

Containerized Docker Application Lifecycle with Microsoft Platform and Tools



DevDiv, .NET and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Copyright © 2016 by Microsoft Corporation

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

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

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

Microsoft and the trademarks listed at <http://www.microsoft.com> on the “Trademarks” webpage are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

Author:

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

Participants and reviewers:

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

Section 1: Summary	1
Purpose.....	1
Who should use this guide	1
How you can use this guide.....	1
Section 2: Introduction to containers and Docker	2
What are containers?.....	2
What is Docker?.....	3
Comparing Docker containers with virtual machines.....	4
What is Container as a Service?.....	5
Basic Docker definitions	5
Basic Docker taxonomy: containers, images, and registries	7
Section 3: Introduction to the Docker application lifecycle	8
Containers as the foundation for DevOps collaboration	8
Introduction to a generic E2E Docker application lifecycle workflow	9
Benefits from DevOps for containerized applications.....	10
Section 4: Introduction to the Microsoft platform and tools for containerized applications	11
Vision	11
Section 5: Architecting and developing containerized applications with Docker and Azure	14
Vision	14
Architecting Docker applications.....	14
Common container design principles.....	14
Container equals a process.....	14
Monolithic applications.....	15
Monolithic application deployed as a container	16
Publishing a single Docker container app to Azure App Service.....	17
State and data in Docker applications	18
Service-oriented architecture applications.....	19
Microservices, multiple services per app and service orientation approaches	20
Docker clusters in Microsoft Azure	22
Azure Container Service	23
Development environment for Docker apps	25
Development tools choices: IDE or editor.....	25

Language and framework choices	25
Inner-loop development workflow for Docker apps	26
Workflow for building a single app inside a Docker container using Visual Studio Code and Docker CLI	26
Using Visual Studio Tools for Docker (Visual Studio on Windows)	35
Using PowerShell commands in Dockerfile to setup Windows Containers (Docker standard based)	37
Section 6: Docker application DevOps workflow with Microsoft tools	38
Steps in the outer-loop DevOps workflow for a Docker application	39
Step 1. Inner loop development workflow	39
Step 2. SCC integration and management with Visual Studio Team Services and Git	39
Step 3. Build, CI, Integrate and Test with VSTS and Docker	40
Step 4. Continuous Delivery (CD), Deploy	45
Step 5. Run and manage	49
Step 6. Monitor and diagnose	49
Section 7: Running, managing and monitoring Docker production environments	50
Running composed and microservices-based applications in production environments	50
Intro to orchestrators, schedulers, and container clusters	50
Managing production Docker environments	51
Azure Container Service and management tools	51
Azure Service Fabric	52
Monitoring containerized application services	53
Microsoft Application Insights	53
Microsoft Operations Management Suite (OMS)	54
Section 8: Conclusions	56
Key takeaways	56

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 DevOps context, containers enable continuity in the CI/CD model as they create a clear boundary between developers by providing containerized apps with all the required environment configuration, and ITOps that builds a generic environment to run app specific content. 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.

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

If you are new to Docker, it is recommended to start from the beginning and read the initial introduction to Docker containing the definition of fundamental Docker terms, including containers, images, registry, clusters, orchestrators, and Docker itself.

On the other hand, if you are already familiar with Docker and you just want to know what Microsoft has to offer about it, it is recommended to start with the section "*Introduction to the Microsoft platform and tools for Containerized Applications*" and continue from there.

Introduction to containers and Docker

What are containers?

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

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

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

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

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

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

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

In short and as the main takeaways, the main benefits provided by containers are Isolation, Portability, Agility, Scalability and Control across the whole application lifecycle workflow. But the most important benefit is the isolation provided between Dev and Ops.

What is Docker?

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

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

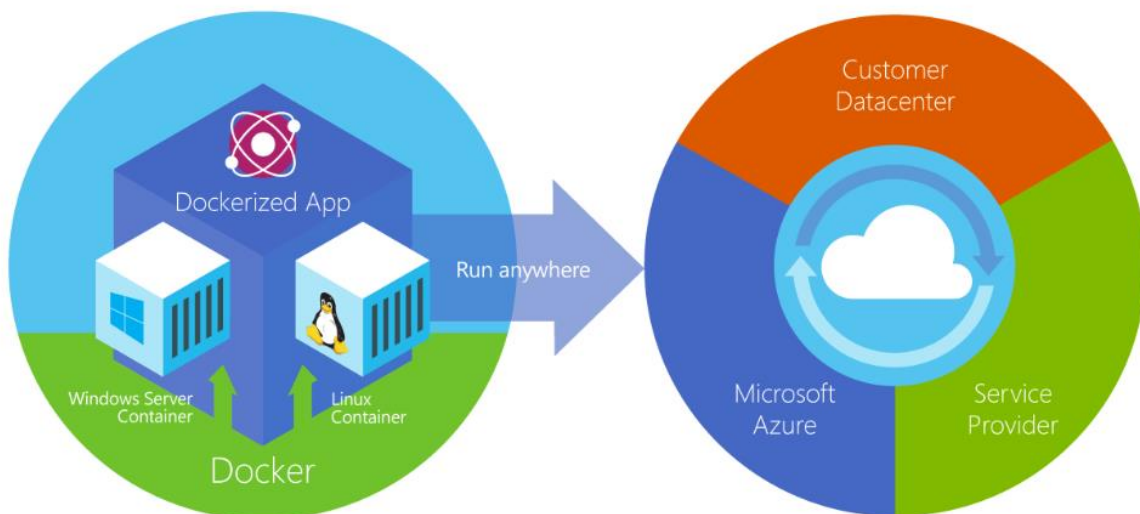


Figure 2-1. Docker deploys containers at all layers of the hybrid cloud

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

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

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

Related to [Windows Containers](#), there are two types or runtimes:

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

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

Comparing Docker containers with virtual machines

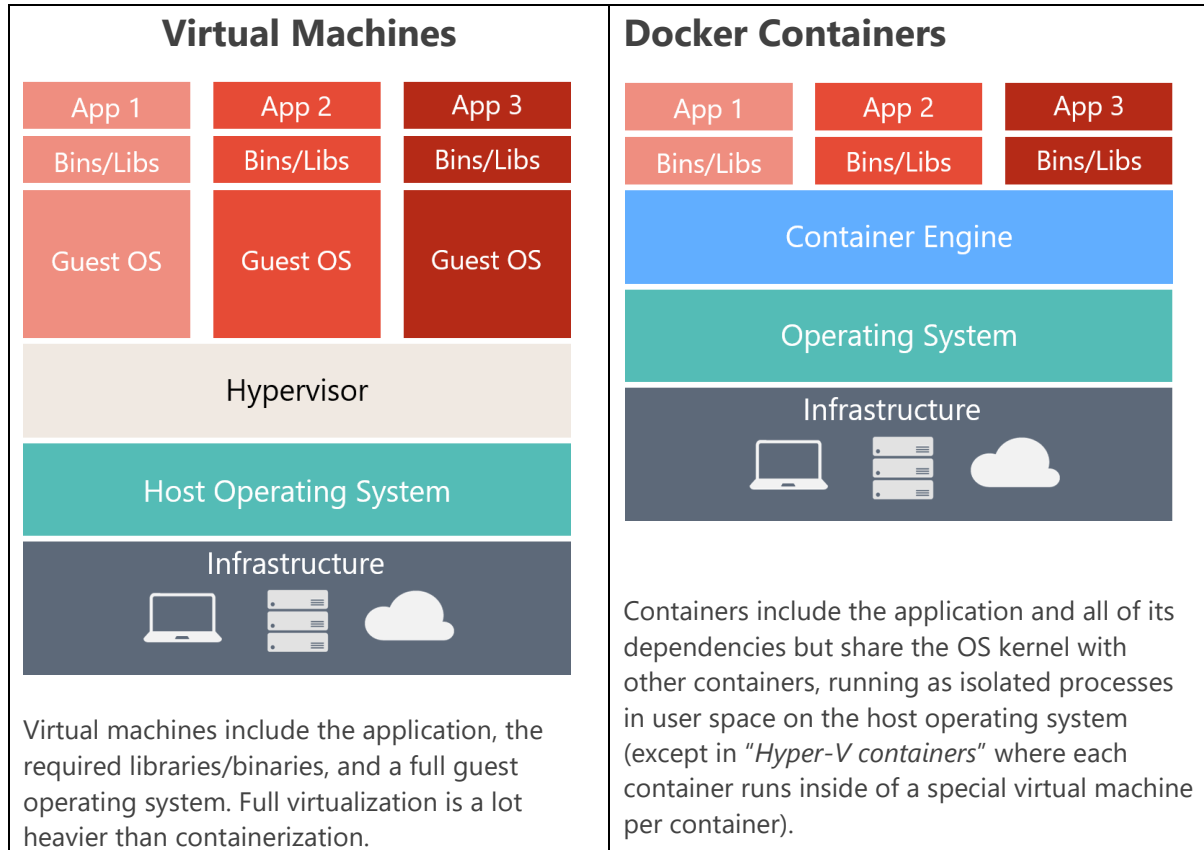


Figure 2-2. Comparison of traditional virtual machines to Docker containers

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

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

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

Coming back to the container's benefits, when using Docker, you won't get the typical developer's statement "it works on my machine". But you can simply say "it runs on Docker" because the packaged Docker application can be executed on any supported Docker environment and it will run the way it was intended to do it on all the deployment targets (Dev/QA/Staging/Production, etc.).

What is Container as a Service?

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

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

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

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

Basic Docker definitions

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Basic Docker taxonomy: containers, images, and registries

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

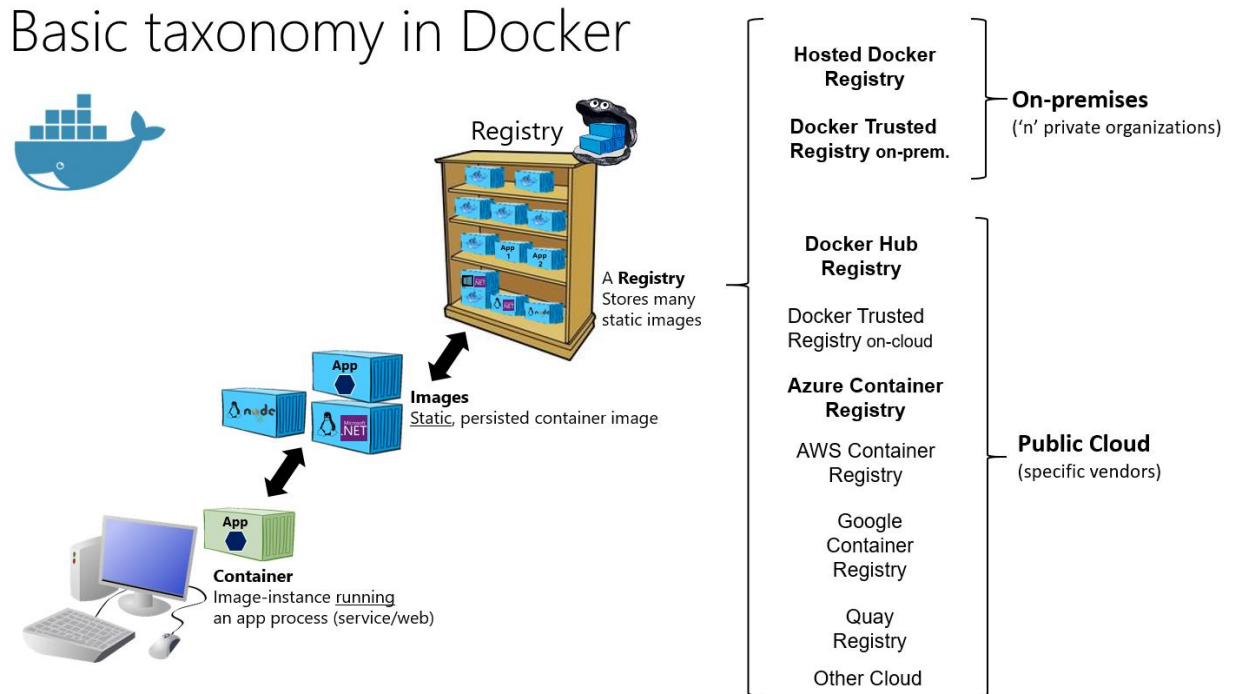


Figure 2-3. Taxonomy of Docker terms and concepts

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

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

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

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

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

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

Introduction to the Docker application lifecycle

Containers as the foundation for DevOps collaboration

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

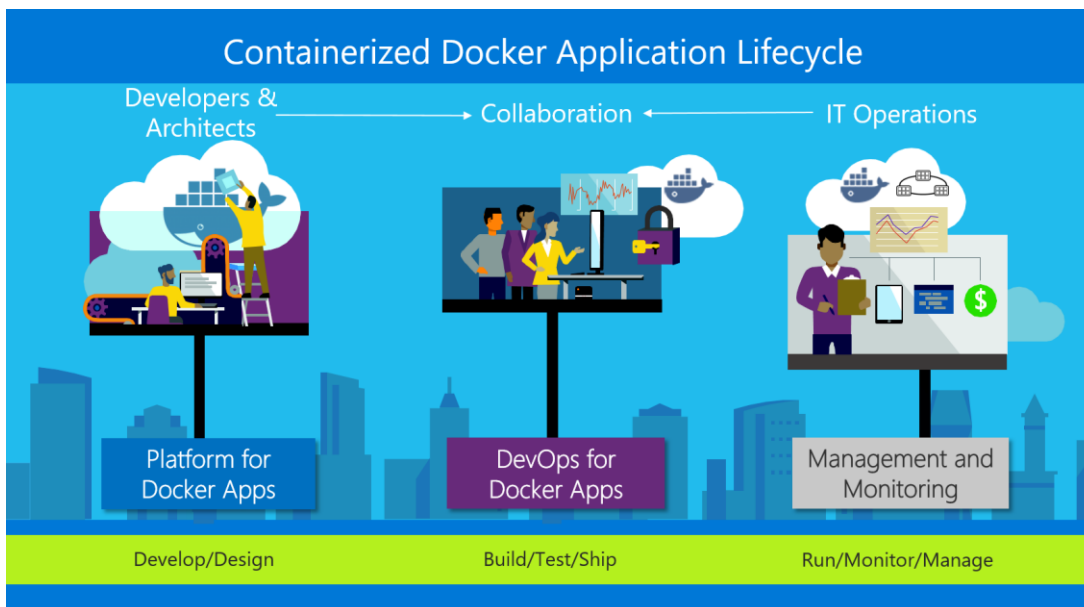


Figure 3-1. Main workloads per “personas” in the lifecycle for containerized Docker applications

