Microsoft

Containerized Docker Application Lifecycle with Microsoft Platform and Tools

Cesar de la Torre
Microsoft Corp.

EDITION v6.0.0 - Updated to ASP.NET Core 6.0

Refer changelog for the book updates and community contributions.

This guide is a general overview for developing and deploying containerized ASP.NET Core applications with Docker, using the Microsoft platform and tools. The guide includes a high-level introduction to Azure DevOps, for implementing CI/CD pipelines, as well as Azure Container Registry (ACR), and Azure Kubernetes Services AKS for deployment.

For low-level, development-related details you can see the .NET Microservices: Architecture for Containerized .NET Applications guide and it related reference application eShopOnContainers.

Send us your feedback!

We wrote this guide to help you understand the architecture of containerized applications and microservices in .NET. The guide and related reference application will be evolving, so we welcome your feedback! If you have comments about how this guide can be improved, submit feedback at https://aka.ms/ebookfeedback.

Credits

Author:

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

Acquisitions Editor:

Janine Patrick

Developmental Editor:

Bob Russell, Solutions Professional at Microsoft

Octal Publishing, Inc.

Editorial Production:

Dianne Russell

Octal Publishing, Inc.

Copyeditor:

Bob Russell, Solutions Professional at Microsoft

Participants and reviewers:

Nish Anil, Sr. Program Manager, .NET team, Microsoft

Miguel Veloso, Software Development Engineer at Plain Concepts

Sumit Ghosh, Principal Consultant at Neudesic

Colin Dembovsky, DevOps Practice Lead, Cognizant Microsoft Business Group

Copyright

PUBLISHED BY

Microsoft Developer Division, .NET and Visual Studio product teams

A division of Microsoft Corporation

One Microsoft Way

Redmond, Washington 98052-6399

Contents

CHAPTER 1

Overview of Containers and Docker

Containerization is an approach to software development in which an application or service, its dependencies, and its configuration (abstracted as deployment manifest files) are packaged together as a container image. You then can test the containerized application as a unit and deploy it as a container image instance to the host operating system (OS).

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

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

Each container can run a whole web application or a service, as shown in Figure 1-1. In this example, Docker host is a container host, and App1, App2, Svc1, and Svc2 are containerized applications or services.

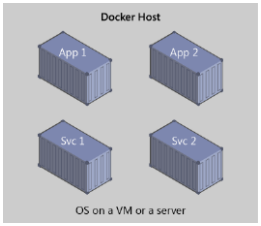

Figure 1-1. Multiple containers running on a container host

Another benefit you can derive from containerization is scalability. You can scale out quickly by creating new containers for short-term tasks. From an application point of view, instantiating an image (creating a container) is similar to instantiating a process like a service or web app. For reliability, however, when you run multiple instances of the same image across multiple host servers, you typically want each container (image instance) to run in a different host server or VM in different fault domains.

In short, containers offer the benefits of isolation, portability, agility, scalability, and control across the entire application lifecycle workflow. The most important benefit is the environment isolation provided between Dev and Ops.

Learn Docker

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

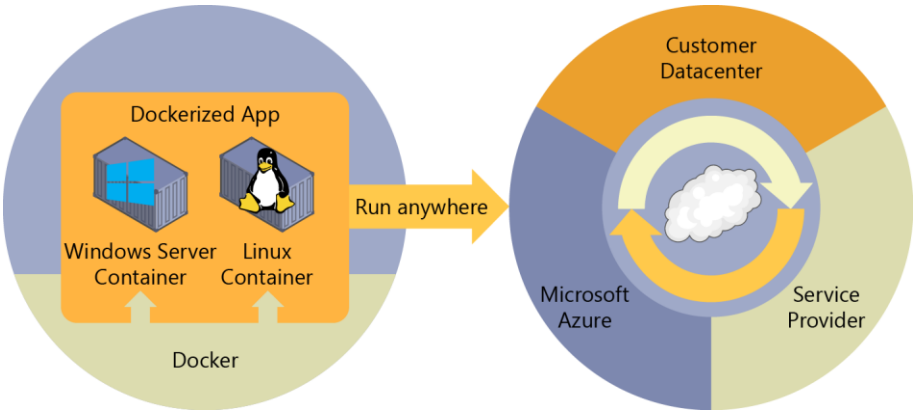

Figure 1-2. Docker deploys containers at all layers of the hybrid cloud

As shown in the above diagram, Docker containers can run anywhere, on-premises in the customer datacenter, in an external service provider or in the cloud, on Azure. Docker image containers can also run natively on Linux and Windows. However, Windows images can run only on Windows hosts and Linux images can run on Linux hosts and Windows hosts (using a Hyper-V Linux VM, so far), where host means a server or a VM.

Developers can use development environments on Windows, Linux, or macOS. On the development computer, the developer runs a Docker host where Docker images are deployed, including the app and its dependencies. Developers who work on Linux or on the Mac, use a Docker host that's Linux-based, and they can only create images for Linux containers. (Developers working on the Mac can edit code or run the Docker command-line interface (CLI) from macOS, but as of this writing, containers don't run directly on macOS.) Developers who work on Windows can create images for either Linux or Windows Containers.

To host containers in development environments and provide additional developer tools, Docker ships Docker Desktop for Windows or for macOS. These products install the necessary VM (the Docker host) to host the containers.

To run Windows Containers, there are two types of runtimes:

- **Windows Server Containers** provide application isolation through process and namespace isolation technology. A Windows Server Container shares a kernel with the container host and with all containers running on the host.
- **Hyper-V Containers** expand on the isolation provided by Windows Server Containers by running each container in a highly optimized virtual machine. In this configuration, the kernel of the container host isn't shared with the Hyper-V Containers, providing better isolation.

The images for these containers are created and work just the same way. The difference is in how the container is created from the image—running a Hyper-V Container requires an extra parameter. For details, see Hyper-V Containers.

Comparing Docker containers with virtual machines

Figure 1-3 shows a comparison between VMs and Docker containers.

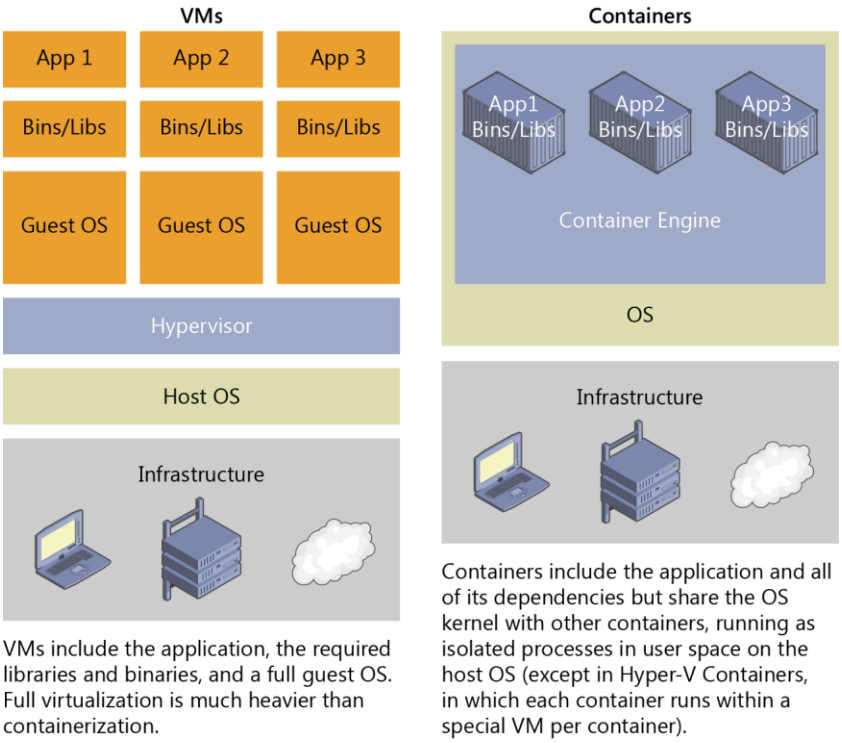

Figure 1-3. Comparison of traditional virtual machines to Docker containers

As shown in the above diagram, for VMs, there are three base layers in the host server. From the bottom-up: Infrastructure, Host Operating System, and a Hypervisor. On top of all that, each VM has its own OS and all necessary libraries. On the other hand, for Docker, the host server only has the Infrastructure and the OS. On top of that, the container engine keeps containers isolated, but lets them share the single base OS's services.

Because containers require far fewer resources (for example, they don't need a full OS), they're easy to deploy and they start fast. This allows you to have higher density, meaning that it allows you to run more services on the same hardware unit, thereby reducing costs.

As a side effect of running on the same kernel, you get less isolation than VMs.

The main goal of an image is to ensure the same environment (dependencies) across different deployments. This means that you can debug it on your machine and then deploy it to another machine, the same environment guaranteed.

A container image is a way to package an app or service and deploy it in a reliable and reproducible way. You could say that Docker isn't only a technology but also a philosophy and a process.

When using Docker, you won't hear developers say, "It works on my machine, why not in production?" They can just say, "It runs on Docker", because the packaged Docker application can be executed on any supported Docker environment, and it runs the way it was intended to on all deployment targets (such as Dev, QA, staging, and production).

A simple analogy

Perhaps a simple analogy can help getting the grasp of the core concept of Docker.

Let's go back in time to the 1950s for a moment. There were no word processors, and the photocopiers were used everywhere (well, kind of).

Imagine you're responsible for quickly issuing batches of letters as required, to mail them to customers, using real paper and envelopes, to be delivered physically to each customer's address (there was no email back then).

At some point, you realize the letters are just a composition of a large set of paragraphs, which are picked and arranged as needed, according to the purpose of the letter, so you devise a system to issue letters quickly, expecting to get a hefty raise.

The system is simple:

1. You begin with a deck of transparent sheets containing one paragraph each.
2. To issue a set of letters, you pick the sheets with the paragraphs you need, then you stack and align them so they look and read fine.
3. Finally, you place the set in the photocopier and press start to produce as many letters as required.

