString=Server=catalog.data;Initial Catalog=CatalogData;User Id=sa;Password=your@password
      expose:
        - "80"
      ports:
        - "5101:80"
      depends_on:
        - catalog.data
```

This containerized service has the following basic configuration in place:

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

Architecting and developing Docker applications

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

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

Additional settings from docker-compose.yml

TBD-CDLTLL

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

  extra_hosts:
    - "CESARDLBOOKVHD:10.0.75.1"
```

A database server running as a container

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

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

```
catalog.data:
  image: microsoft/mssql-server-linux
  environment:
```

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

A similar command could be run directly with "docker run".

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

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

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

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

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

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

References – SQL Server for Linux running on Docker containers
Run the SQL Server Docker image on Linux, Mac, or Windows https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-setup-docker
Connect and query SQL Server on Linux with sqlcmd https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-connect-and-query-sqlcmd

Seeding with Test Data on the Web API Startup

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

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

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

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

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

  }
```

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

```csharp
public class CatalogContextSeed
{
    public static async Task SeedAsync(IApplicationBuilder applicationBuilder)
    {
        var context = (CatalogContext)applicationBuilder
            .ApplicationServices.GetService(typeof(CatalogContext));

        using (context)
        {
            context.Database.Migrate();

            if (!context.CatalogBrands.Any())
            {
                context.CatalogBrands.AddRange(
                    GetPreconfiguredCatalogBrands());

                await context.SaveChangesAsync();
            }

            if (!context.CatalogTypes.Any())
            {
                context.CatalogTypes.AddRange(
                    GetPreconfiguredCatalogTypes());

                await context.SaveChangesAsync();
            }

        }
    }

    static IEnumerable<CatalogBrand> GetPreconfiguredCatalogBrands()
    {
        return new List<CatalogBrand>()
        {
            new CatalogBrand() { Brand = "Azure"},
            new CatalogBrand() { Brand = ".NET" },
            new CatalogBrand() { Brand = "Visual Studio" },
            new CatalogBrand() { Brand = "SQL Server" }
        };
    }

    static IEnumerable<CatalogType> GetPreconfiguredCatalogTypes()
    {
        return new List<CatalogType>()
        {
            new CatalogType() { Type = "Mug"},
            new CatalogType() { Type = "T-Shirt" },
            new CatalogType() { Type = "Backpack" },
            new CatalogType() { Type = "USB Memory Stick" }
        };
    }
}
```

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

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

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

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

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

        //(Versus commented DbContext using a SQL Server provider
        //services.AddDbContext<CatalogContext>(c =>
        //{
        //    c.UseSqlServer(Configuration["ConnectionString"]);
        //
        //});
    }
    //Other Startup code…
}
```

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

Redis cache service running in a container

TBD

Testing ASP.NET Core services and web apps

Controllers are a central part of any ASP.NET Core API service and MVC web app. As such, you should have confidence they behave as intended for your app. Automated tests can provide you with this confidence and can detect errors before they reach production.

You need to Test how the controller behaves based on valid or invalid inputs and test controller responses based on the result of the business operation it performs.

However, there are several main differentiated types of tests you should have for your microservices. Unit Tests, Integration Tests, Functional Tests (per microservice) and Service Tests.

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

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

Implementing Unit Tests for ASP.NET Core Web APIs

Unit testing involves testing a part of an app in isolation from its infrastructure and dependencies. When unit testing controller logic, only the contents of a single action is tested, not the behavior of its dependencies or of the framework itself. As you unit test your controller actions, make sure you focus only on its behavior. A controller unit test avoids things like filters, routing, or model binding. By focusing on testing just one thing, unit tests are generally simple to write and quick to run. A well-written set of unit tests can be run frequently without much overhead. However, unit tests do not detect issues in the interaction between components, which is the purpose of integration testing.

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

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

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

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

Implementing Integration and Functional Tests per isolated microservice

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

Integration testing ensures that an application's components function correctly when assembled together. ASP.NET Core supports integration testing using unit test frameworks and a built-in test web host that can be used to handle requests without network overhead.

Unlike Unit testing, integration tests frequently involve application infrastructure concerns, such as a database, file system, network resources, or web requests and responses. Unit tests use fakes or mock objects in place of these concerns, but the purpose of integration tests is to confirm that the system works as expected with these systems, so in this case you won't use fakes or mock objects but including the infrastructure, like database access or services invocation from the outside.