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

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

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

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

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

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

Introduction to a generic E2E Docker application lifecycle workflow

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

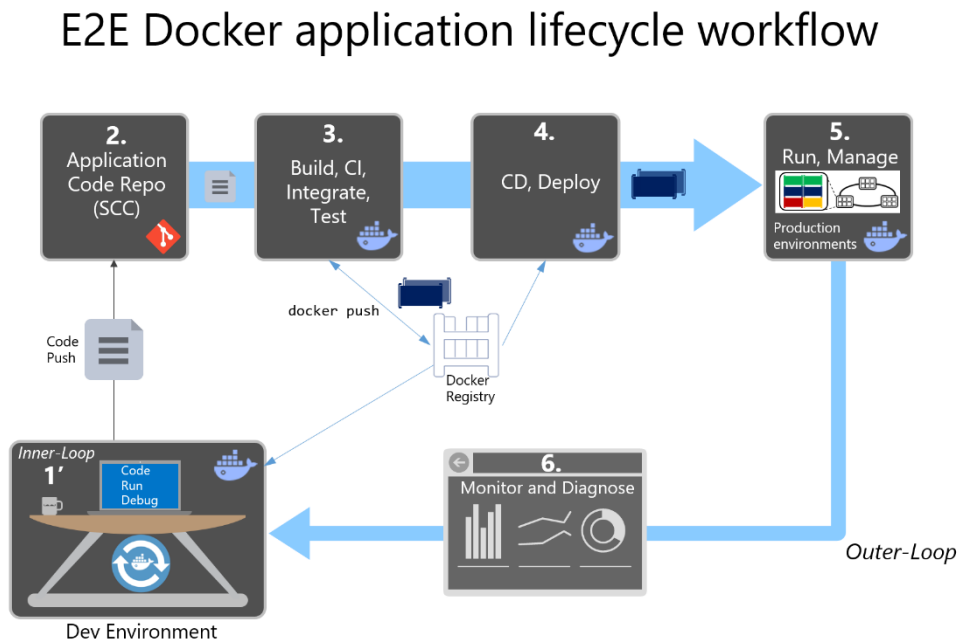


Figure 3-2. High level workflow for the Docker containerized application lifecycle

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

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

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

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

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

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

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

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

Benefits from DevOps for containerized applications

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

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

Introduction to the Microsoft platform and tools for containerized applications

Vision

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

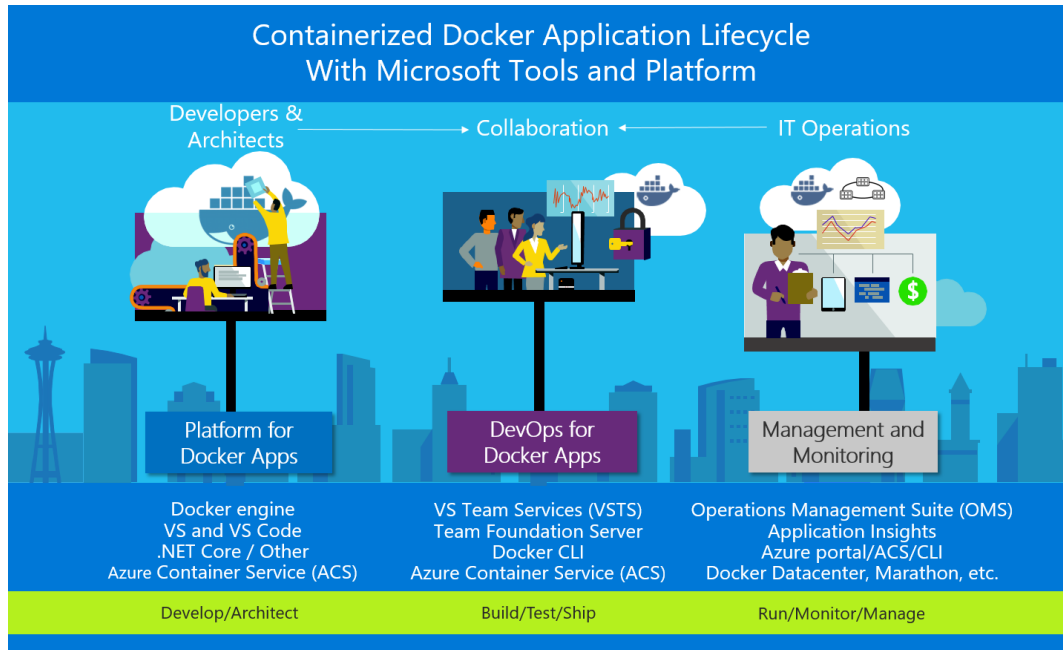


Figure 4-1. Main pillars in lifecycle for Containerized Docker Applications with Microsoft Platform & Tools

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

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

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

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

Figure 4-2. Open DevOps workflow to any technology

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

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

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

- **IT production management and monitoring.**

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

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

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

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

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

Architecting and developing containerized applications with Docker and Microsoft Azure

Vision

Architect and design scalable solutions with Docker in mind.

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

Architecting Docker applications

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

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

Common container design principles

Container equals a process

In the container model, a container represents a single process. By defining a container as a process boundary, you start to create the primitives used to scale, or batch off processes. When running a

Docker container, you'll see an [ENTRYPOINT](#) definition. This defines the process and the lifetime of the container. When the process completes, the container lifecycle ends. There are long running processes, like web servers and short lived processes like batch jobs, which may have been implemented as Azure [WebJobs](#). If the process fails, the container ends, and the orchestrator takes over. If the orchestrator was told to keep 5 instances running, and one fails. The orchestrator will instance another container to replace the failed process. In a batch job, the process is started with parameters. When the process completes, the work is complete.

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

Monolithic applications

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

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

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

Monolithic Containerized application

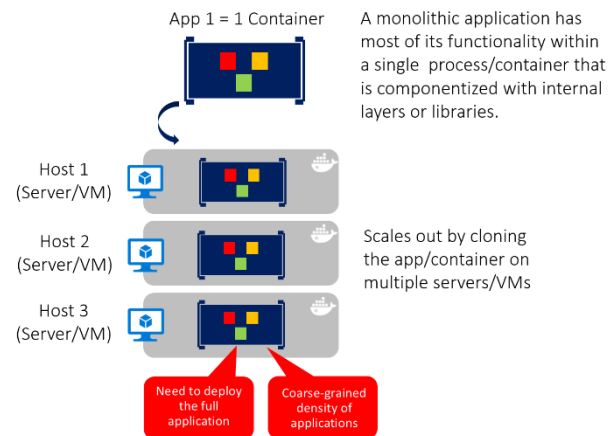


Figure 5-1. Monolithic application architecture example

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

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

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

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

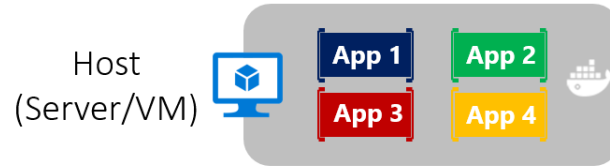


Figure 5-2. Host running multiple apps/containers

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

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

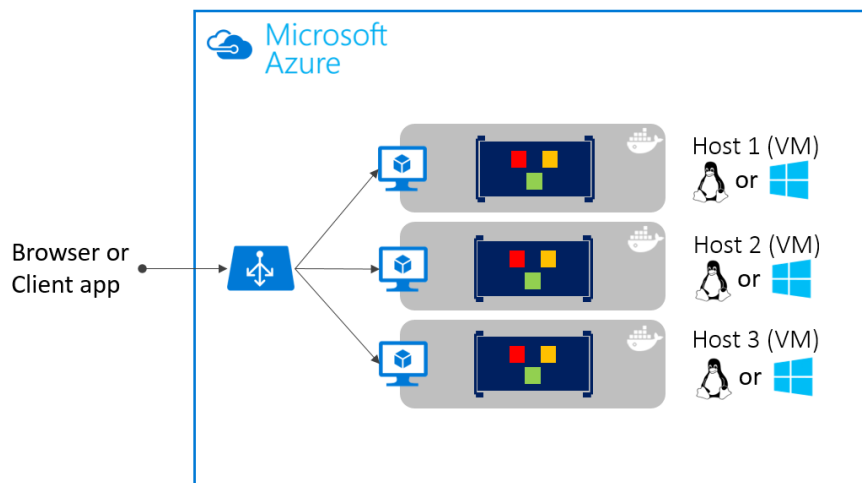


Figure 5-3. Multiple hosts scaling-out a single Docker application

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

Monolithic application deployed as a container

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

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

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

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

Publishing a single Docker container app to Azure App Service

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

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

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

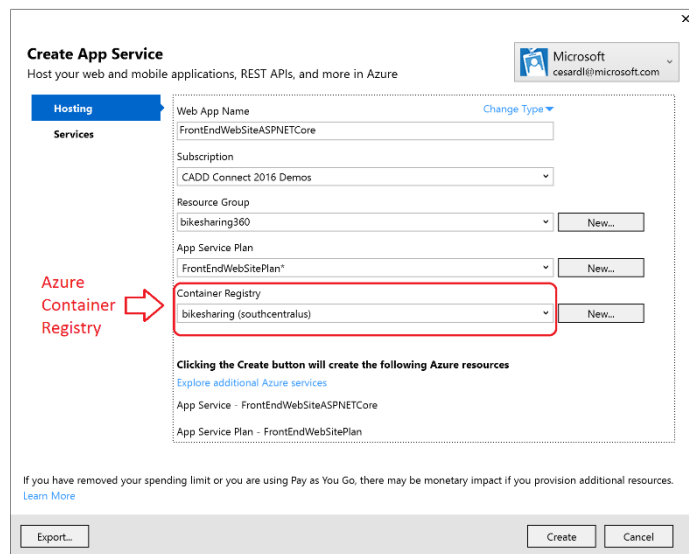


Figure 5-4. Publishing a Container to Azure App Service from Visual Studio

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

State and data in Docker applications

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

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

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

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

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

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

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

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

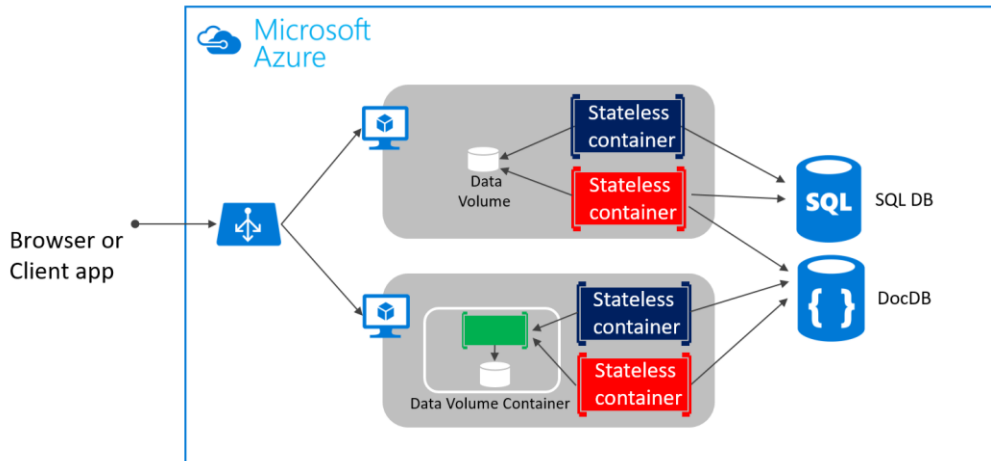


Figure 5-5. Data Volumes and external data sources for containers apps/containers

Due to the inability to manage data shared between containers that run on separate physical hosts, it is not recommended to use volumes for business data unless the Docker host is a fixed host/VM, because when using Docker containers in an orchestrator, containers are expected to be moved from one to another host depending on the optimizations to be performed by the cluster.

Therefore, regular data volumes are a good mechanism to work with trace files, temporal files or any similar concept that won't impact the business data consistency if/when your containers are moved across multiple hosts.

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

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

Service-oriented architecture applications

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

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

At the end of the day, the container clustering solutions are useful for both, a traditional SOA architecture or for a more advanced microservices architecture where each microservice owns its data model and thanks to multiple databases you can also scale-out the data tier instead of working with

monolithic databases shared by the SOA services. However, that discussion about splitting the data is purely about architecture and design. As mentioned, Docker containers are useful for both, traditional SOA architectures and the more advanced microservices architectures.