So, simplifying, that's the core idea of Docker.

In Docker, each layer is the resulting set of changes that happen to the filesystem after executing a command, such as, installing a program.

So, when you "look" at the filesystem after the layer has been copied, you see all the files, included the layer when the program was installed.

You can think of an image as an auxiliary read-only hard disk ready to be installed in a "computer" where the operating system is already installed.

Similarly, you can think of a container as the "computer" with the image hard disk installed. The container, just like a computer, can be powered on or off.

Learn Docker specific terminologies

This section lists terms and definitions you should be familiar with before getting deeper into Docker. For further definitions, see the extensive glossary provided by Docker.

Container image: A package with all the dependencies and information needed to create a container. An image includes all the dependencies (such as frameworks) plus deployment and execution configuration to be used by a container runtime. Usually, an image derives from multiple base images that are layers stacked on top of each other to form the container's filesystem. An image is immutable once it has been created.

Dockerfile: A text file that contains instructions for building a Docker image. It's like a batch script, the first line states the base image to begin with and then follow the instructions to install required programs, copy files, and so on, until you get the working environment you need.

Build: The action of building a container image based on the information and context provided by its Dockerfile, plus additional files in the folder where the image is built. You can build images with the following Docker command:

```
docker build
```

Container: An instance of a Docker image. A container represents the execution of a single application, process, or service. It consists of the contents of a Docker image, an execution environment, and a standard set of instructions. When scaling a service, you create multiple instances of a container from the same image. Or a batch job can create multiple containers from the same image, passing different parameters to each instance.

Volumes: Offer a writable filesystem that the container can use. Since images are read-only but most programs need to write to the filesystem, volumes add a writable layer, on top of the container image, so the programs have access to a writable filesystem. The program doesn't know it's accessing a layered filesystem, it's just the filesystem as usual. Volumes live in the host system and are managed by Docker.

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

Multi-stage Build: Is a feature, since Docker 17.05 or higher, that helps to reduce the size of the final images. For example, a large base image, containing the SDK can be used for compiling and publishing and then a small runtime-only base image can be used to host the application.

Repository (repo): A collection of related Docker images, labeled with a tag that indicates the image version. Some repos contain multiple variants of a specific image, such as an image containing SDKs (heavier), an image containing only runtimes (lighter), etc. Those variants can be marked with tags. A single repo can contain platform variants, such as a Linux image and a Windows image.

Registry: A service that provides access to repositories. The default registry for most public images is Docker Hub (owned by Docker as an organization). A registry usually contains repositories from multiple teams. Companies often have private registries to store and manage images they've created. Azure Container Registry is another example.

Multi-arch image: For multi-architecture, it's a feature that simplifies the selection of the appropriate image, according to the platform where Docker is running. For example, when a Dockerfile requests a base image **FROM mcr.microsoft.com/dotnet/sdk:6.0** from the registry, it actually gets **6.0-nanoserver-20H2**, **6.0-nanoserver-1809** or **6.0-bullseye-slim**, depending on the operating system and version where Docker is running.

Docker Hub: A public registry to upload images and work with them. Docker Hub provides Docker image hosting, public or private registries, build triggers and web hooks, and integration with GitHub and Bitbucket.

Azure Container Registry: A public resource for working with Docker images and its components in Azure. This provides a registry that's close to your deployments in Azure and that gives you control over access, making it possible to use your Azure Active Directory groups and permissions.

Docker Trusted Registry (DTR): A Docker registry service (from Docker) that can be installed on-premises so it lives within the organization's datacenter and network. It's convenient for private images that should be managed within the enterprise. Docker Trusted Registry is included as part of the Docker Datacenter product. For more information, see Docker Trusted Registry (DTR).

Docker Desktop: Development tools for Windows and macOS for building, running, and testing containers locally. Docker Desktop for Windows provides development environments for both Linux and Windows Containers. The Linux Docker host on Windows is based on a Hyper-V virtual machine. The host for Windows Containers is directly based on Windows. Docker Desktop for Mac is based on the Apple Hypervisor framework and the xhyve hypervisor, which provides a Linux Docker host virtual machine on macOS. Docker Desktop for Windows and for Mac replaces Docker Toolbox, which was based on Oracle VirtualBox.

Compose: A command-line tool and YAML file format with metadata for defining and running multi-container applications. You define a single application based on multiple images with one or more .yml files that can override values depending on the environment. After you've created the definitions, you can deploy the whole multi-container application with a single command (docker-compose up) that creates a container per image on the Docker host.

Cluster: A collection of Docker hosts exposed as if it were a single virtual Docker host, so that the application can scale to multiple instances of the services spread across multiple hosts within the cluster. Docker clusters can be created with Kubernetes, Azure Service Fabric, Docker Swarm and Mesosphere DC/OS.

Orchestrator: A tool that simplifies the management of clusters and Docker hosts. Orchestrators enable you to manage their images, containers, and hosts through a command-line interface (CLI) or a graphical UI. You can manage container networking, configurations, load balancing, service discovery, high availability, Docker host configuration, and more. An orchestrator is responsible for running, distributing, scaling, and healing workloads across a collection of nodes. Typically, orchestrator products are the same products that provide cluster infrastructure, like Kubernetes and Azure Service Fabric, among other offerings in the market.

Learn docker containers, images, and registries

When using Docker, you create an app or service and package it and its dependencies into a container image. An image is a static representation of the app or service and its configuration and dependencies.

To run the app or service, the app's image is instantiated to create a container, which will be running on the Docker host. Containers are initially tested in a development environment or PC.

You store images in a registry that acts as a library of images. You need a registry when deploying to production orchestrators. Docker maintains a public registry via Docker Hub; other vendors provide registries for different collections of images, including Azure Container Registry. Alternatively, enterprises can have a private registry on-premises for their own Docker images.

Figure 1-4 shows how images and registries in Docker relate to other components. It also shows the multiple registry offerings from vendors.

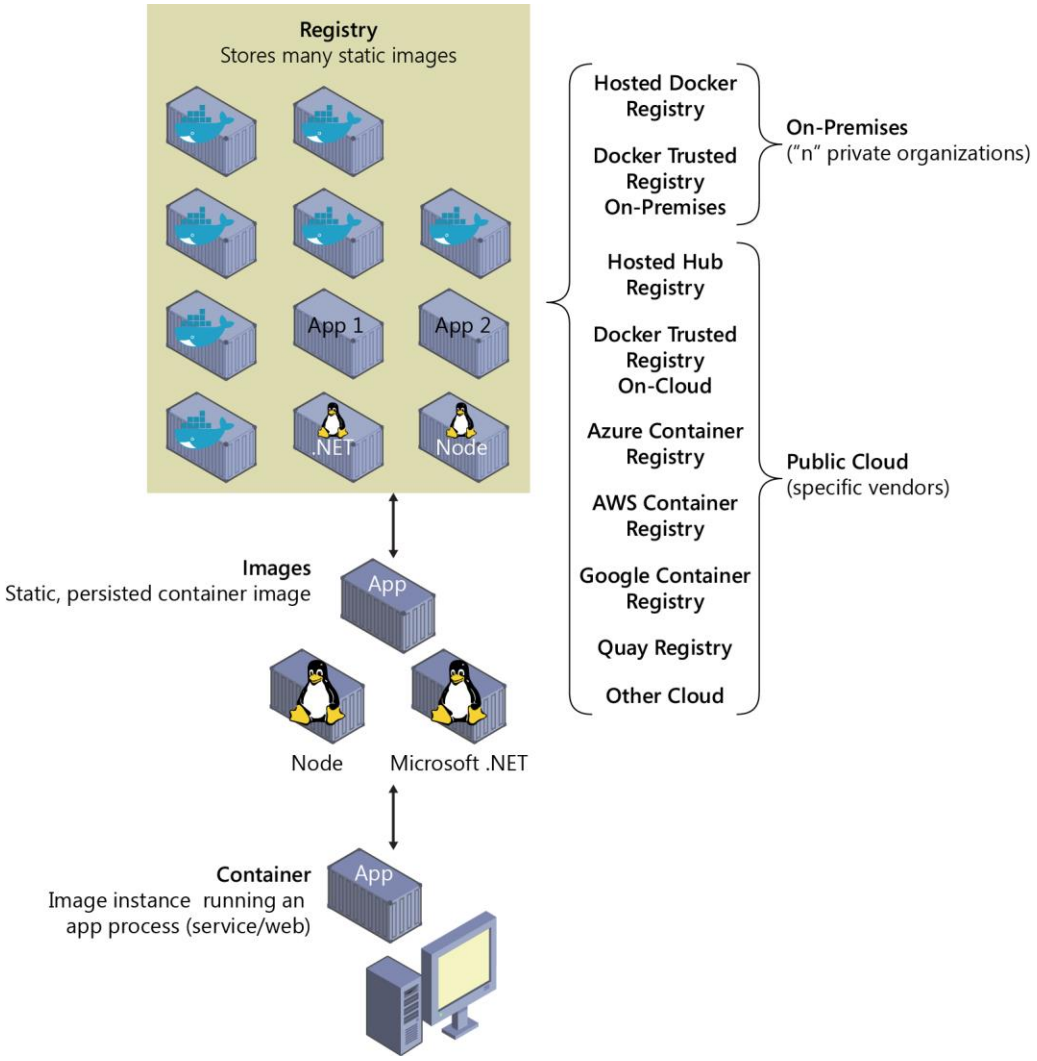

Figure 1-4. Taxonomy of Docker terms and concepts

The registry is like a bookshelf where images are stored and available to be pulled for building containers to run services or web apps. There are private Docker registries on-premises and on the public cloud. Docker Hub is a public registry maintained by Docker, along the Docker Trusted Registry an enterprise-grade solution, Azure offers the Azure Container Registry. AWS, Google and others also have container registries.

By putting images in a registry, you can store static and immutable application bits, including all of their dependencies, at a framework level. You then can version and deploy images in multiple environments and thus provide a consistent deployment unit.