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

ASP.NET Core includes a *test host* available in a NuGet component as Microsoft.AspNetCore.TestHost that can be added to integration test projects and used to host ASP.NET Core applications, serving test requests without the need for a real web host.

AS you can see in the following code, when creating integration tests of ASP.NET Core controllers, you would 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

Implementing Service Tests on a multi-container application

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

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

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

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

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's 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 intead of just partially unresponsive, as show in the image X-XX below.

Partial failures
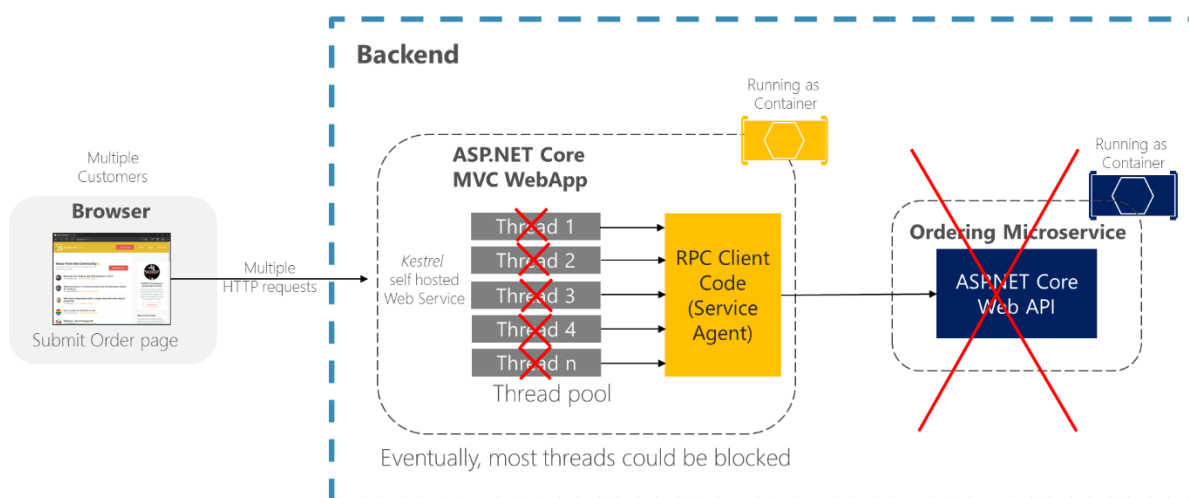
In a large microservice based application this partial failure can be very much amplified. Think about a system that receives millions of incoming calls per day which in turn fans out to many more millions of

outgoing calls (let's suppose a ratio of 1:5) to tens of underlying or internal microservices as dependencies, as shown in image X-XX.

# Multiple distributed dependencies
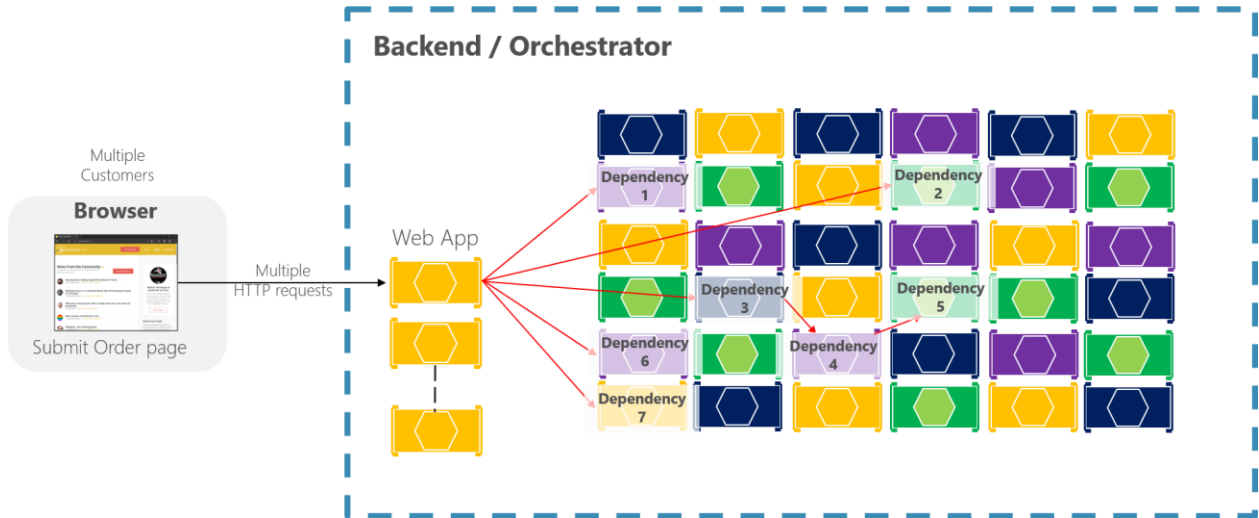
Intermittent failure is guaranteed that will happen in a distributed and cloud based system, even if every dependency itself has excellent availability and uptime.