Microservices, multiple services per app and service orientation approaches

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

An architecture for composed and microservices approaches architecture would be similar to the diagram in Figure 5-6.

Composed Docker Applications in a Cluster

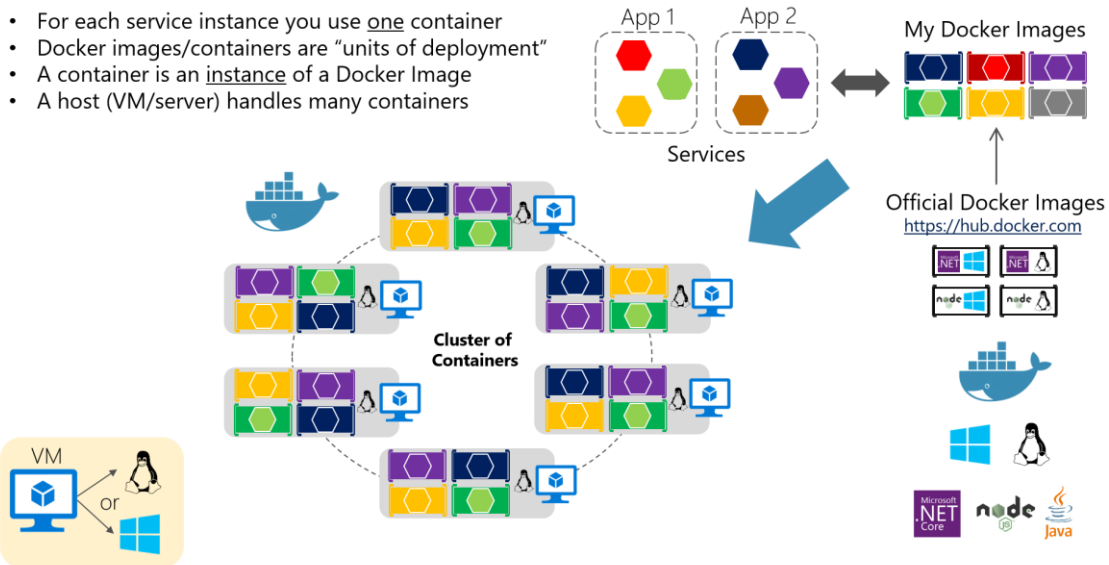


Figure 5-6. Cluster of containers

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**, etc. available as part of Microsoft **Azure Container Service** or Microsoft's container orchestrator **Azure Service Fabric**. In addition to that, in the last chapter of this document called "*Running, Managing and Monitoring production environments of Docker applications*" you can find extended information about the management tools for those clusters (IT Operations tools).

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	
 <p>Docker Swarm</p>	<p>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.</p> <p>Docker Swarm is a product created by Docker itself.</p> <p>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)</p>
 <p>Mesosphere DC/OS</p>	<p>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.</p>



	<p>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.</p>
<p>Google Kubernetes</p> 	<p>Kubernetes spans cluster infrastructure plus containers scheduling and orchestrating capabilities. It is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.</p> <p>It groups containers that make up an application into logical units for easy management and discovery.</p>
<p>Azure Service Fabric</p> 	<p>Service Fabric is a Microsoft's microservices platform for building applications. It is an orchestrator of services and creates clusters of machines. By default, Service Fabric deploys and activates services as processes but Service Fabric can deploy services in Docker container images and more importantly you can mix both services in processes and services in containers together in the same application.</p> <p>This feature (Service Fabric deploying services as Docker containers) is in preview for Linux and will be in preview for Windows Server 2016 in the upcoming release</p> <p>Service Fabric services can be developed in many ways from using the Service Fabric programming models to deploying guest executables as well as containers. Service Fabric supports prescriptive application models like Stateful services and Reliable Actors.</p>

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

Docker clusters in Microsoft Azure

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

Microsoft Azure provides Docker cluster and orchestrator support through **Azure Container Service (ACS)** as explained in the next section.

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

Azure Container Service

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

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

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

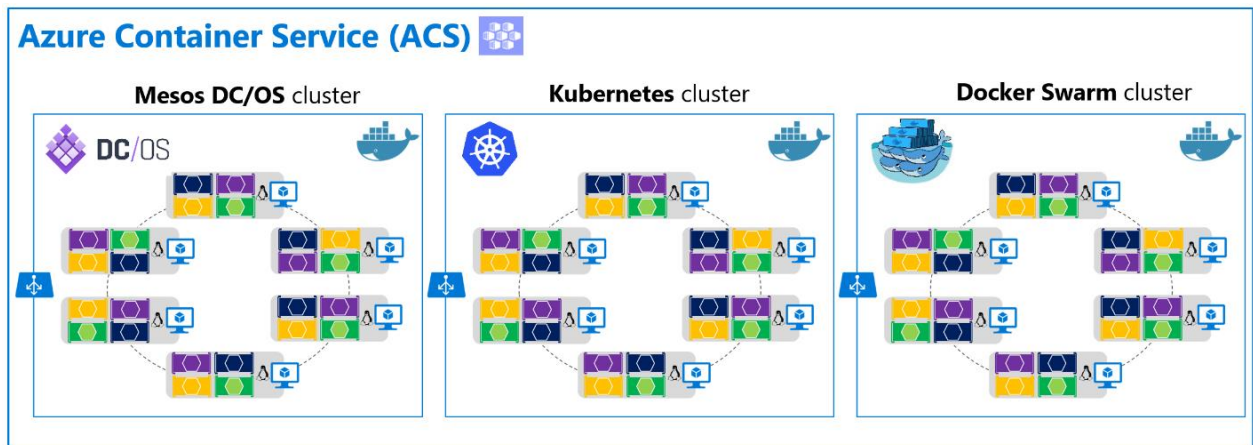


Figure 5-8. Clustering choices in ACS

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

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

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

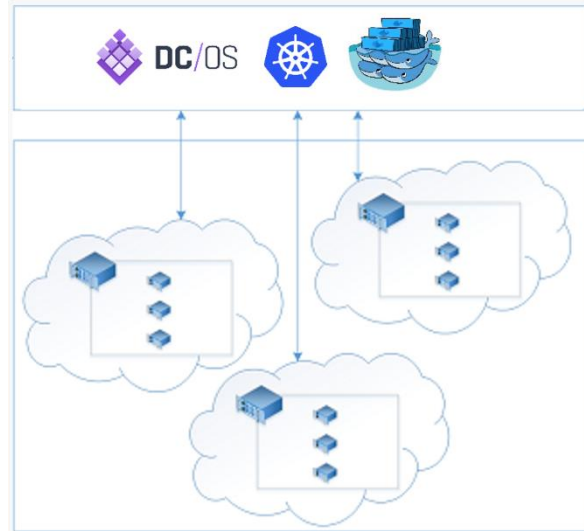


Figure 5-9. Orchestrators in ACS

Getting started with Azure Container Service

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

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

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

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

Development environment for Docker apps

Development tools choices: IDE or editor

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

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

[Download Visual Studio Code](#)

[Download Docker for Mac and Windows](#)

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

[Download Docker Tools for Visual Studio](#)

[Download Docker for Mac and Windows](#)

Language and framework choices

You can develop Docker applications and Microsoft tools with most modern languages. The following is an initial list, but you are not limited to it.

- .NET Core and ASP.NET Core
- Node.js
- Go Lang
- Java
- Ruby
- Python

Basically, you can use any modern language supported by Docker in Linux or Windows.

Inner-loop development workflow for Docker apps

Before triggering the outer-loop workflow spanning the whole DevOps cycle, it all starts from each developer's machine working coding the app itself, using his preferred languages/platforms, and testing it locally. But in every case, you will have a with a very important point in common no matter what language/framework/platforms you choose. In this specific workflow you are always developing and testing Docker containers, but locally.

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

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

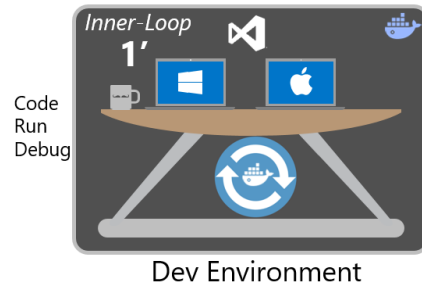


Figure 5-10. Inner-loop development context

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

Workflow for building a single app inside a Docker container using Visual Studio Code and Docker CLI

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

The following steps are the basic steps usually needed when building a Docker app, as illustrated in Figure 5-11 which is a "double-click" from Figure 5-10.

Inner-Loop development workflow for Docker apps

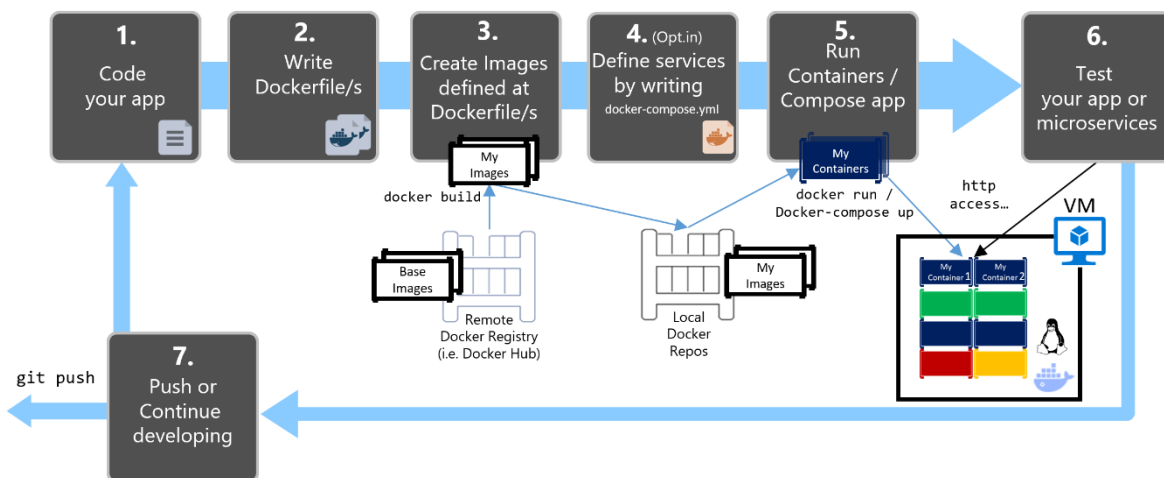


Figure 5-11. High level workflow for the lifecycle for Docker containerized applications using Docker CLI



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

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

Setup of your local environment

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

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

Installing Docker for Mac: <https://docs.docker.com/docker-for-mac/>

In addition, you'll need any code editor so you can actually develop your application while using Docker CLI.

Microsoft provides **Visual Studio Code** which is a light code editor that is cross-Platform supported on Mac, Windows and Linux and provides *intellisense* with [support for many languages](#) (JS, .NET, Go, Java, Ruby, Python and most modern languages), [debugging](#), [integration with Git](#) and [extensions-support](#). This editor is a great fit for Mac and Linux developers. In Windows you can also use the full Visual Studio.

Installing Visual Studio for Windows, Mac or Linux:

<http://code.visualstudio.com/docs/setup/setup-overview/https://docs.docker.com/docker-for-mac/>

Working with Docker and Visual Studio Code

You can work with Docker CLI and write your code with any code editor, but if using Visual Studio Code, it makes it easy to author `dockerfile` and `docker-compose.yml` files in your workspace plus you can run VS Code tasks from the IDE that will trigger scripts that can be running elaborated operations using Docker CLI underneath.

Install the Docker extension. Docker support for VS Code is provided by an extension. To install the Docker extension, Press `Ctrl+Shift+P`, type "ext install" and run the Extensions: Install Extension command to bring up the Marketplace extension list. Now type "docker" to filter the results and select the Dockerfile and Docker Compose File (yaml) Support extension.

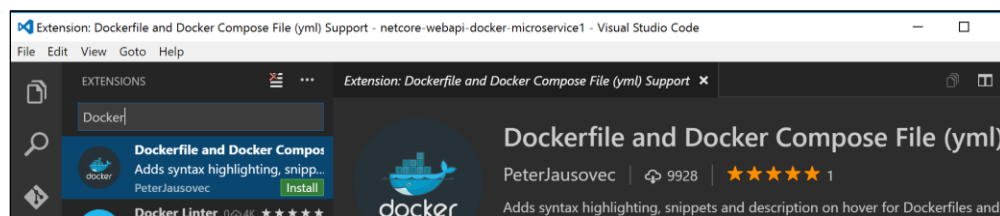


Figure 5-12. Installing Docker Extension in VS Code

2.
Write
Dockerfile/s

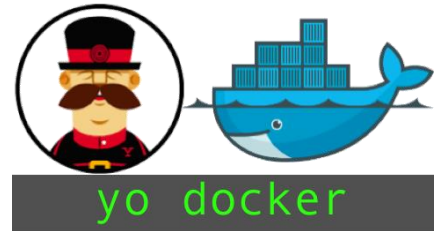


Step 2. Create Dockerfile related to an existing image (Plain OS or dev environments like .NET Core, Node.js, Ruby, etc.)

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

The Dockerfile is usually placed within the root folder of your app/service and contains the required commands so Docker knows how to setup up and run your app/service. You can create your Dockerfile and add it to your project along with your code (node.js, .NET Core, etc.), or if you are new to the environment, you can use the following tip.