Private image registries, either hosted on-premises or in the cloud, are recommended when:

- Your images must not be shared publicly due to confidentiality.
- You want to have minimum network latency between your images and your chosen deployment environment. For example, if your production environment is Azure, you

probably want to store your images in Azure Container Registry so that network latency is minimal. In a similar way, if your production environment is on-premises, you might want to have an on-premises Docker Trusted Registry available within the same local network.

Road to modern applications based on containers

You're probably reading this book because you're planning the development of new applications or you're assessing the impact of using Docker, Containers, and new approaches like Microservices in your company.

The adoption of new development paradigms must be taken with caution before starting a project, to assess the impact on your dev teams, your budget, or your infrastructure.

Microsoft has been working on rich guidance, sample applications, and a suite of e-books that can help you make an informed decision and guide your team through a successful development, deployment, and operations of your new applications.

This book belongs to a Microsoft suite of guides that cover many of the needs and challenges you'll face during the process of developing new modern applications based on containers.

You can find additional Microsoft e-books related to Docker containers in the list below:

- **.NET Microservices: Architecture for Containerized .NET Applications**
  https://docs.microsoft.com/dotnet/architecture/microservices/
- **Modernize existing .NET applications with Azure cloud and Windows Containers**
  https://docs.microsoft.com/dotnet/architecture/modernize-with-azure-containers/

CHAPTER 2

Introduction to the Docker application life cycle

The life cycle of containerized applications is a journey that begins with the developer. The developer chooses to implement containers and Docker because it eliminates frictions in deployments and IT operations, which ultimately helps everyone to be more agile, more productive end-to-end, and faster.

Containers as the foundation for DevOps collaboration

By the very nature of the containers and Docker technology, developers can share their software and dependencies easily 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 operations closer together, making it easier for them to collaborate effectively. Adopting the container workflow provides many customers with the DevOps continuity they've sought but previously had to implement via more complex configuration for release and build pipelines. Containers simplify the build/test/deploy pipelines in DevOps.

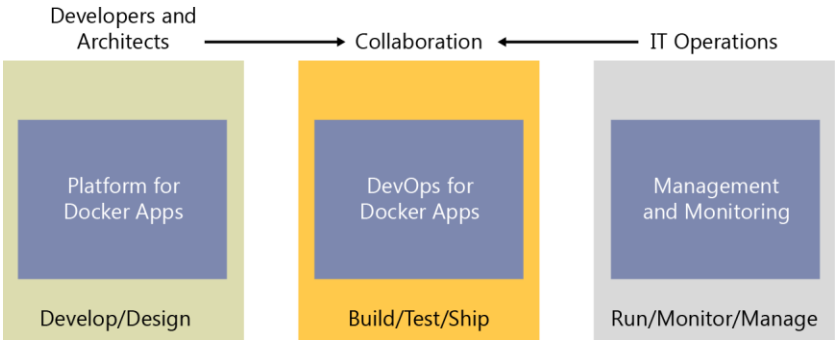

Figure 2-1. Main workloads per "personas" in the life cycle for containerized Docker applications

With Docker containers, developers own what's within the container (application and service, and dependencies to frameworks and components) and how the containers and services behave together as an application composed by a collection of services. The

interdependencies of the multiple containers are defined in a `docker-compose.yml` file, or what could be called a *deployment manifest*. Meanwhile, IT operations teams (IT professionals and management) can focus on the management of production environments; infrastructure; scalability; monitoring; and, ultimately, ensuring that the applications are delivering properly for the end users, without having to know the contents of the various containers. Hence, the name "container," recalling the analogy to real-world shipping containers. Thus, the owners of a container's content need not concern themselves with how the container will be shipped, and the shipping company transports a container from its point of origin to its destination without knowing or caring about the contents. In a similar manner, developers can create and own the contents within a Docker container without the need to concern themselves with the "transport" mechanisms.

In the pillar on the left side of Figure 2-1, developers write and run code locally in Docker containers by using Docker for Windows or Mac. They define the operating environment for the code by using a Dockerfile that specifies the base operating system to run as well as the build steps for building their code into a Docker image. The developers define how one or more images will interoperate using the aforementioned `docker-compose.yml` file deployment manifest. As they complete their local development, they push their application code plus the Docker configuration files to the code repository of their choice (that is, Git repository).

The DevOps pillar defines the build–Continuous Integration (CI) pipelines using the Dockerfile provided in the code repository. The CI system pulls the base container images from the selected Docker registry and builds the custom Docker images for the application. The images then are validated and pushed to the Docker registry used for the deployments to multiple environments.

In the pillar on the right, operations teams manage deployed applications and infrastructure in production while monitoring the environment and applications so that they can provide feedback and insights to the development team about how the application might be improved. Container apps are typically run in production using container orchestrators like Kubernetes, where usually Helm charts are used to configure deployment units, instead of docker-compose files.

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 life cycle. The developers own the container contents, its operating environment, and the container interdependencies, whereas the operations teams take the built images along with the manifest and runs them in their orchestration system.

Challenges in the application life cycle when using Docker.

There are many reasons that will increase the number of containerized applications in the upcoming years, and one of these reasons is the creation of applications based on microservices.

During the last 15 years, the use of web services has been the base of thousands of applications, and probably, after a few years, you'll find the same situation with microservice-based applications running on Docker containers.

It is also worth to mention that you can also use Docker containers for monolithic applications and you still get most of the benefits of Docker. Containers are not targeting only microservices.

The use of Docker containerization and microservices causes new challenges in the development process of your organizations and therefore, you need a solid strategy to maintain many containers and microservices running on production systems. Eventually, enterprise applications will have hundreds or thousands of containers/instances running in production.

These challenges create new demands when using DevOps tools, so you'll have to define new processes in your DevOps activities, and find answers for the following type of questions:

- Which tools can I use for development, CI/CD, management and operations??
- How can my company manage errors in containers when running in production?
- How can we change pieces of our software in production with minimum downtime?
- How can we scale and monitor our production system?
- How can we include the testing and deployment of containers in our release pipeline?
- How can we use Open Source tools/platforms for containers in Microsoft Azure?

If you can answer all those questions, you'll be better prepared to move your applications (existing or new apps) to Docker containers.

Introduction to a generic end-to-end Docker application life cycle workflow

Figure 2-2 presents a more detailed workflow for a Docker application life cycle, focusing in this instance on specific DevOps activities and assets.

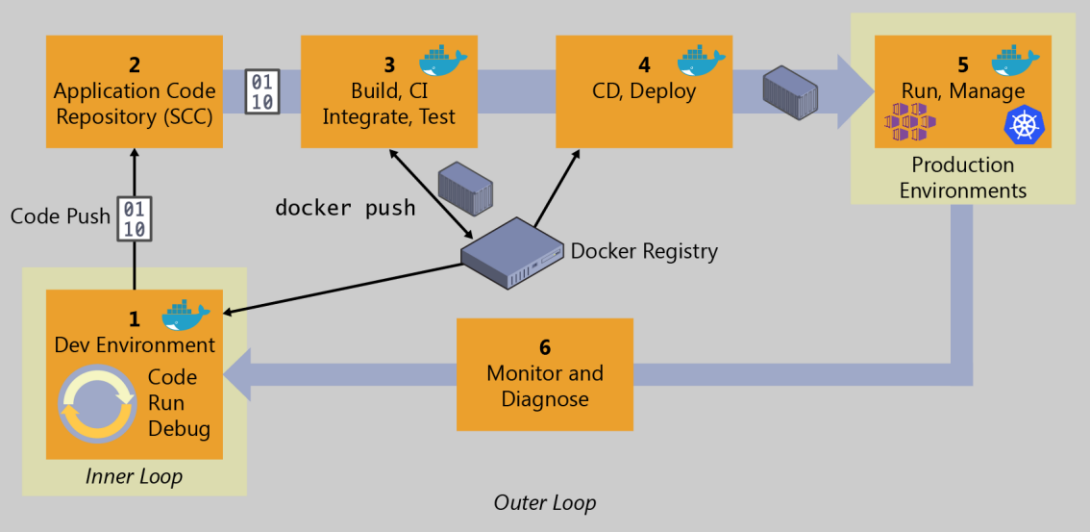

Figure 2-2. High-level workflow for the Docker containerized application life cycle

Everything begins with the developer, who starts writing code in the inner-loop workflow. The inner-loop stage is where developers define everything that happens before pushing code into the code repository (for example, a source control system such as Git). After it's committed, the repository triggers Continuous Integration (CI) and the rest of the workflow.

The inner loop consists of typical steps like "code," "run," "test," and "debug," plus the additional steps needed right before running the app locally. This is the developer's process to run and test the app as a Docker container. The inner-loop workflow will be explained in the sections that follow.

Taking a step back to look at the end-to-end workflow, the DevOps workflow is more than a technology or a tool set, it's a mindset that requires cultural evolution. It's people, processes, and the appropriate tools to make your application life cycle faster and more predictable. Enterprises that adopt a containerized workflow typically restructure their organizations to represent people and processes that match the containerized workflow.

Practicing DevOps can help teams respond faster together to competitive pressures by replacing error-prone manual processes with automation, which results in improved traceability and repeatable workflows. Organizations also can manage environments more efficiently and realize 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 technologies are present in almost every stage of the workflow, from your development box while working in the inner loop (code, run, debug), the build-test-CI phase, and, finally, the deployment of those containers to the staging and production 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 and adopting a philosophy of deploying the same image across multiple environments, you promote 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 you can demonstrate success, learn, and evolve.

Benefits of DevOps for containerized applications

Here are some of the most important benefits provided by a solid DevOps workflow:

- 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.

CHAPTER 3

Introduction to the Microsoft platform and tools for containerized apps

Vision: Create an adaptable, enterprise-grade, containerized application life cycle that spans your development, IT operations, and production management.

Figure 3-1 shows the main pillars in the life cycle 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.