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 are Failure Amplified in Microservices
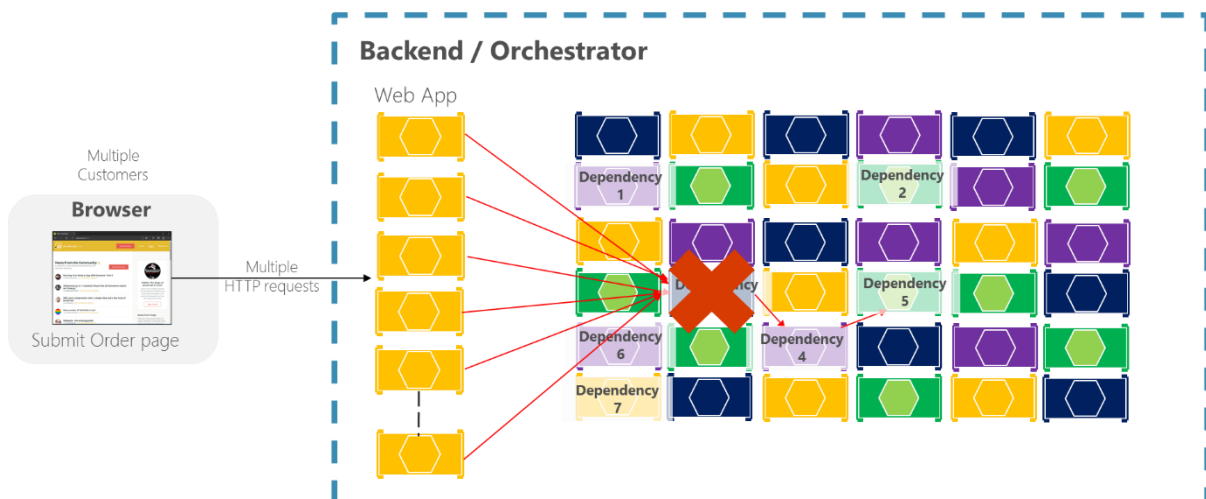
Therefore, it is a requirement of high volume, high availability applications to design and build resilient microservices and client applications into their architecture.

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.

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.

Implementing Retries Logic

TBD – Implementation

Implementing Circuit Breaker pattern

TBD – Implementation

Implementing Fallbacks

TBD – Implementation

Implementing timeouts

TBD – Implementation

Implementing Graceful Shutdowns

TBD – Implementation

References – Securing .NET Applications
TBD https://tbd

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

Securing .NET microservices and web applications

Encrypting application settings

TBD

Safe storage of app secrets during development

TBD

Using Azure Key Vault to protect secrets in production time

TBD

Securing the microservices' communication

TBD

References – Securing .NET Applications
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

TBD

Conclusions

Key takeaways

- Container based solutions provide important benefits of cost savings because containers are a solution to deployment problems cause by the lack of dependencies in production environments, therefore, improving DevOps and production operations significantly.

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

- A Docker container is becoming the standard unit of deployment for any server-based application or service.

- Docker orchestrators like the ones provided in Azure Container Service (Mesos DC/OS, Docker Swarm, Kubernetes) and Azure Service Fabric are fundamental and indispensable for any microservice-based or multi-container application with significant complexity and scalability needs.

- An end-to-end DevOps environment supporting CI/CD connecting to the production Docker environments provides agility and ultimately improves the time to market of your applications.

- Visual Studio Team Services greatly simplifies your DevOps environment targeting Docker environments from your Continuous Deployment (CD) pipelines, including simple Docker environments or more advanced microservice and container orchestrators based on Azure.