Tip: You can use a command line tool called [yo-docker](#) which scaffolds files from your project in the language you choose and adds the required Docker configuration files. Basically, to assist developers getting started, it creates the appropriate dockerfile, docker-compose.yml and other associated scripts to build and run your Docker containers. This yeoman generator will prompt you with a few questions, asking your selected development language and target container host.



For instance, in Figure 5-13 you can see two screenshots from the terminals in a Mac and in Windows, in both cases running "Yo Docker" which will generate the sample code projects (currently **.NET Core**, **Golang** and **Node.js** as supported languages) already configured to work on top of Docker.

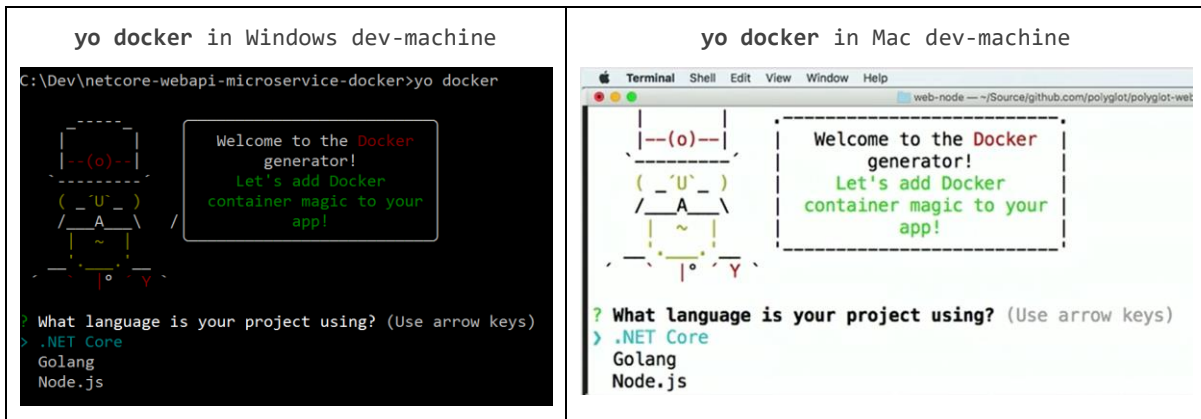


Figure 5-13. Yo docker command tool in Windows and Mac

And the following screenshot is an example using Yo Docker once you have an existing .NET Core project in place to be scaffolded.

```
C:\Dev\dockercomposedapp\netcorewebapimicroservice1>yo docker

Welcome to the Docker generator!
Let's add Docker container magic to your app!

? What language is your project using? .NET Core
? Which version of .NET Core is your project using? rtm
? Which port is your app listening to? 5000
? What do you want to name your image? cesardl/netcorewebapimicroservice1
? What do you want to name your service? netcorewebapimicroservice1
? What do you want to name your compose project? dockercomposedapp
```

Figure 5-14. Yo Docker screenshot example

From the Dockerfile you specify what base Docker image you'll be using (like using "FROM microsoft/dotnet:1.0.0-core"). You usually will build your custom image on top of a base-image you can get from any official repository at the [Docker Hub registry](#) (like an [image for .NET Core](#) or [image for Node.js](#)).

Option A – Use an existing official Docker image

Using an official repository of a language stack 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 a .NET Core container would be the following:

```
# Base Docker image to use
FROM microsoft/aspnetcore:1.0.1

# Set the Working Directory and files to be copied to the image
ARG source
WORKDIR /app
COPY ${source:-bin/Release/PublishOutput} .

# Configure the listening port to 80 (Internal/Secured port within Docker host)
EXPOSE 80

# Application entry point
ENTRYPOINT ["dotnet", "MyCustomMicroservice.dll"]
```

Figure 5-15. Sample Dockerfile for a .NET Core container

In this case, it is using the version 1.0.1 of the official ASP.NET Core Docker image for Linux named "microsoft/aspnetcore:1.0.1". For further details, consult the [ASP.NET Core Docker Image page](#) and the [.NET Core Docker Image page](#). You could also be using another comparable images like "node:4-onbuild" for **Node.js**, or many other pre-cooked images for development languages available at [Docker Hub](#).

In the Dockerfile, you also need to instruct Docker to listen to the TCP port you will use at runtime (like port 80).

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\", \"MyCustomMicroservice.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 is a new feature coming in Docker that allows vendors to use a single repo to cover multiple platforms. Such as [microsoft/aspdotnetcore](#) repository available at DockerHub registry. As the feature comes alive, pulling this image from a Windows host will pull the Windows variant. While pulling the same image name from a Linux host will pull the Linux variant.

Option B - Create your base-image from scratch

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



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

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

Note that when taking into account the "outer-loop DevOps workflow", the images will be created by an automated build process whenever you push your source code to a git repository (Continuous Integration) so the images will be created in that global environment from your source code.

But before we consider going to that outer-loop route, developers need to make sure that the Docker application is actually working properly so they don't push code to the source control system (Git, etc.) that might not work right.

Therefore, each developer needs first to do the whole inner-loop process to test locally and continue developing until they want to push a complete feature or change to the source control system.

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

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

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

Figure 5-16. Code example – Running “docker build”

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

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

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

Figure 5-17. Viewing existing images using “docker images”



Step 4. (Optional) Define your services in docker-compose.yml when building a composed Docker app with multiple services

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

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

```
version: '2'
services:
  web:
    build: .
    ports:
      - "81:80"
    volumes:
      - ./code
    depends_on:
      - redis
  redis:
    image: redis
```

Figure 5-18. Example “docker-compose.yml” file content

In this particular case, this file defines two services. The “web” service (your custom service) and the “redis” service (popular cache service). Each service will be deployed as a container, so we need to use a concrete Docker image for each. For that particular “web” service:

- Builds from the Dockerfile in the current directory.
- Forwards the exposed port 80 on the container to port 81 on the host machine.
- Mounts the project directory on the host to /code inside the container allowing you to modify the code without having to rebuild the image.
- Links the web service to the Redis service.

The “redis” service uses the [latest public Redis image](#) pulled from the Docker Hub registry. [Redis](#) is a very popular Cache system for server side applications.



Step 5. Build and run your Docker app

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

Option A. Run a single container/service

You can run the Docker image 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 5-19. Code example – running the Docker image using the “docker run” command

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

Option B. Compose and run a multiple-container application

In most enterprise scenarios, a Docker application will be composed by multiple services.

In this case, you can execute the command “**docker-compose up**” that will use the “docker-compose.yml” file that you might have created previously, so it deploys a composed application with all its related containers, as in the following example when running the command from your main project directory containing the docker-compose.yml file.

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

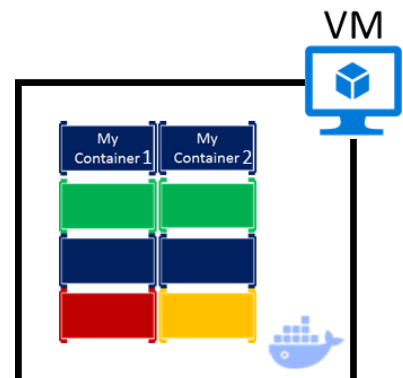


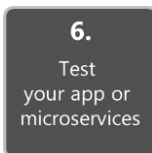
Figure 5-20. VM with Docker containers

```
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 of running the "docker-compose up" command

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

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



Step 6. Test your Docker application (locally, in your local CD VM)

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

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

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

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

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

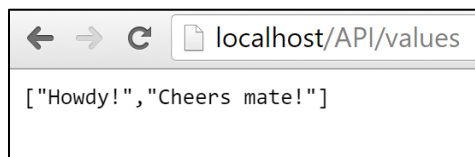


Figure 5-22. Example of testing your Docker application locally using localhost

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

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

```
PS C:\dev\netcore-webapi-microservice-docker> curl http://10.0.75.1/API/values
StatusCode      : 200
StatusDescription : OK
Content         : ["Howdy!", "Cheers mate!"]
RawContent      : HTTP/1.1 200 OK
                  Transfer-Encoding: chunked
                  Content-Type: application/json; charset=utf-8
                  Date: Thu, 14 Jul 2016 19:48:18 GMT
                  Server: Kestrel

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

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

Debugging a container running on Docker

VS Code supports Docker containers debugging if using Node.js. and other platforms like .NET Core containers.

You can also debug .NET Core containers in Docker when using Visual Studio, as described in the next section.

References for VS Code and Docker container debugging
Live Debugging Node.js Docker containers: https://blog.docker.com/2016/07/live-debugging-docker/
https://blogs.msdn.microsoft.com/user_ed/2016/02/27/visual-studio-code-new-features-13-big-debugging-updates-rich-object-hover-conditional-breakpoints-node-js-mono-more/

Using Visual Studio Tools for Docker (Visual Studio on Windows)

The developer workflow when using VS Tools for Docker is similar to the workflow when using VS Code and Docker CLI (in fact, it is based on the same Docker CLI) but it is easier to get started, simplifies the process and provides greater productivity for the build, run and compose tasks while being able to execute and debug your containers in simple actions like F5 and Ctrl+F5 from Visual Studio. Even more, with **Visual Studio 2017** in addition of being able to run and debug a single container you can also run and debug a group of containers (a whole solution) at the same time if they are defined in the same docker-compose.yml file at the solution level.

Setup of your local environment

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

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

If using **Visual Studio 2015** you need to have installed the **Update 3** or later version plus the **Visual Studio Tools for Docker**.

Install Visual Studio: <https://www.visualstudio.com/products/vs-2015-product-editions>

Installing Visual Studio Tools for Docker:

<http://aka.ms/vstoolsfordocker>

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

If using **Visual Studio 2017**, Docker support is already built-in.

Using Docker Tools in Visual Studio 2015

The Visual Studio Tools for Docker provides a consistent way to develop and validate locally your Docker containers for Linux in a Linux Docker host/VM or your Windows Containers directly on Windows.

If using a single container, the first thing you need to get started is to enable Docker support into your .NET Core project, by right-clicking on your project file, like shown in Figure 5-24.

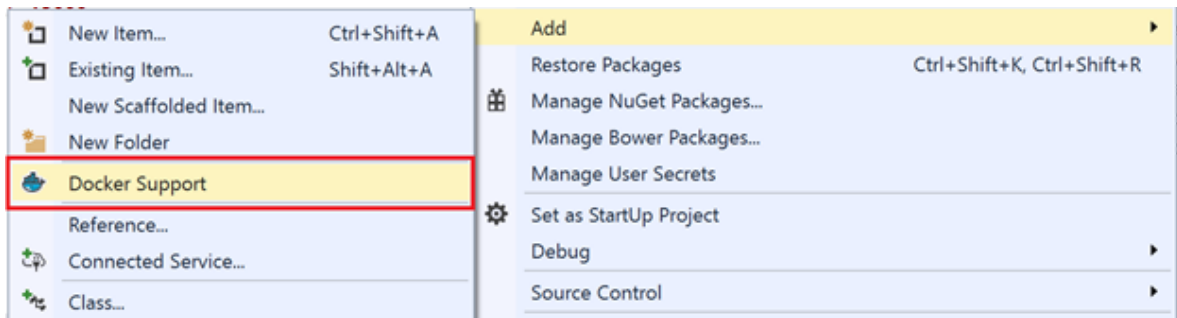


Figure 5-24. Enabling Docker support to your VS project

Using Docker Tools in Visual Studio 2017

As mentioned before, with Visual Studio 2017 you can add “Docker Solution Support” to any project which means VS not only will add the docker files to your project (as Docker Project Support does) but also will add the required configuration lines of code to a global docker-compose.yml set at the solution level.

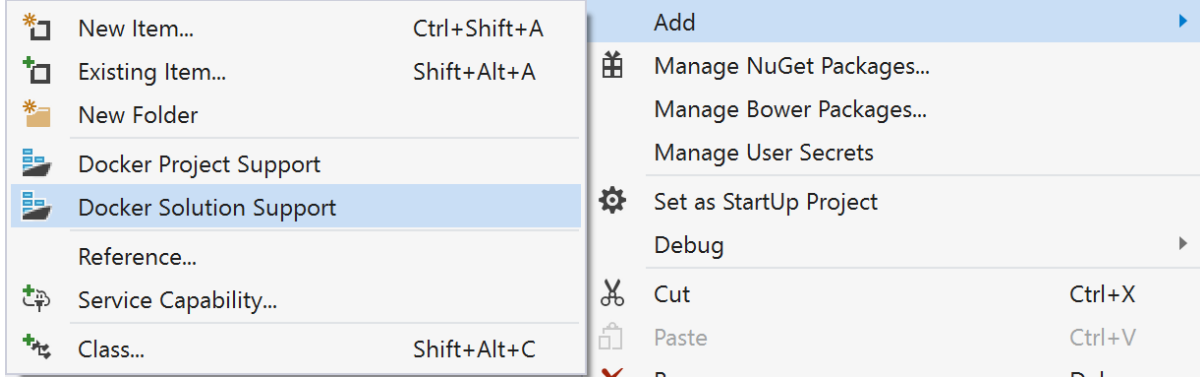


Figure 5-25. Enabling Docker Solution support in Visual Studio 2017

Each time you add “Docker Solution Support” to a specific project within a solution, you will get that project configured in the global/solution docker-compose.yml, so you will be able to run or debug the whole solution at once because Visual Studio will spin up a container per project that has Docker solution Support enabled while creating all the internal steps for you (dotnet publish, docker build to build the Docker images, etc.).