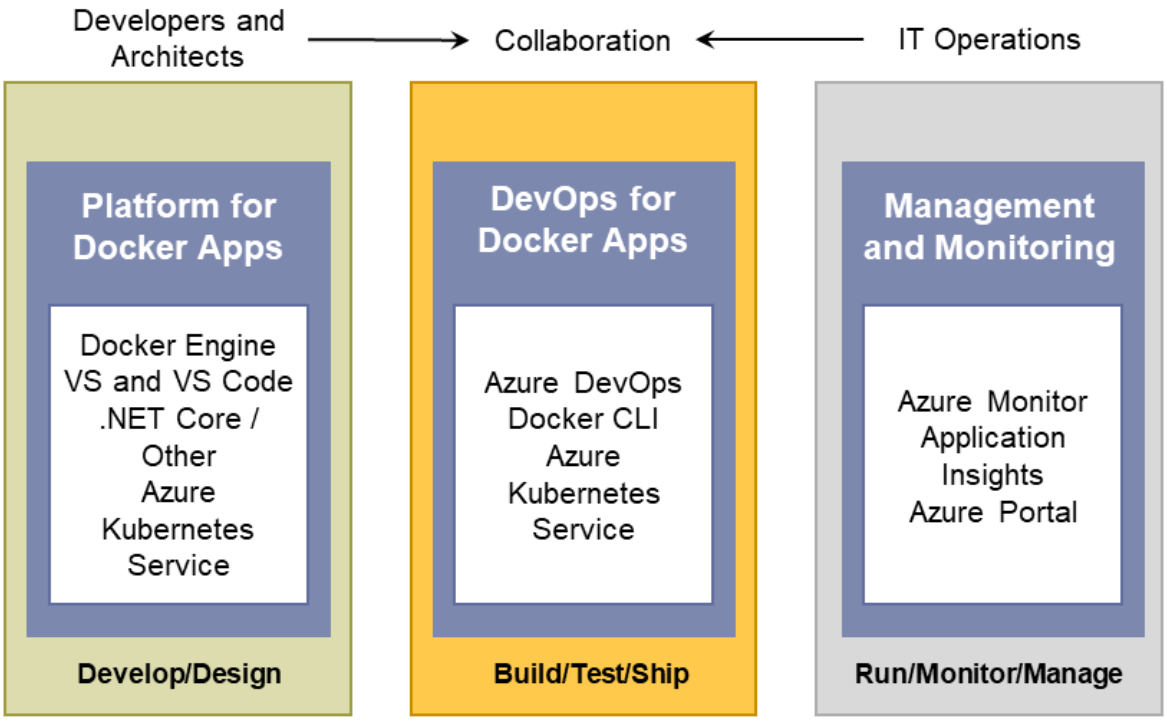

Figure 3-1. Main pillars in the life cycle for containerized Docker applications with Microsoft platform and tools

A containerized Docker life-cycle workflow can be initially prescriptive based on "by-default product choices," making it easier for developers to get started faster, but it's fundamental that under the hood there must be an open framework so that it will be a flexible workflow capable of adjusting to the different contexts from each organization or 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 the broad choice of technologies in the platform and infrastructure are precisely the Microsoft priorities for containerized Docker applications, as explained in the chapters that follow.

Table 3-1 demonstrates that the intention of the Azure DevOps for containerized Docker applications is to provide an open DevOps workflow so that you can choose what products to use for each phase (Microsoft or third-party) while providing a simplified workflow that provides "by-default-products" already connected; thus, you can quickly get started with your enterprise-level DevOps workflow for Docker apps.

Table 3-1. DevOps workflows, open to any technology

Host	Microsoft technologies	Third-party (Azure pluggable)
Platform for Docker apps	• Microsoft Visual Studio and Visual Studio Code • .NET • Microsoft Azure Kubernetes Service (AKS) • Azure Container Registry\	• Any code editor (for example, Sublime) • Any language (Node.js, Java, Go, etc.) • Any orchestrator and scheduler • Any Docker registry\
DevOps for Docker apps	• Azure DevOps Services • Microsoft Team Foundation Server • GitHub • Azure Kubernetes Service (AKS)\	• GitHub, Git, Subversion, etc. • Jenkins, Chef, Puppet, Velocity, CircleCI, TravisCI, etc. • On-premises Docker Datacenter, Kubernetes, Mesos DC/OS, etc.\
Management and monitoring	• Azure Monitor	• Marathon, Chronos, etc.\

The Microsoft platform and tools for containerized Docker apps, as defined in Table 3-1, comprise 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 developers requires prior to pushing their code to a shared code repository. Developing services, deployed as containers, are 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 destination, you develop your services in the Docker environment. You build, run, test, and debug your code in containers locally, providing the destination environment at development time. By providing the destination environment locally, Docker containers set up what will drastically help you improve your DevOps life cycle. Visual Studio and Visual Studio Code have extensions to integrate Docker containers within your development process.
- **DevOps for Docker Apps** Developers creating Docker applications can use Azure DevOps, GitHub or any other third-party product, like Jenkins, to build out a comprehensive automated application life-cycle management (ALM).

With Azure DevOps and/or GitHub, developers can create container-focused DevOps for a fast, iterative process that covers source-code control from anywhere (Azure DevOps-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 also can automate their Docker application releases into Azure Kubernetes Service (AKS), from development to staging and production environments.

- **Management and Monitoring** IT can manage and monitor production applications and services in several ways, integrating both perspectives in a consolidated experience.
- **Azure portal** Azure Kubernetes Service (AKS) helps you to set up and maintain your Docker environments. You can also use other orchestrators 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 also can use other third-party tools to manage your Docker applications or even CLI Docker tools.

Even if you're familiar with Linux commands, you can manage your container applications using Microsoft Windows and PowerShell with a Linux Subsystem command line and the products (Docker, Kubernetes…) clients running on this Linux Subsystem capability. You'll learn more about using these tools under Linux Subsystem using your favorite Microsoft Windows OS later in this book.

- **Open-source tools** Because AKS exposes the standard API endpoints for the orchestration engine, the most popular tools are compatible with AKS and, in most cases, will work out of the box—including visualizers, monitoring, command-line tools, and even future tools as they become available.
- **GitHub Advanced Security** GitHub Advanced Security offers a suite of tools for securing the software supply chain that can seamlessly integrate security into the daily workflow of teams developing containerized applications.
- **Azure Monitor** Is Azure's solution to monitor every angle of your production environment. You can monitor production Docker applications by just setting up its SDK into your services so that you can get system-generated log data from the applications.

Thus, Microsoft offers a complete foundation for an end-to-end containerized Docker application life cycle. However, it's *a collection of products and technologies that allow you to optionally select and integrate with existing tools and processes*. The flexibility in a broad approach along with the strength in the depth of capabilities place Microsoft in a strong position for containerized Docker application development.

CHAPTER 4

Designing and developing containerized apps using Docker and Microsoft Azure

Vision: Design and develop 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.

Design Docker applications

Chapter 1 introduced the fundamental concepts regarding containers and Docker. That information is the basic level of information you need to get started. But, enterprise applications can be complex and composed of multiple services instead of a single service or container. For those optional use cases, you need to know additional approaches to design, such as Service-Oriented Architecture (SOA) 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 life cycle, therefore, it does not explore microservices architecture in depth because you can also use containers and Docker with regular SOA, background tasks or jobs, or even with monolithic application deployment approaches.

More info To learn more about enterprise applications and microservices architecture in depth, read the guide NET Microservices: Architecture for Containerized .NET Applications that you can also download from https://aka.ms/MicroservicesEbook.

However, before you get into the application life cycle and DevOps, it's important to know how you're going to design and construct your application and what are your design choices.

Common container design principles

Ahead of getting into the development process there are a few basic concepts worth mentioning with regard to how you use containers.

Container equals a process

In the container model, a container represents a single process. By defining a container as a process boundary, you begin to create the primitives used to scale, or batch-off, processes. When you run 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 life-cycle ends. There are long-running processes, such as web servers, and short-lived processes, such as batch jobs, which might have been implemented as Microsoft Azure WebJobs. If the process fails, the container ends, and the orchestrator takes over. If the orchestrator was instructed to keep five instances running and one fails, the orchestrator will create 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 might find a scenario in which you 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 Supervisor.

Monolithic applications

In this scenario, you're building a single and monolithic web application or service and deploying it as a container. Within the application, the structure might not be monolithic; it might comprise several libraries, components, or even layers (application layer, domain layer, data access layer, etc.). Externally, it's a single container, like a single process, single web application, or single service.

To manage this model, you deploy a single container to represent the application. To scale it, 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 virtual machine (VM).

Following the principal that a container does one thing only, 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 4-1.

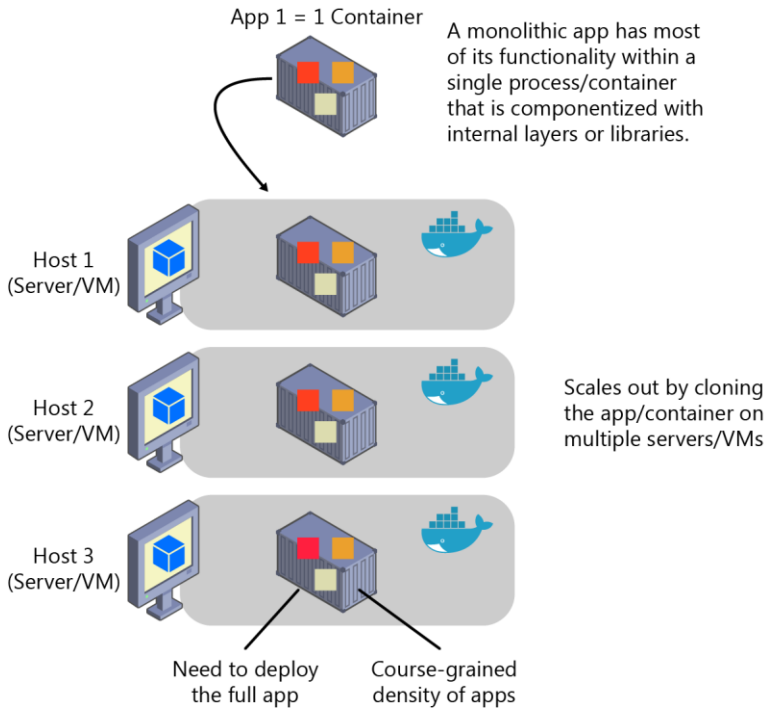

Figure 4-1. An example of monolithic application architecture

A monolithic app has all or most of its functionality within a single process or container and it's componentized in internal layers or libraries. The downside to this approach comes if or 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 that require scaling, whereas other components are used less.