The important point here is that, as shown in figure 5-26, in **Visual Studio 2017** you have an additional F5 button that we have added so you can run or debug a whole multiple container application by running all the containers that are defined in the docker-compose.yml

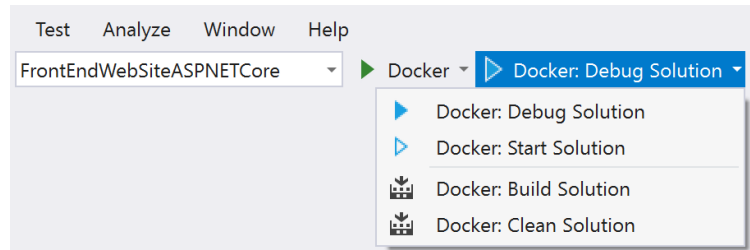


Figure 5-26. Enabling Docker Solution support in Visual Studio 2017

file at the solution level that was modified by Visual Studio while adding “Docker Solution Support” to each of your projects. This means that you could set several breakpoints up, each breakpoint in a different project/container and while debugging from Visual Studio you will be stopping in breakpoints defined in different projects and running on different containers.

For further details on the services implementation and usage of VS Tools for Docker, read the following articles.

Build, Debug, Update and Refresh apps in a local Docker container:

<https://azure.microsoft.com/en-us/documentation/articles/vs-azure-tools-docker-edit-and-refresh/>

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/>

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

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

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

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

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

In this case, it is using a Windows Server Core base image plus installing IIS with a PowerShell command.

In a similar way, you could also use PowerShell commands and setup additional components like the traditional ASP.NET 4.x and .NET 4.6 or any other Windows software. For example:

RUN powershell add-windowsfeature web-asp-net45

Docker application DevOps workflow with Microsoft tools

Visual Studio, Visual Studio Team Services, TFS and Application Insights provide a comprehensive ecosystem for development and IT operations that allow your team to manage projects and to rapidly build, test, and deploy containerized applications.

With Visual Studio and Visual Studio Team Services in the cloud, along with Team Foundation Server on-premises, development teams can productively build, test, and release containerized applications targeting any platform (Windows/Linux).

Docker DevOps lifecycle workflow with Microsoft Tools

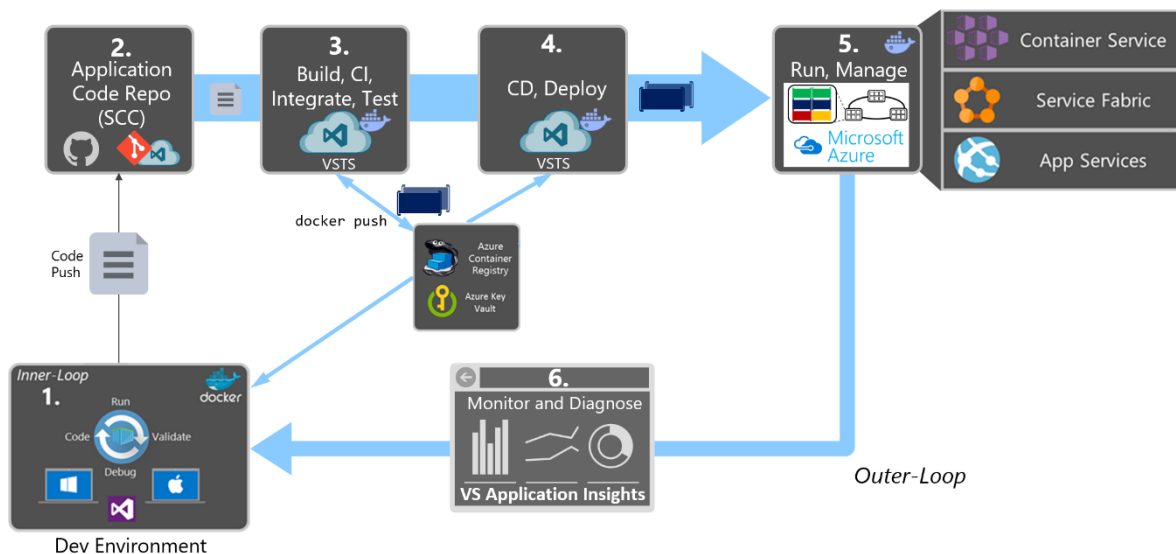


Figure 6-1. DevOps outer-loop workflow for Docker applications with Microsoft Tools

Microsoft tools can automate the pipeline for specific implementations of containerized applications (Docker, .NET Core, or any combination with other platforms) from global builds and Continuous Integration (CI) and tests with VSTS/TFS, to Continuous Deployment (CD) to Docker environments (Dev/Staging/Production), and to provide analytics information about the services back to the development team through *Application Insights*. Every code commit can trigger a build (CI) and automatically deploy the services to specific containerized environments (CD).

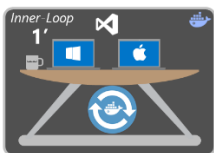
Developers and testers can easily and quickly provision production-like dev and test environments based on Docker by using templates from Azure.

The complexity of containerized application development increases steadily depending on the business complexity and scalability needs. A good example of that are applications based on Microservices architectures. To succeed in such kind of environment, your project must automate the whole lifecycle—not only build and deployment but also management of versions along with the collection of telemetry. In summary, VSTS and Azure offer the following capabilities:

- VSTS/TFS source code management (based on Git or Team Foundation Version Control), agile planning (Agile, Scrum, and CMMI are supported), continuous integration, release management, and other tools for agile teams.
- VSTS/TFS include a powerful and growing ecosystem of first- and third-party extensions that allow you to easily construct a continuous integration, build, test, delivery, and release management pipeline for microservices.
- Run automated tests as part of your build pipeline in VSTS.
- VSTS tighten the DevOps lifecycle with delivery to multiple environments, not just for production environments but also for testing, including A/B experimentation, [canary releases](#), etc.
- Docker, Azure Container Registry and Azure Resource Manager. Organizations can easily provision Docker containers from private images stored in Azure Container Registry along with any dependency on Azure components (Data, PaaS, etc.) using Azure Resource Manager (ARM) templates with tools they are already comfortable working with.

Steps in the outer-loop DevOps workflow for a Docker application

The outer-loop workflow is end-to-end represented in Figure 6-1. Now, let's drill down on each of its steps.



Step 1. Inner loop development workflow

This step was explained in the detail in the previous section, but here is where the outer-loop also starts, in the very precise moment when a developer pushes code to the source control management system (like Git) triggering Continuous Integration (CI) pipeline executions.



Step 2. SCC integration and management with Visual Studio Team Services and Git

At this step, you need to have a Version Control system to gather a consolidated version of all the code coming from the different developers in the team.

Even when SCC and source-code management might sound trivial to most developers, when developing Docker applications in a DevOps lifecycle, it is critical to highlight that the Docker images with the application must not be submitted directly to the global Docker Registry (like Azure Container Registry or Docker Hub) from the developer's machine. On the contrary, the Docker images to be released and deployed to production environments have to be created based just on the source code that is being integrated in your global build/CI pipeline based on your source-code repository (like Git).

The local images generated by the developers themselves should be used just by the developer when testing within his own machine. This is why is critical to have the DevOps pipeline triggered from the SCC code.

Microsoft VSTS and TFS support Git and Team Foundation Version Control. You can choose between them and use it for an end-to-end Microsoft experience. However, you can also manage your code in external repositories (like GitHub, on-premises Git repos or Subversion) and still being able to connect to it and get the code as the starting point for your DevOps CI pipeline.



Step 3. Build, CI, Integrate and Test with VSTS and Docker

Continuous Integration (CI) has emerged as a standard for modern software testing and delivery. The Docker solution maintains a clear separation of concerns between the development and operations teams. The immutability of Docker images ensures a repeatable deployment with what's developed, tested through CI, and run in production. Docker Engine deployed across the developer laptops and test infrastructure allows the containers to be portable across environments.

At this point, once you have a Version Control system with the right code submitted, you need a *build service* to pick up the code and run the global build and tests.

The internal workflow for this step (CI, build, Test) is about the construction of a CI pipeline consisting of your code repository (Git, etc.), your build server (VSTS), Docker Engine and a Docker Registry.

Visual Studio Team Services can be used as the foundation for building your applications and setting your CI pipeline and for publishing the built "artifacts" to an "artifacts repository" as it will be explained in the next step.

When using Docker for the deployment, the "final artifacts" to be deployed are Docker images with your application/services embedded within it. Those images will be pushed or published to a *Docker Registry* (private repo like the ones you can have in *Azure Container Registry*, or public like *Docker Hub Registry* which is commonly used for official base images).

This is the basic bottom line: The CI pipeline will be kicked off by a commit to a source-code-control repository like Git. The commit will cause VSTS to run a build job inside a Docker container, and, upon successful completion of that job, push a Docker image up to the Docker Registry.

As shown in Figure 6-2, the basic **CI workflow steps with Docker and VSTS** are:

1. Developer pushes a commit to a SCC repo (Git/VSTS, GitHub, etc.)
2. If using VSTS/Git, CI integration is integrated then it is as simple as enabling a check-box in VSTS. If using and external SCC (like GitHub) it uses a *webhook* to notify VSTS of the update/push to Git/GitHub.
3. VSTS pulls the SCC repository, including the Dockerfile describing the image, as well as the application and test code.
4. VSTS builds a Docker image and labels it with a build number.
5. VSTS instantiates the Docker container within the provisioned Docker Host, and executes the appropriate tests.

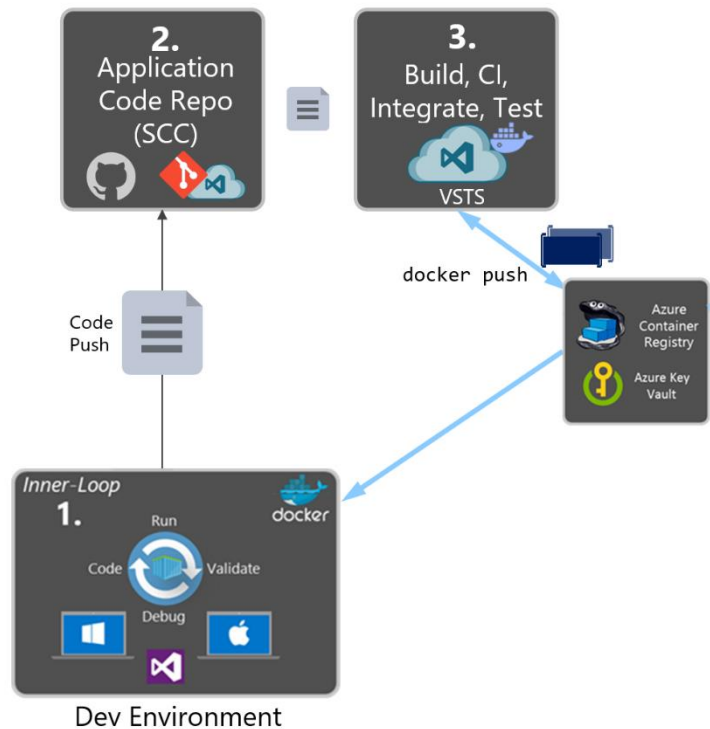


Figure 6-2. Continuous Integration steps

6. If the tests are successful the image is first re-labeled to a meaningful label so you know it was a "blessed build" (like "/1.0.0" or any other label), then pushed up to your Docker Registry (Docker Hub, Azure Container Registry, DTR, etc.)

Implementing the CI pipeline with VSTS and the Docker Extension for VSTS

The [VSTS Docker extension](#) adds a task to your CI pipeline that enables you to build Docker images, push Docker images to an authenticated Docker registry, run Docker images or execute other operations offered by the Docker CLI. It also adds a Docker Compose task that enables you to build, push and run multi-container Docker applications or execute other operations offered by the Docker Compose CLI.

The Docker extension can use service endpoints for Docker hosts and for container/image registries. The tasks default to using a local Docker host if available (this currently requires a custom VSTS agent), otherwise they require a Docker host connection to be provided. Actions that depend on being authenticated with a Docker registry, such as pushing an image, require a Docker registry connection to be provided.

The VSTS Docker extension installs the following components in your VSTS account:

- A service endpoint for connecting to Docker Registry
- A service endpoint for connecting to Docker Container Host

- A Docker task to:
 - Build an image
 - Push an image or a repository to a Registry
 - Run an image in a Container
 - Run a Docker command
- A Docker Compose Task to run a Docker Compose command

With these VSTS tasks, a build Linux-Docker Host/VM provisioned in Azure and your preferred Docker Registry (Azure Container Registry, Docker Hub, private Docker DTR or any other Docker Registry) you can assemble your Docker CI pipeline in a very consistent way.

Requirements

- Visual Studio Team Services is required, or for on-premises installations, Team Foundation Server 2015 Update 3 or later.
- A VSTS agent that has the Docker binaries. An easy way to create one of these is to use Docker to run a container based on the VSTS agent Docker image. See references below for more information.