Using the typical e-commerce example, what you likely need is to scale the product information component. Many more customers browse products than purchase them. More customers use their basket than use the payment pipeline. Fewer customers add comments or view their purchase history. And you likely have only a handful of employees, in a single region, that need to manage the content and marketing campaigns. By scaling the monolithic design, all of 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 as well as a complete redeployment of all the instances.

The monolithic approach is common, and many organizations are developing with this architectural method. Many enjoy good enough results, whereas others encounter limits. Many designed their applications in this model because the tools and infrastructure were too difficult to build SOAs, and they didn't see the need—until the app grew.

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 4-2.

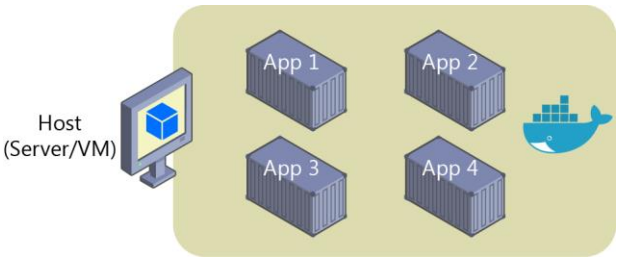

Figure 4-2. A host running multiple apps/containers

Finally, from an availability perspective, monolithic applications must be deployed as a whole; that means that in case you must *stop and start*, all functionality and all users will be affected during the deployment window. In certain situations, the use of Azure and containers can minimize these situations and reduce the probability of downtime of your application, as you can see in Figure 4-3.

You can deploy monolithic applications in Azure by using dedicated VMs for each instance. Using Azure VM Scale Sets, you can scale the VMs easily.

You can also use Azure App Services to run monolithic applications and easily scale instances without having to manage the VMs. Azure App Services can run single instances of Docker containers, as well, simplifying the deployment.

You can deploy multiple VMs as Docker hosts and run any number of containers per VM. Then, by using an Azure Load Balancer, as illustrated in the Figure 4-3, you can manage scaling.

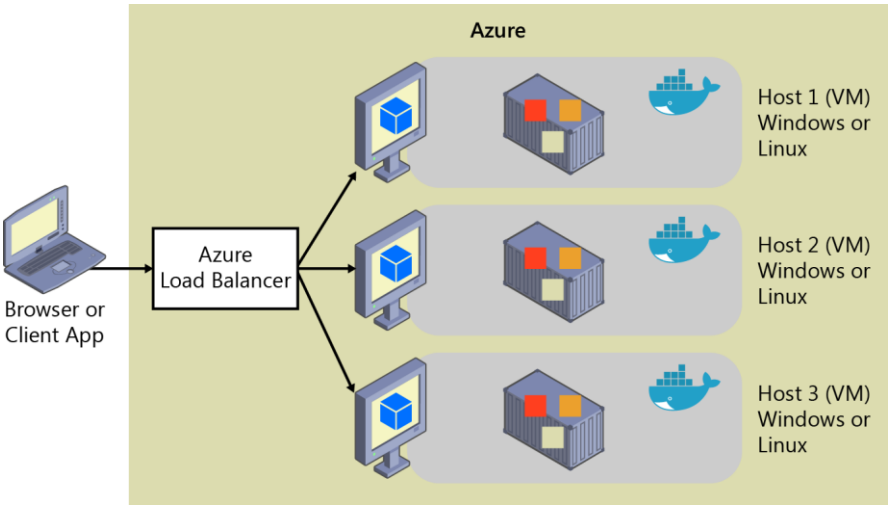

Figure 4-3. Multiple hosts scaling out a single Docker application

You can manage the deployment of the hosts themselves via traditional deployment techniques.

You can manage Docker containers from the command line by using commands like `docker run` and `docker-compose up`, and you can also automate it in Continuous Delivery (CD) pipelines and deploy to Docker hosts from Azure DevOps Services, for instance.

Monolithic application deployed as a container

There are benefits to using containers to manage monolithic deployments. Scaling the instances of containers is far faster and easier than deploying additional VMs.

Deploying updates as Docker images is far faster and network efficient. Docker containers typically start in seconds, speeding rollouts. Tearing down a Docker container is as easy as invoking the `docker stop` command, typically completing in less than a second.

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

Although monolithic apps can benefit from Docker, we're touching on only the tips of the benefits. The larger benefits of managing containers come from deploying with container orchestrators that manage the various instances and life cycle of each container instance. Breaking up the monolithic application into subsystems that can be scaled, developed, and deployed individually is your entry point into the realm of microservices.

To learn about how to "lift and shift" monolithic applications with containers and how you can modernize your applications, you can read this additional Microsoft guide, Modernize existing .NET applications with Azure cloud and Windows Containers, which you can also download as PDF from https://aka.ms/LiftAndShiftWithContainersEbook.

Publish a single Docker container app to Azure App Service

Either because 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 intuitive and you can get up and running quickly because it provides great Git integration to take your code, build it in Microsoft Visual Studio, and directly deploy it to Azure. But, traditionally (with no Docker), if you needed other capabilities, frameworks, or dependencies that aren't supported in App Services, you needed to wait for it until the Azure team updates those dependencies in App Service or switched to other services like Service Fabric, Cloud Services, or even plain VMs, for which you have further control and can install a required component or framework for your application.

Now, as shown in Figure 4-4, when using Visual Studio 2022, container 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, because you're running it in a container, you get the capability of including those dependencies in your Dockerfile or Docker image.

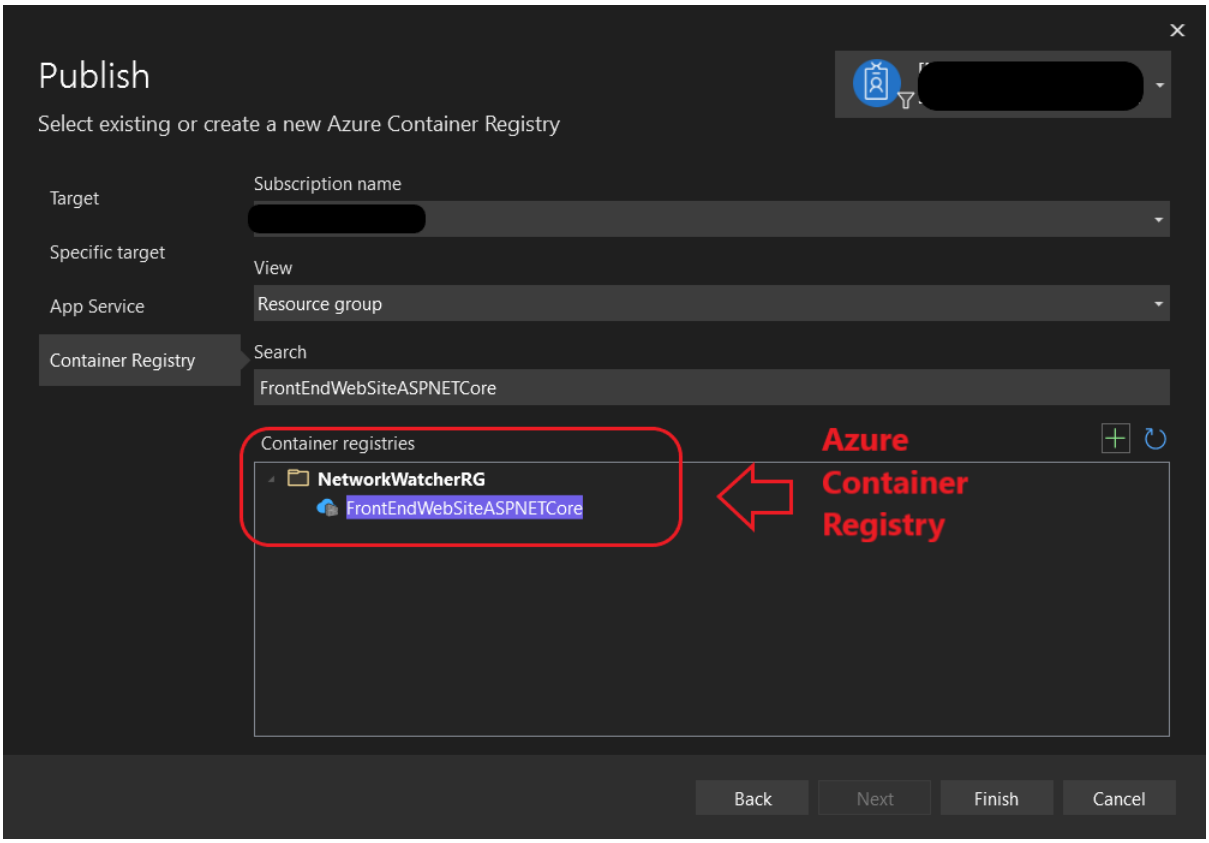

Figure 4-4. Publishing a container to Azure App Service from Visual Studio apps/containers

Figure 4-4 also shows that 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

In most cases, you can think of a container as an instance of a process. A process does not maintain persistent state. While a container can write to its local storage, assuming that an instance will be around indefinitely is like assuming that a single location in memory will be durable. Container images, like processes, should be assumed to have multiple instances and that they will eventually be killed; if they're managed with a container orchestrator, it should be assumed that they might get moved from one node or VM to another.

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

From the Docker host, as Docker Volumes:

- **Volumes** are stored in an area of the host filesystem that's managed by Docker.
- **Bind mounts** can map to any folder in the host filesystem, so access can't be controlled from a Docker process and can pose a security risk as a container could access sensitive OS folders.
- **tmpfs mounts** are like virtual folders that only exist in the host's memory and are never written to the filesystem.