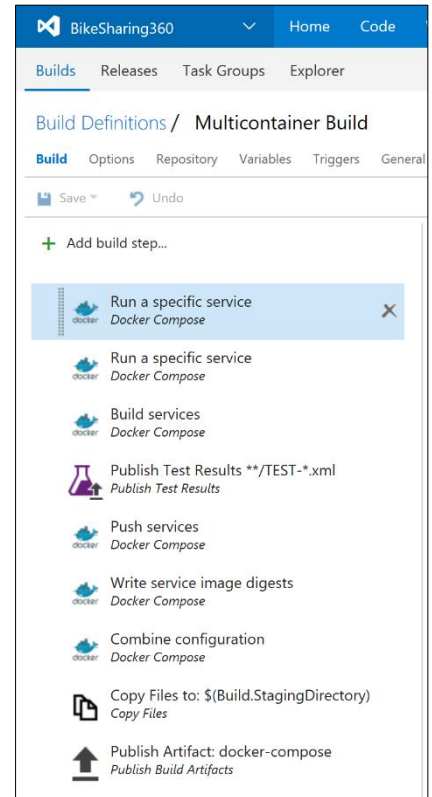


Figure 6-3. Sample Docker CI pipeline in VSTS

References and Walkthroughs for assembling a VSTS Docker CI pipeline
Running a VSTS agent as a Docker container https://hub.docker.com/r/microsoft/vsts-agent/
VSTS Docker extension https://aka.ms/vstsdockerextension
Building .NET Core Linux Docker Images with Visual Studio Team Services https://blogs.msdn.microsoft.com/stevelasker/2016/06/13/building-net-core-linux-docker-images-with-visual-studio-team-services/
Building a Linux-Based Visual Studio Team Service Build Machine with Docker Support http://donovanbrown.com/post/2016/06/03/Building-a-Linux-Based-Visual-Studio-Team-Service-Build-Machine-with-Docker-Support

Integrate, test and validate multi-container Docker applications

Typically, most Docker applications are composed by multiple containers rather than a single container. A good example is a microservice oriented application where you will have one container per microservice, but even without strictly following the microservices approach patterns, it is very probable that your Docker application is composed by multiple containers/services.

Therefore, after building the application containers in the CI pipeline, you also need to deploy, integrate and test the application as a whole with all its containers within an integration/validation Docker host or even into a test-cluster where your containers are distributed.

If using a single host, you can use Docker commands like “**docker-compose**” to build and deploy related containers to the test/validation Docker environment/VM. But if you are targeting a cluster/scheduler like Mesos, Kubernetes or Docker Swarm then you need to deploy your containers through a different mechanism or orchestrator depending on your selected cluster/scheduler.

The following are several types of tests that can be done against Docker containers:

- Unit tests for Docker containers
- Testing groups of interrelated applications/microservices
- Test in prod and canary releases

The important point is that when running integration and functional tests you have to run those tests from outside of the containers. Tests must not be defined and run within the containers you are deploying as the containers are based on static images which should be exactly like what you will be deploying into production.

A very feasible option when testing more advanced scenarios like testing several clusters (test cluster, staging cluster and production cluster) is to publish the images to a registry to test in various clusters.

Push the custom application Docker image into your global Docker Registry

Once the Docker images have been tested and validated, they tagged and published to your Docker Registry. The Docker Registry is a critical piece in Docker application lifecycle as it is the central place where you store your custom test or “blessed images” to be deployed into QA and production environments.

In the same way where your “source of truth” in regards your application code is what you have stored in your SCC repository (Git, etc.), the Docker Registry is your “source of truth” in regards your binary application or bits to be deployed to the QA/Production environments.

Typically, you might want to have your private repositories for your custom images either in a private repository in Azure Container Registry, or in an on-premises registry like Docker Trusted Registry (DTR) or in a public-cloud registry with restricted access (like Docker Hub) although in this last case if your code is not open source you must trust the vendor’s security. Either way, the way you do it is pretty similar and ultimately based on the “docker push” command.

There are multiple offerings of Docker registries from cloud vendors like Azure Container Registry, *Amazon AWS Container Registry*, *Google Container Registry*, *Quay Registry*, etc.

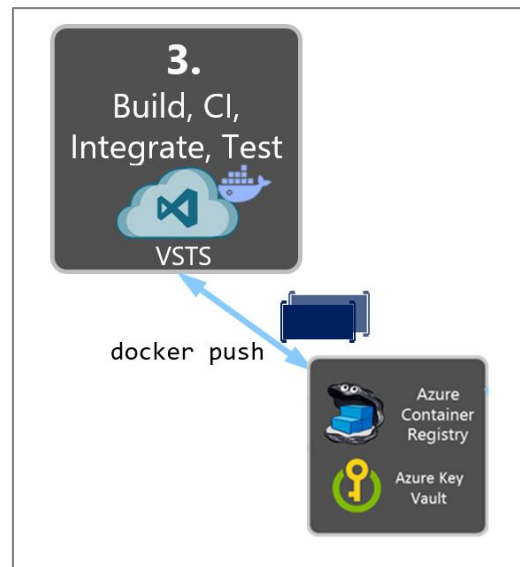


Figure 6-4. Publishing custom images to Docker Registry

The VSTS Docker extension allows you to push a set of service images defined by a docker-compose.yml file, with multiple tags, to an authenticated Docker Registry (like Azure Container Registry), as shown in the image 6-5.

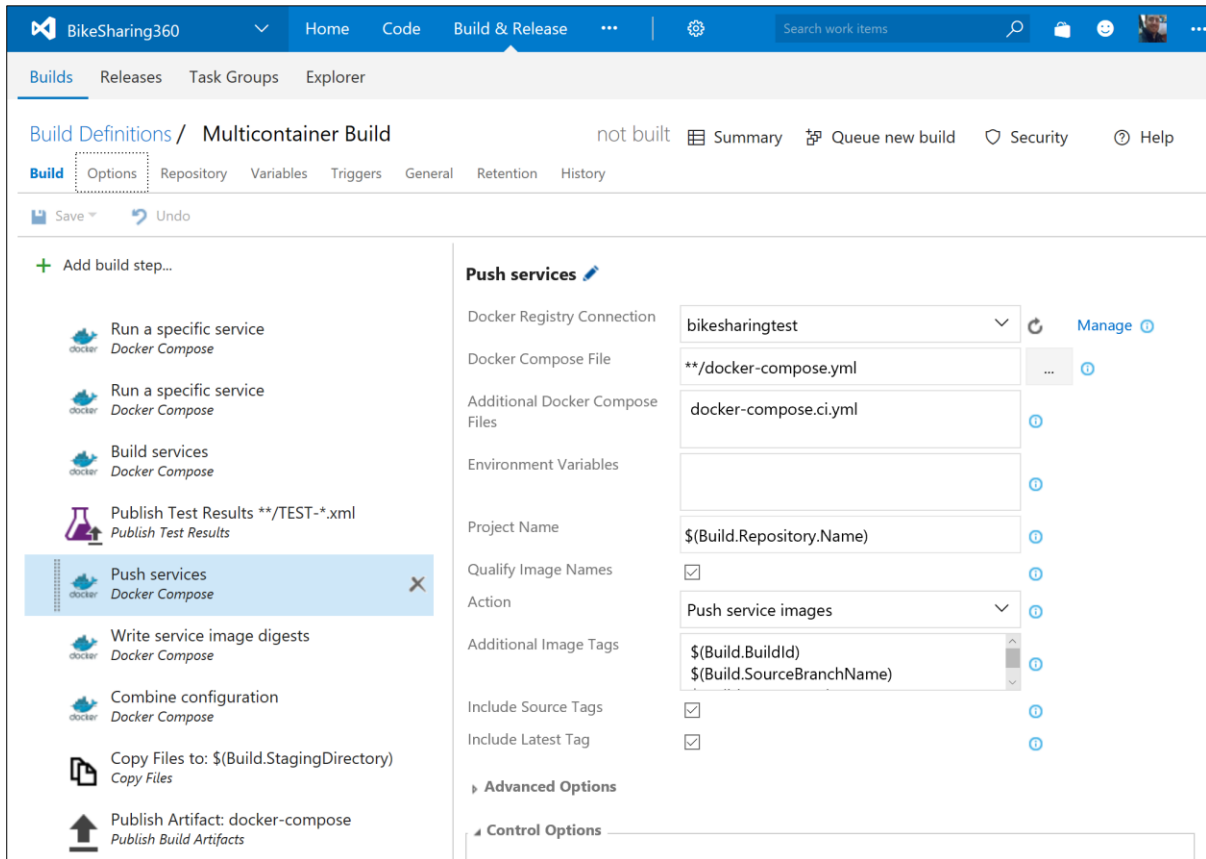


Figure 6-5. Using VSTS to publishing custom images to a Docker Registry

References on CI pipeline with VSTS and Docker

Docker Extension for VSTS:

<https://aka.ms/vstsdockerextension>

Azure Container Registry

<https://aka.ms/azurecontainerregistry>



Step 4. Continuous Delivery (CD), Deploy

The immutability of Docker images ensures a repeatable deployment with what's developed, tested through CI, and run in production. Once you have the application Docker images published in our Docker Registry (either private or public) you can deploy them to the several environments you might have (production, QA, staging, etc.) from your CD pipeline, by using VSTS pipeline tasks and/or VSTS Release Management.

However, at this point it depends of what kind of Docker application you are deploying. Deploying a simple application (from a composition and deployment point of view) like a monolithic application composed by few containers/services and deployed to a few servers/VMs is very different than deploying a more complex application like a microservices oriented application with hyper-scale capabilities. These two scenarios are explained in the following sections.

CD deploying composed Docker applications to multiple Docker environments

Going for the simpler scenario and deploying to simple Docker hosts (VMs or servers) in a single or multiple environments (QA, staging, production), your CD pipeline can internally be using `docker-compose` (from your VSTS deployment Tasks) to deploy the Docker applications with its related set of container/services as illustrated in Figure 6-6.

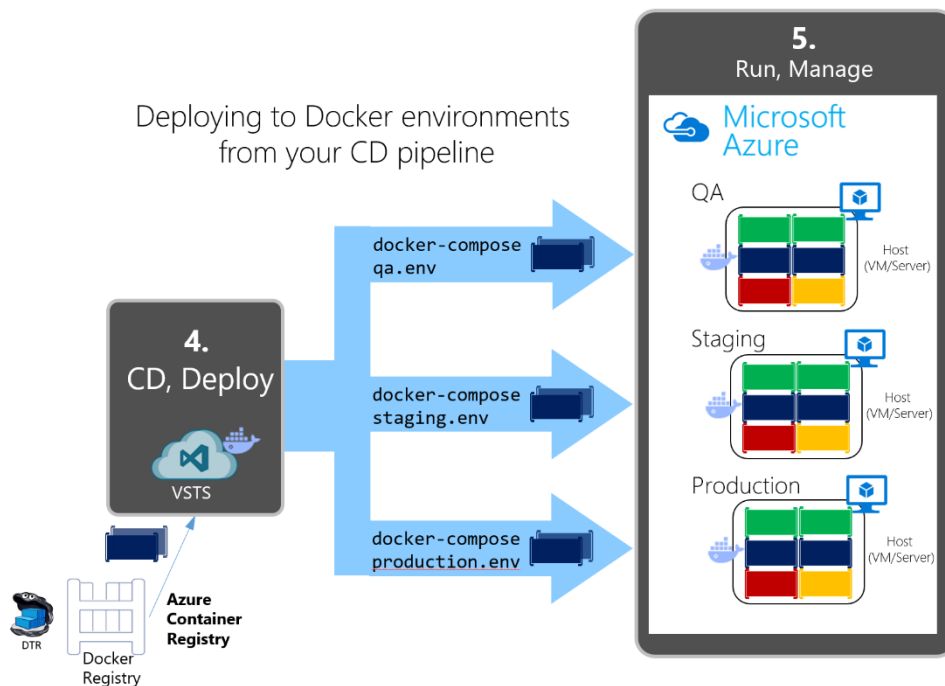


Figure 6-6. Deploying application containers to simple Docker host environments

As highlighted in Figure 6-7 it is pretty easy to connect your build CI to QA/Tests environments by deploying through the VSTS Task “Docker compose”. However, when deploying to staging/production environments, you would usually use *Release Management* features handling multiple environments (like QA, staging and production) but internally it will be using the VSTS “Docker Compose” task (which is invoking the `docker-compose up` command under the covers) if deploying to single Docker hosts, or the Docker Deployment task if deploying to ACS and clusters like DC/OS, as explained in the following section.

When you create a release in VSTS, it takes a set of input artifacts. These are intended to be immutable throughout the lifetime of the release across multiple environments. When you introduce containers, the input artifacts identify images in a registry to deploy. Depending on how these are identified, they are not guaranteed to remain the same through the duration of the release, the most obvious case being when you reference “myimage:latest” from a docker-compose file.

The Docker extension for VSTS enables you to generate build artifacts that contain specific registry image digests that are guaranteed to uniquely identify the same image binary. These are what you really want to use as input to a release.

Managing releases to Docker environments with VSTS Release Management

Through the VSTS extensions, you can build a new image, publish it to a Docker registry, run it on Linux or Windows hosts, and use commands such as `docker-compose` to deploy multiple containers as a whole application, all from the VSTS Release Management capabilities targeting multiple environments, as shown in Figure 6-8.