From remote storage:

- Azure Storage provides geo-distributable storage, providing a good long-term persistence solution for containers.
- Remote relational databases like Azure SQL Database, NoSQL databases like Azure Cosmos DB, or cache services like Redis.

From the Docker container:

- Docker provides a feature named the *overlay file system*. This feature implements a copy-on-write task that stores updated information to the root file system of the container. That information "lays on top of" the original image on which the container is based. If the container is deleted from the system, those changes are lost. Therefore, while it's possible to save the state of a container within its local storage, designing a system based on this feature would conflict with the premise of container design, which by default is stateless.
- However, Docker Volumes is now the preferred way to handle local data in Docker. If you need more information about storage in containers, check on Docker storage drivers and About images, containers, and storage drivers.

The following provides additional detail about these options.

Volumes are directories mapped from the host OS to directories in containers. When code in the container has access to the directory, that access is actually to a directory on the host OS. This directory is not tied to the lifetime of the container itself, and the directory is managed by Docker and isolated from the core functionality of the host machine. Thus, data volumes are designed to persist data independently of the life of the container. If you delete a container or an image from the Docker host, the data persisted in the data volume is not deleted.

Volumes can be named or anonymous (the default). Named volumes are the evolution of **Data Volume Containers** and make it easy to share data between containers. Volumes also support volume drivers that allow you to store data on remote hosts, among other options.

Bind mounts have been available for a long time and allow the mapping of any folder to a mount point in a container. Bind mounts have more limitations than volumes and some important security issues, so volumes are the recommended option.

`tmpfs` **mounts** are virtual folders that live only in the host's memory and are never written to the filesystem. They are fast and secure but use memory and are only meant for non-persistent data.

As shown in Figure 4-5, regular Docker volumes can be stored outside of the containers themselves but within the physical boundaries of the host server or VM. However, Docker containers cannot access a volume from one host server or VM to another. In other words, with these volumes, it isn't possible to manage data shared between containers that run on different Docker hosts, although it could be achieved with a volume driver that supports remote hosts.

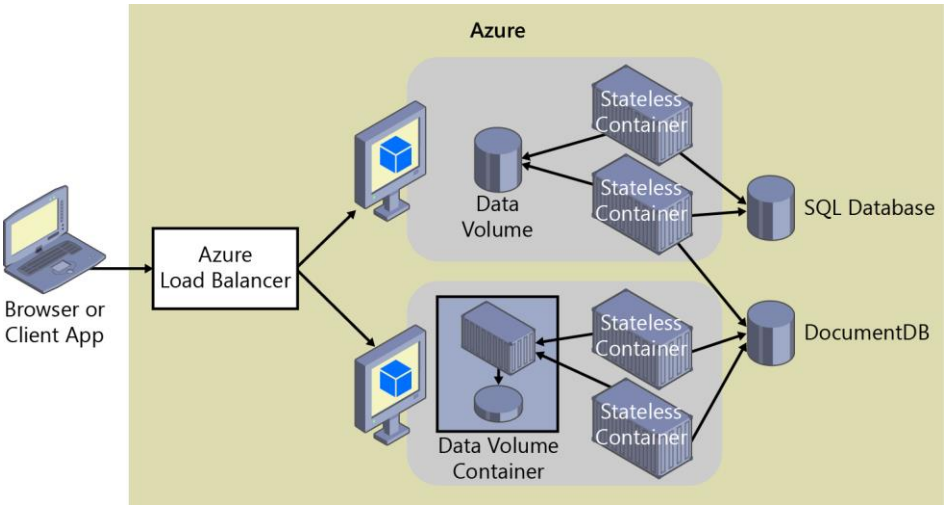

Figure 4-5. Volumes and external data sources for container-based applications

In addition, when Docker containers are managed by an orchestrator, containers might "move" between hosts, depending on the optimizations performed by the cluster. Therefore, it isn't recommended that you use data volumes for business data. But they are a good mechanism to work with trace files, temporal files, or similar, that will not impact business data consistency.

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

Azure Storage. Business data usually needs to be placed in external resources or databases, like Azure Storage. Azure Storage provides the following services in the cloud:

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

Relational databases and NoSQL databases. There are many choices for external databases, from relational databases like SQL Server, PostgreSQL, Oracle, or NoSQL databases like Azure Cosmos DB, MongoDB, etc. These databases are not going to be explained as part of this guide since they are a different topic altogether.

Service-oriented applications

Service-Oriented Architecture (SOA) was an overused term that meant many different things to different people. But as a common denominator, SOA means that you structure the architecture of your application by decomposing it into several services (most commonly as HTTP services) that can be classified in different types like subsystems or, in other cases, as tiers.

Today, you can deploy those services as Docker containers, which solve deployment-related issues because all of the dependencies are included in the container image. However, when you need to scale out SOAs, you might encounter challenges if you're deploying based on single instances. This challenge can be handled using Docker clustering software or an orchestrator. You'll get to look at orchestrators in greater detail in the next section, when you explore microservices approaches.

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

At the end of the day, the container clustering solutions are useful for both a traditional SOA architecture and for a more advanced microservices architecture in which 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, the discussion about splitting the data is purely about architecture and design.

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

Using orchestrators for production-ready applications is essential if your application is based on microservices or split across multiple containers. As introduced previously, in a microservice-based approach, each microservice owns its model and data so that it will be autonomous from a development and deployment point of view. But even if you have a more traditional application that's composed of multiple services (like SOA), you'll also have multiple containers or services comprising a single business application that need to be deployed as a distributed system. These kinds of systems are complex to scale out and manage; therefore, you absolutely need an orchestrator if you want to have a production-ready and scalable multi-container application.

Figure 4-6 illustrates deployment into a cluster of an application composed of multiple microservices (containers).

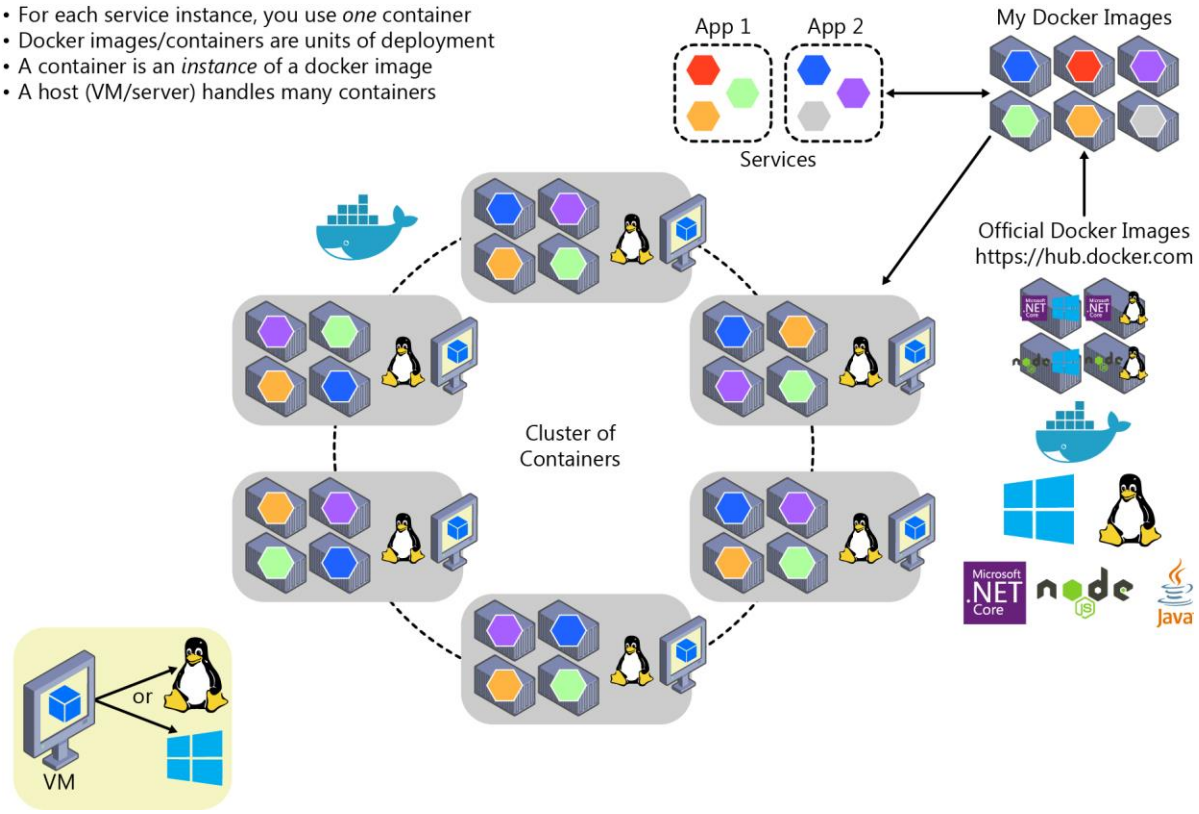

Figure 4-6. A cluster of containers

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

The Docker CLI meets the needs of managing one container on one host, but it falls short when it comes to managing multiple containers deployed on multiple hosts for more complex distributed applications. In most cases, you need a management platform that will automatically start containers, scale out containers with multiple instances per image, 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 simple composed apps and move toward larger enterprise applications with microservices, you must turn to orchestration and clustering platforms.

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

- **Clusters and orchestrators.** When you need to scale out applications across many Docker hosts, such as with a large microservices-based application, it's critical to be able to manage all of those hosts as a single cluster by abstracting the complexity of the underlying platform. That's what the container clusters and orchestrators provide. Examples of orchestrators are Azure Service Fabric and Kubernetes. Kubernetes is available in Azure through Azure Kubernetes Service.
- **Schedulers.** *Scheduling* means to have the capability for an administrator to launch containers in a cluster, so schedulers also provide a user interface for doing so. A cluster scheduler has several responsibilities: to use the cluster's resources efficiently, to set the constraints provided by the user, to efficiently load-balance containers across nodes or hosts, and to be robust against errors while providing high availability.

The concepts of a cluster and a scheduler are closely related, so the products provided by different vendors often provide both sets of capabilities. The section below shows the most important platform and software choices you have for clusters and schedulers. These orchestrators are widely offered in public clouds like Azure.

Software platforms for container clustering, orchestration, and scheduling

Platform	Comments
Kubernetes 	*Kubernetes* is an open-source product that provides functionality that ranges from cluster infrastructure and container scheduling to orchestrating capabilities. It lets you automate deployment, scaling, and operations of application containers across clusters of hosts. *Kubernetes* provides a container-centric infrastructure that groups application containers into logical units for easy management and discovery. *Kubernetes* is mature in Linux, less mature in Windows.
Azure Kubernetes Service (AKS) 	Azure Kubernetes Service (AKS) is a managed Kubernetes container orchestration service in Azure that simplifies Kubernetes cluster's management, deployment, and operations.
Azure Service Fabric 	Service Fabric is a Microsoft microservices platform for building applications. It's an orchestrator of services and creates clusters of machines. Service Fabric can deploy services as containers or as plain processes. It can even mix services in processes with services in containers within the same application and cluster. *Service Fabric* clusters can be deployed in Azure, on-premises or in any cloud. However, deployment in Azure is simplified with a managed approach. *Service Fabric* provides additional and optional prescriptive Service Fabric programming models like stateful services and Reliable Actors. *Service Fabric* is mature in Windows (years evolving in Windows), less mature in Linux. Both Linux and Windows containers are supported in Service Fabric since 2017.

Platform	Comments
Azure Service Fabric Mesh 	*Azure Service Fabric Mesh* offers the same reliability, mission-critical performance and scale as Service Fabric, but also offers a fully managed and serverless platform. You don't need to manage a cluster, VMs, storage or networking configuration. You just focus on your application's development. *Service Fabric Mesh* supports both Windows and Linux containers, allowing you to develop with any programming language and framework of your choice.
Azure Container Apps 	Azure Container Apps is a managed serverless container service for building and deploying modern apps at scale.

Using container-based orchestrators in Azure

Several cloud vendors offer Docker containers support plus Docker clusters and orchestration support, including Azure, Amazon EC2 Container Service, and Google Container Engine. Azure provides Docker cluster and orchestrator support through Azure Kubernetes Service (AKS), Azure Service Fabric, and Azure Service Fabric Mesh.

Using Azure Kubernetes Service

A Kubernetes cluster pools several Docker hosts and exposes them as a single virtual Docker host, so you can deploy multiple containers into the cluster and scale-out with any number of container instances. The cluster will handle all the complex management plumbing, like scalability, health, and so forth.

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

Azure Kubernetes 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 the orchestrator tools, and AKS handles everything else.

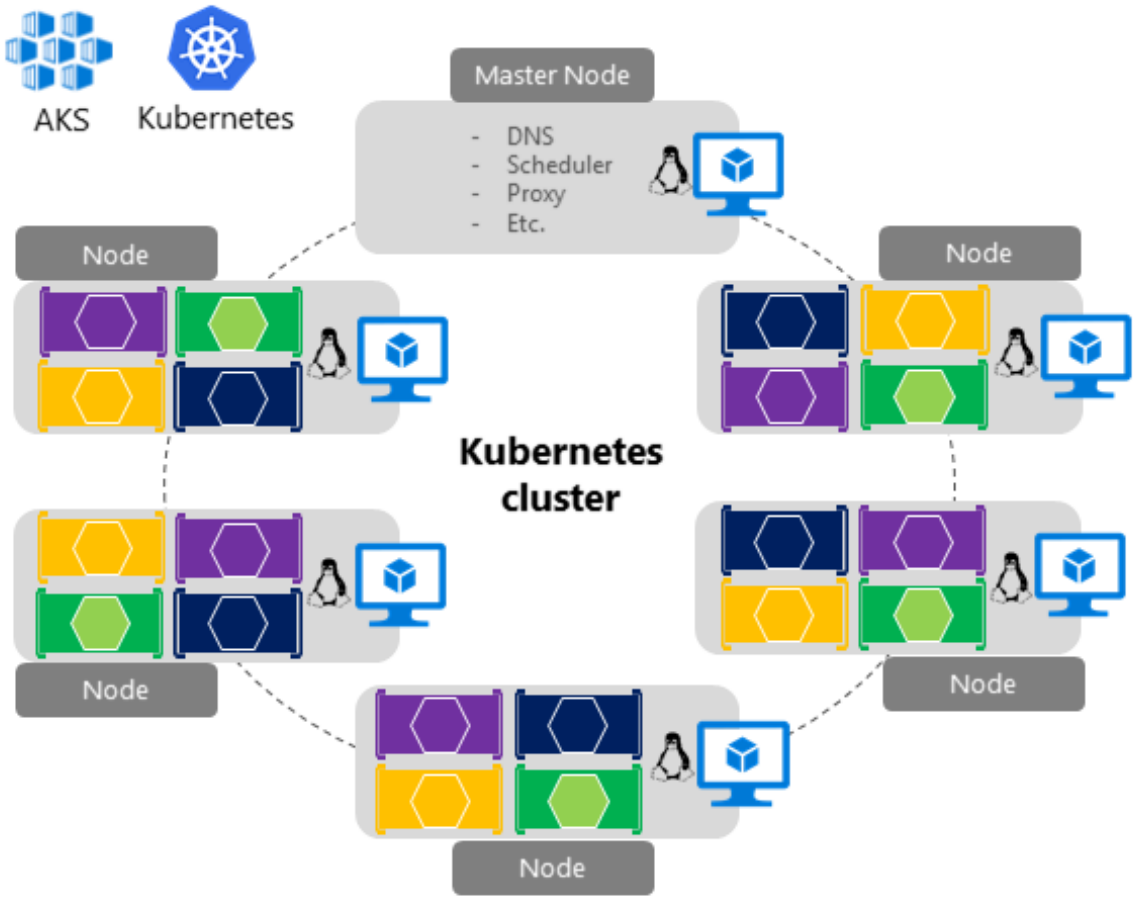

Figure 4-7. Kubernetes cluster's simplified structure and topology

Figure 4-7 shows the structure of a Kubernetes cluster where a master node (VM) controls most of the coordination of the cluster, and you can deploy containers to the rest of the nodes that are managed as a single pool from an application point of view. This allows you to scale to thousands or even tens of thousands of containers.

Development environment for Kubernetes

In the development environment that Docker announced in July 2018, Kubernetes can also run in a single development machine (Windows 10 or macOS) by just installing Docker Desktop. You can later deploy to the cloud (AKS) for further integration tests, as shown in figure 4-8.

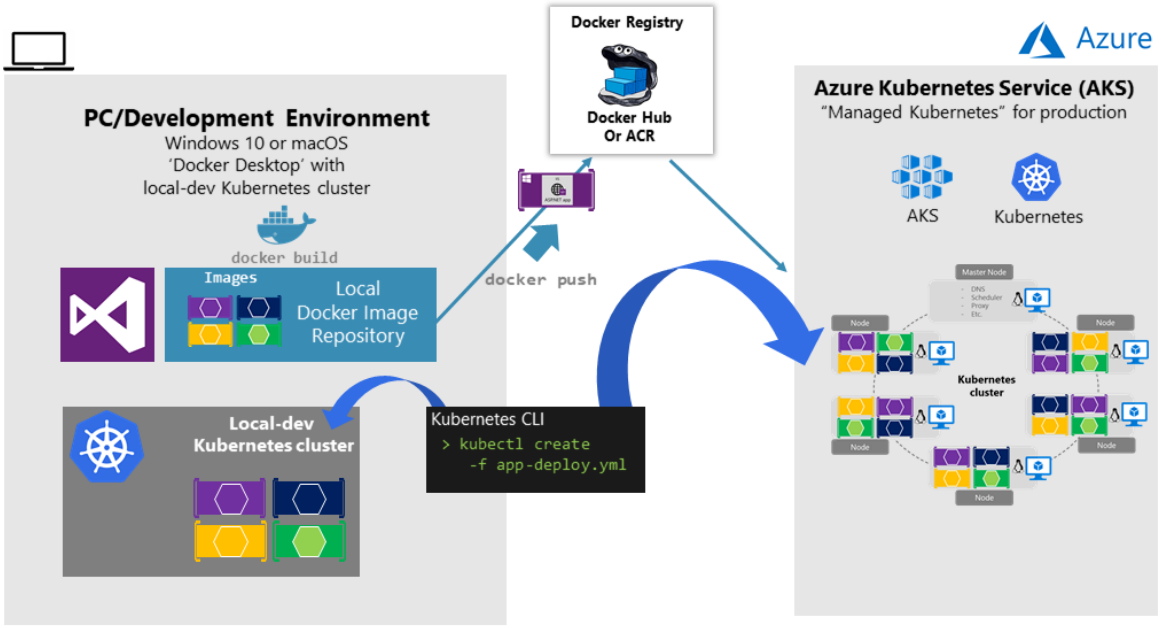

Figure 4-8. Running Kubernetes in dev machine and the cloud

Get started with Azure Kubernetes Service (AKS)

To begin using AKS, you deploy an AKS cluster from the Azure portal or by using the CLI. For more information on deploying a Kubernetes cluster to Azure, see Deploy an Azure Kubernetes Service (AKS) cluster.

There are no fees for any of the software installed by default as part of AKS. All default options are implemented with open-source software. AKS is available for multiple virtual machines in Azure. You're charged only 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 AKS itself.

For further implementation information on deployment to Kubernetes based on `kubectl` and original `.yaml` files, see Deploy to Azure Kubernetes Service (AKS).

Deploy with Helm charts into Kubernetes clusters

When deploying an application to a Kubernetes cluster, you can use the original `kubectl.exe` CLI tool using deployment files based on the native format (`.yaml` files), as already mentioned in the previous section. However, for more complex Kubernetes applications such as when deploying complex microservice-based applications, it's recommended to use Helm.