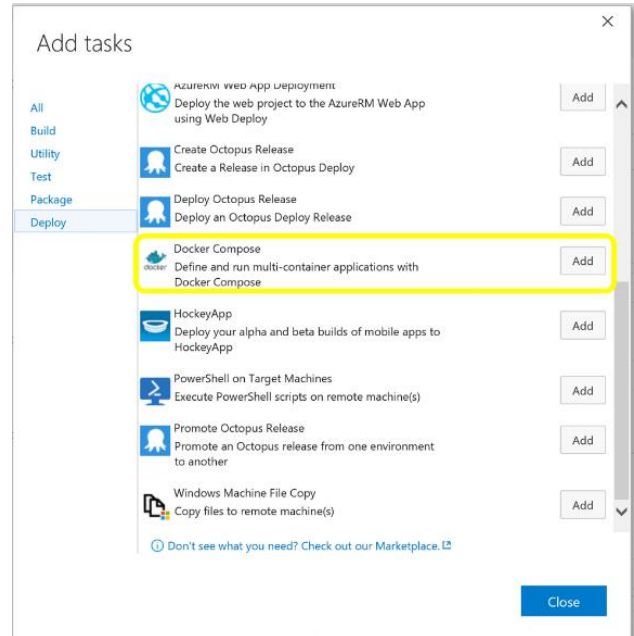


Figure 6-7. Adding a Docker Compose Task in a VSTS pipeline

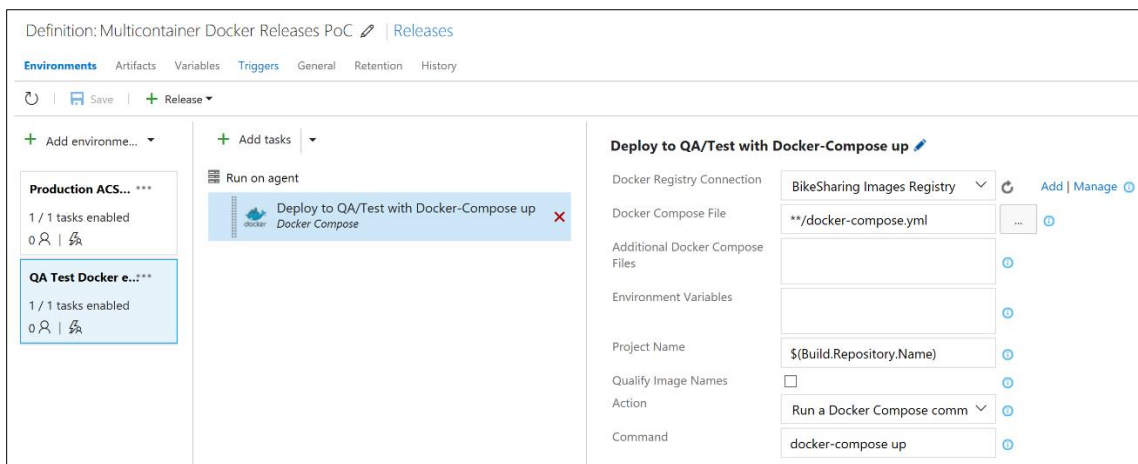


Figure 6-8. Configuring VSTS Docker Compose tasks from VSTS Release Management

However, take into account that the scenario shown in Figure 6-6 implemented in figure 6-8 is pretty basic (it is deploying to simple Docker hosts/VMs and there will be a single container/instance per image) and probably should be used only for development/test scenarios. In most enterprise production scenarios, you would want to have HA (High Availability) and easy to manage scalability by load balancing across multiple nodes/servers/VMs plus “intelligent failovers” so if a server/node fails, its services/containers will be moved to another host server/VM, etc. In that case, you need more advanced technologies like container clusters, container orchestrators and schedulers. In that case, the way to deploy to those clusters is precisely through the advanced scenarios explained in next section.

CD deploying complex Docker applications to Docker clusters (DC/OS, Kubernetes and Docker Swarm)

The nature of distributed applications requires compute resources that are also distributed. In order to have production scale capabilities you need to have clustering capabilities that provides high scalability and HA based on pooled resources.

You could deploy containers manually to those clusters from a CLI tool like Docker Swarm mode (like using “[docker service create](#)”) or a web UI like in [Mesosphere Marathon](#) for DC/OS clusters, but that way should be used just for punctual deployment testing or for management purposes like scaling-out or monitoring purposes.

From a CD (Continuous Delivery) point of view and VSTS in concrete, you can run specially made deployment tasks from your VSTS Release Management environments which will deploy your containerized applications into distributed clusters in Azure Container Service.

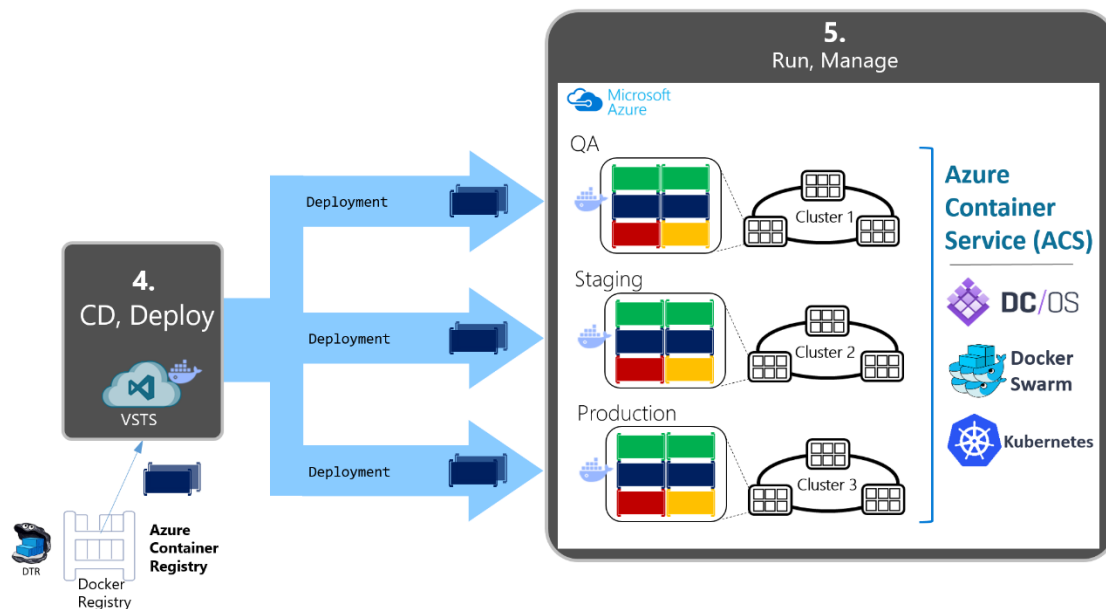


Figure 6-9. Deploying distributed applications to Azure Container Service

Initially, when deploying to certain clusters/orchestrators you would traditionally use specific deployment scripts and mechanisms per each orchestrator (i.e., Mesosphere DC/OS or Kubernetes have different deployment mechanisms than Docker and Docker Swarm) instead of the simpler and easy to use docker-compose tool based on the docker-compose.yml definition file. However, thanks to the Microsoft VSTS **Docker Deploy** task shown in figure 6-10, you can now also deploy to DC/OS

by just using your familiar docker-compose.yml file because Microsoft will be performing that “translation” for you (from your docker-compose.yml file to other formats needed by DC/OS).

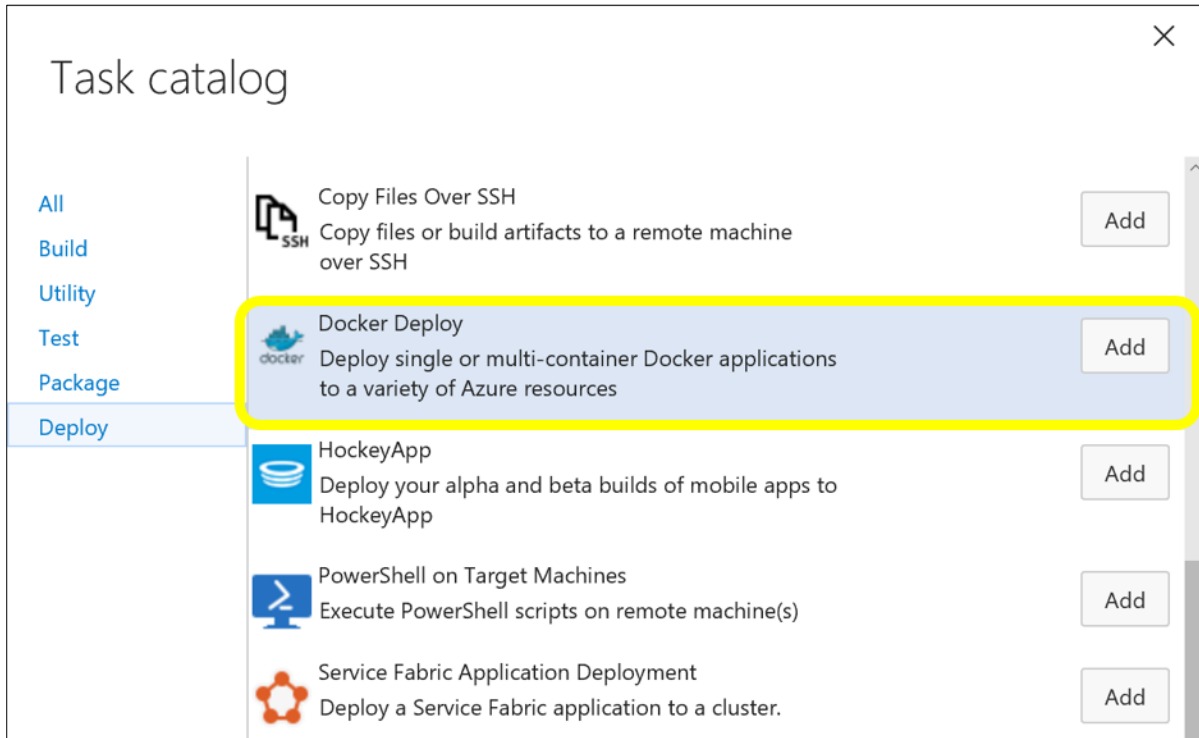


Figure 6-10. Adding Docker Deploy task to your Environment RM

In Figure 6-11 you can see how you can edit the Docker deploy task and specify the Target type (ACS DC/OS in this case), your Docker Compose File and the Docker Registry connection (like Azure Container Registry or Docker Hub) from where the task will get your ready-to-use custom Docker images to be deployed as containers in the DC/OS cluster.

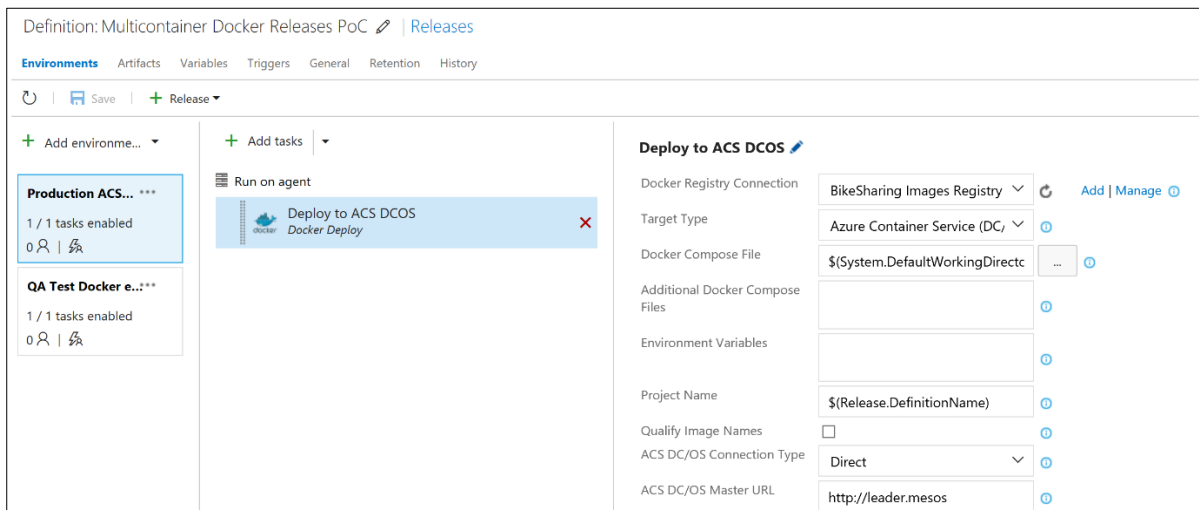


Figure 6-11. Docker Deploy task definition deploying to ACS DC/OS

References on CD pipeline with VSTS and Docker

VSTS Extension for Docker and Azure Container Service:

<https://aka.ms/vstsdockerextension>

Azure Container Service

<https://aka.ms/azurecontainerservice>

Mesosphere DC/OS

<https://mesosphere.com/product/>



Step 5. Run and manage

Since running and managing applications at enterprise production level is a major subject and due to the type of operations and people working at the level (IT Operations) as well as the large volume of this area, this step is explained in the next whole chapter.

However, even when running and managing applications in production is a different subject, this step within DevOps is intended to illustrate the fact that this is the “next step” after CD (Continuous Delivery). “Now you need to run the applications effectively and reliably”.



Step 6. Monitor and diagnose

In a similar way, this topic is covered in the next chapter as part of the tasks that IT operations performs in production systems, although it is important to highlight that the insights obtained in this step have to feed the dev team so the application is improved. From that point of view, it is also part of DevOps, although the tasks and operations are usually performed by IT.

When monitoring and Diagnosing are 100% part of DevOps is when the monitoring processes and analytics are performed by the dev team against testing or beta environments, either performing Load testing or simply by monitoring beta or QA environments where beta testers are trying the new versions.

Running, managing and monitoring Docker production environments

Vision

Enterprise applications need to run with high availability and high scalability while IT operations needs to be able to manage and monitor the environments and the applications themselves.

This last pillar in the containerized Docker applications lifecycle is centered on how you can run, manage and monitor your applications in scalable and high available production environments.

How you run your containerized applications in production (infrastructure architecture and platform technologies) is also very much related and completely founded on the chosen architecture and development platform introduced in the first chapter, however, this chapter is diving into specific products and technologies from Microsoft and other vendors that you can use to effectively run highly scalable and highly available distributed applications plus how you can manage and monitor them from the IT perspective.

Running composed and microservices-based applications in production environments

Intro to orchestrators, schedulers, and container clusters

Clusters and *schedulers* were already introduced in the initial section of this document when tackling on Software Architecture and development. Examples of Docker clusters are Docker Swarm and Mesosphere DC/OS. Both can run as part of the infrastructure provided by Microsoft Azure Container Service.

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 orchestrators and schedulers provide.

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.

Orchestration platforms extend lifecycle management capabilities to complex, multi-container workloads deployed on a cluster of hosts. By abstracting the host infrastructure, orchestration tools allow users to treat the entire cluster as a single deployment target.

The process of orchestration involves tooling and platform that can automate all aspects of application management from initial placement/deployment per container, moving containers to different hosts depending on its host's health or performance, versioning and rolling updates and health monitoring functions that support scaling and failover and many more.

Orchestration is a broad term that refers to container scheduling, cluster management, and possibly the provisioning of additional hosts.

Those capabilities provided by orchestrators and schedulers are very complex to develop/create from scratch and therefore you usually would want to make use of orchestration solutions offered by vendors.

Managing production Docker environments

Azure Container Service and management tools

Cluster management and orchestration is the process of controlling a group of hosts. This can involve adding and removing hosts from a cluster, getting information about the current state of hosts and containers, and starting and stopping processes. Cluster management and orchestration are closely tied to scheduling because the scheduler must have access to each host in the cluster in order to schedule services. For this reason, the same tool is often used for both purposes.

Azure Container Service (ACS) provides rapid deployment of popular open-source container clustering and orchestration solutions. ACS leverages Docker images to ensure that your application containers are fully portable. By using Azure Container Service, you can deploy **DC/OS** (Powered by Mesosphere and Apache Mesos) and **Docker Swarm** clusters with Azure Resource Manager templates or the Azure portal to ensure that these applications can be scaled to thousands, even tens of thousands of containers.

You deploy these clusters by using Azure Virtual Machine Scale Sets, and the clusters take advantage of Azure networking and storage offerings. To access Azure Container Service, you need an Azure subscription. 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.

In the following table you can see a list of common management tools related to their orchestrators, schedulers and clustering platform.










Management tools for Docker-based clusters		
Management Tools	Description	Related orchestrators
Azure Container Service (UI management in Azure portal) 	<p>ACS provides an easy to get started way to deploy a container-cluster in Azure based on popular orchestrators like Mesosphere DC/OS, Kubernetes and Docker Swarm.</p> <p>ACS optimizes the configuration of those platforms. You just need to select the size, the number of hosts, and choice of orchestrator tools, and Container Service handles everything else.</p>	Mesosphere DC/OS Kubernetes Docker Swarm
Docker Universal Control Plane (UCP)  (On-premises or cloud)	<p>Docker Universal Control Plane (UCP) is the enterprise-grade cluster management solution from Docker. It helps you manage your whole cluster from a single place.</p> <p>Docker UCP is include as part of the commercial product named Docker Datacenter which provides Docker Swarm, Docker UCP and Docker Trusted Registry (DTR).</p> <p>Docker Datacenter can be installed on-premises or provisioned from a public cloud like Azure.</p>	Docker Swarm  (Supported by Azure Container Service)
Docker Cloud (aka. Tutum)  (Cloud SaaS)	<p>Docker Cloud is a hosted management service (SaaS) that provides orchestration capabilities and a Docker Registry with build and testing facilities for Dockerized application images, tools to help you set up and manage your host infrastructure, and deployment features to help you automate deploying your images to your concrete infrastructure. You can connect your SaaS Docker Cloud account to your infrastructure in Azure Container Service running a Docker Swarm cluster.</p>	Docker Swarm  (Supported by Azure Container Service)
Mesosphere Marathon  MARATHON (On-premises or cloud)	<p>Marathon is a production-grade container orchestration and scheduler platform for Mesosphere's Datacenter Operating System (DC/OS) and Apache Mesos.</p> <p>It works with Mesos (DC/OS is based on Apache Mesos) to control long-running services and provides a web UI for process and container management. It provides a web UI management tool</p>	Mesosphere DC/OS (Based on Apache Mesos)  (Supported by Azure Container Service)
Google Kubernetes 	<p>Kubernetes spans orchestrating, scheduling and cluster infrastructure. It is an open-source platform for automating deployment, scaling, and operations of application containers across clusters of hosts, providing container-centric infrastructure.</p>	Google Kubernetes  (Supported by Azure Container Service)

Figure 7-1. Docker Management Tools

Azure Service Fabric

Another choice for cluster-deployment and management is Azure Service Fabric. [Azure Service Fabric](#) is a Microsoft's microservice platform and includes container orchestration as well as developer programming models to build high-scalable microservice applications. Service Fabric supports Docker

in current Linux preview versions, like in the [Azure Service Fabric preview on Linux](#), and for Windows Containers [in the next release](#).

Azure Service Fabric management tools are:

- [Azure portal for Service Fabric](#) cluster related operations (create/update/delete) a cluster or configure its infrastructure (VMs, Load Balancer, Networking, etc.)
- [Azure Service Fabric Explorer](#) is a specialized web UI tool that provides insights and certain operations on the Service Fabric cluster from the nodes/VMs point of view and from the application and services point of view.

Monitoring containerized application services

Microsoft Application Insights

Analyzing Docker apps in QA environments using Application Insights

[Application Insights](#) is an extensible analytics service that monitors your live application. It helps you detect and diagnose performance issues and to understand what users actually do with your app. It's designed for developers, to help you continuously improve the performance and usability of your services or applications. It works with both web/services and stand-alone apps on a wide variety of platforms like .NET, Java, Node and many other platforms, hosted on-premises or in the cloud.

Related to Docker, lifecycle events and performance counters from Docker containers can be charted on Application Insights. You just need to run the [Application Insights Docker image](#) as a container in

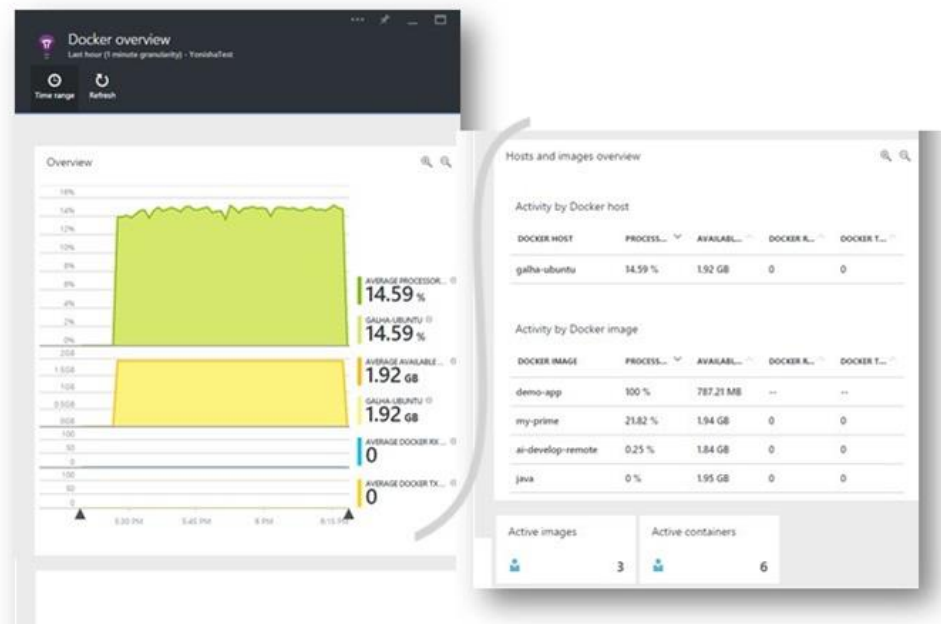


Figure 7-2. Application Insights monitoring Docker hosts and containers

your host, and it will display performance counters for the host, as well as for the other Docker images. This Application Insights Docker image helps you monitor your containerized applications by collecting telemetry about the performance and activity of your Docker host (i.e. your Linux VMs), Docker containers and the applications running within them.

When you run the [Application Insights Docker image](#) on your Docker host, you'll get these benefits:

- Lifecycle telemetry about all the containers running on the host - start, stop, and so on.
- Performance counters for all the containers. CPU, memory, network usage, and more.
- If you also installed [Application Insights SDK](#) in the apps running in the containers, all the telemetry of those apps will have additional properties identifying the container and host machine. So, for example, if you have instances of an app running in more than one host, you'll easily be able to filter your app telemetry by host.

Setting up Application Insights to monitor Docker applications and Docker hosts

Follow the instructions in the following articles to create an Application Insights resource. Azure Portal will create the necessary script for you.

Monitor Docker applications in Application Insights

<https://azure.microsoft.com/en-us/documentation/articles/app-insights-docker/>

Application Insights Docker image at Docker Hub and Github

<https://hub.docker.com/r/microsoft/applicationinsights/>

<https://github.com/Microsoft/ApplicationInsights-Docker>

Set up Application Insights for ASP.NET

<https://azure.microsoft.com/en-us/documentation/articles/app-insights-asp-net/>

Application Insights for web pages

<https://azure.microsoft.com/en-us/documentation/articles/app-insights-javascript/>

Microsoft Operations Management Suite (OMS)

[Operations Management Suite \(OMS\)](#) is a simplified IT Management Solution that provides Log Analytics, Automation, Backup and Site Recovery. Based on [queries](#) in OMS, you can raise [alerts](#) and set remediation via [Azure Automation](#). It also seamlessly integrates with your existing management solutions to provide a single pane of glass view. It helps you manage and protect your on-premises and cloud infrastructure.

OMS Container Solution for Docker

In addition to providing valuable services on its own, the **OMS Container Solution can manage and monitor Docker hosts and containers** by showing information about where your containers and container hosts are, which containers are running or failed, and Docker daemon and container logs sent to *stdout* and *stderr*. It also shows performance metrics such as CPU, memory, network and storage for the container and hosts to help you troubleshoot and find noisy neighbor containers.

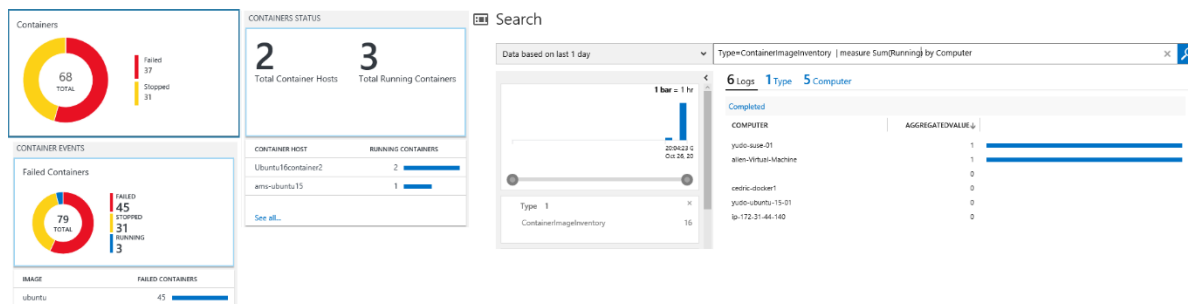


Figure 7-3. Information about Docker containers shown by OMS

Application Insights and OMS both focus on monitoring activities but Application Insights focuses more on monitoring the apps themselves thanks to its SDK running within the app. However, OMS focuses much more on the infrastructure around the hosts plus it offers deep analysis on logs at scale while providing a very flexible data-driven search/query system.

Since OMS is implemented as a cloud-based service, you can have it up and running quickly with minimal investment in infrastructure services. New features are delivered automatically, saving your ongoing maintenance and upgrade costs.

The OMS Container Solution allows you to:

- Centralize and correlate millions of logs from Docker containers at scale
- See information about all container hosts in a single location
- Know which containers are running, what image they're running, and where they're running
- Quickly diagnose "noisy neighbor" containers that can cause problems on Container hosts
- See an audit trail for actions on containers
- Troubleshoot by viewing and searching centralized logs without remoting to the Docker hosts
- Find containers that may be "noisy neighbors" and consuming excess resources on a host
- View centralized CPU, memory, storage, and network usage and performance information for containers
- Generate test Docker containers with [Azure Automation](#)

You can see performance information by running queries like "Type=Perf*" as shown in Figure 7-4.

Saving [queries](#) is also a standard feature in OMS and can help you keep queries you've found useful and discover trends in your system.

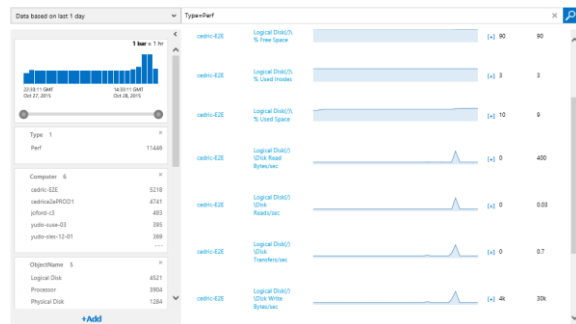


Figure 7-4. Performance of Docker hosts shown by OMS

Setting up OMS for Docker

Follow the instructions in the following article to drill down on OMS for Docker.

Getting started with OMS for Docker

Installing and configuring the Docker Container solution in OMS

<https://azure.microsoft.com/en-us/documentation/articles/log-analytics-containers/>

Conclusions

Key takeaways

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