Helm Charts helps you define, version, install, share, upgrade, or rollback even the most complex Kubernetes application. Helm is maintained by the Cloud Native Computing

Foundation (CNCF) in collaboration with Microsoft, Google, Bitnami, and the Helm contributor community.

For further implementation information on Helm charts and Kubernetes, see the section called Install eShopOnContainers using Helm.

Additional resources

- **Getting started with Azure Kubernetes Service (AKS)**
  https://docs.microsoft.com/azure/aks/kubernetes-walkthrough-portal
- **Kubernetes.** The official site.
  https://kubernetes.io/

Using Azure Service Fabric

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

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

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

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

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

As shown in Figure 4-10, you can create and run microservices in Service Fabric either as simple processes or as Docker containers. It's also possible to mix container-based microservices with process-based microservices within the same Service Fabric cluster.

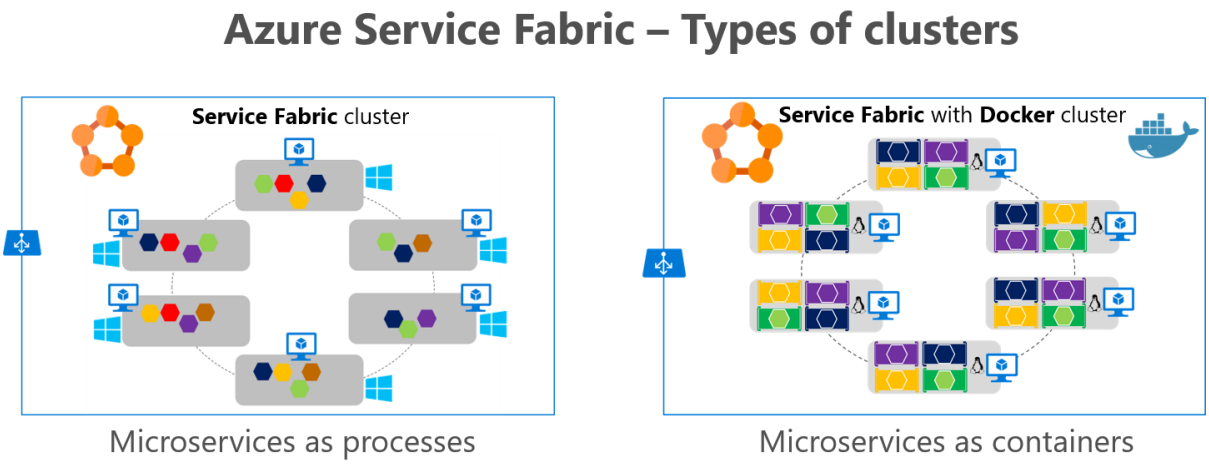

Figure 4-10. Deploying microservices as processes or as containers in Azure Service Fabric

In the first image, you see microservices as processes, where each node runs one process for each microservice. In the second image, you see microservices as containers, where each node runs Docker with several containers, one container per microservice. Service Fabric clusters based on Linux and Windows hosts can run Docker Linux containers and Windows Containers, respectively.

For up-to-date information about containers support in Azure Service Fabric, see Service Fabric and containers.

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

As shown in Figure 4-10, and thinking from a logical/business microservice perspective, when implementing a Service Fabric Stateful Reliable Service, you usually will need to implement two tiers of services. The first is the back-end stateful reliable service, which handles multiple partitions (each partition is a stateful service). The second is the front-end service, or Gateway service, in charge of routing and data aggregation across multiple partitions or stateful service instances. That Gateway service also handles client-side communication with retry loops accessing the back-end service. It's called a Gateway service if you implement your custom service, or alternatively you can also use the out-of-the-box Service Fabric reverse proxy.

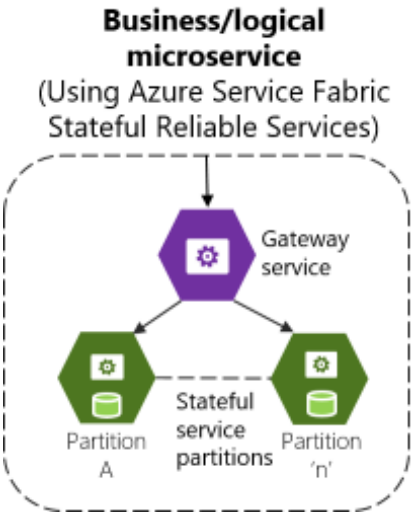

Figure 4-11. Business microservice with several stateful service instances and a custom gateway front end

In any case, when you use Service Fabric Stateful Reliable Services, you also have a logical or business microservice (Bounded Context) that's composed of multiple physical services. Each of them, the Gateway service, and Partition service could be implemented as ASP.NET Web API services, as shown in Figure 4-11. Service Fabric has prescription to support several stateful reliable services in containers.

In Service Fabric, you can group and deploy groups of services as a Service Fabric Application, which is the unit of packaging and deployment for the orchestrator or cluster. Therefore, the Service Fabric Application could be mapped to this autonomous business and logical microservice boundary or Bounded Context, as well, so you could deploy these services autonomously.

Service Fabric and containers

With regard to containers in Service Fabric, you can also deploy services in container images within a Service Fabric cluster. As Figure 4-12 shows, most of the time there will only be one container per service.

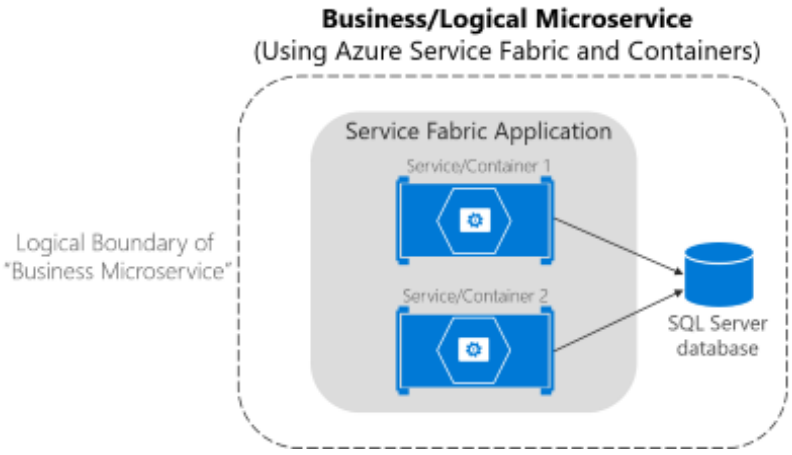

Figure 4-12. Business microservice with several services (containers) in Service Fabric

A Service Fabric application can run several containers accessing an external database and the whole set would be the logical boundary of a Business Microservice. However, so-called "sidecar" containers (two containers that must be deployed together as part of a logical service) are also possible in Service Fabric. The important thing is that a business microservice is the logical boundary around several cohesive elements. In many cases, it might be a single service with a single data model, but in some other cases you might have several physical services as well.

Note that you can mix services in processes, and services in containers, in the same Service Fabric application, as shown in Figure 4-13.

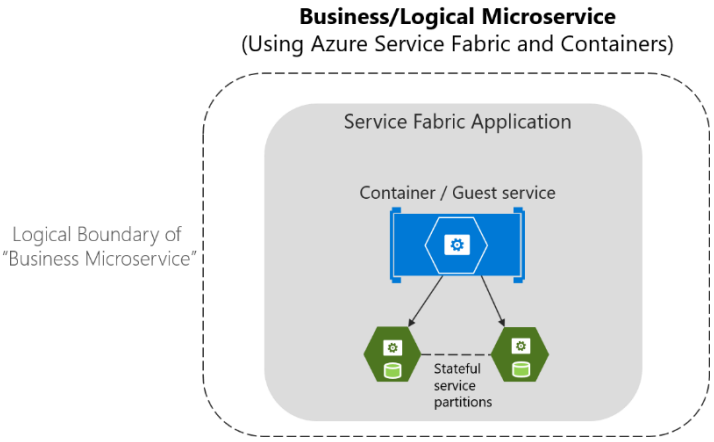

Figure 4-13. Business microservice mapped to a Service Fabric application with containers and stateful services

For more information about container support in Azure Service Fabric, see Service Fabric and containers.

Stateless versus stateful microservices

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

But the services themselves can also be stateful in Service Fabric, which means that the data resides within the microservice. This data might exist not just on the same server, but within the microservice process, in memory and persisted on hard drives and replicated to other nodes. Figure 4-14 shows the different approaches.

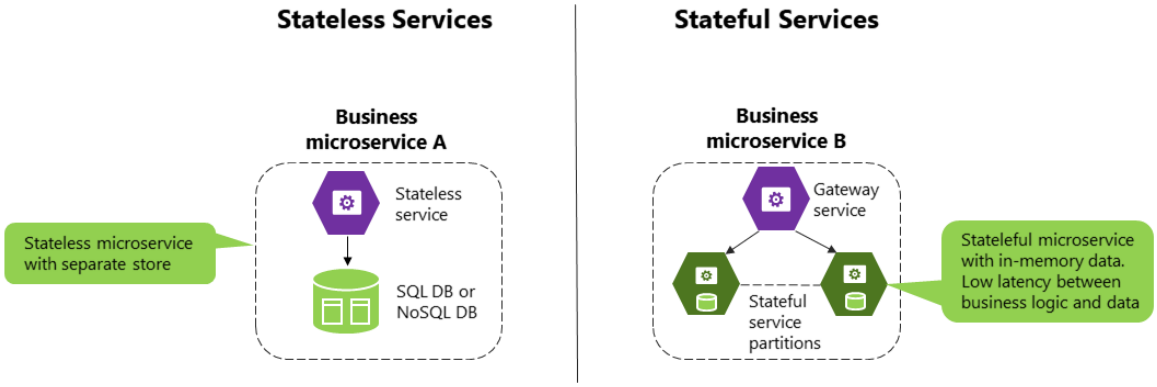

Figure 4-14. Stateless versus stateful microservices

In stateless services, the state (persistence, database) is kept out of the microservice. In stateful services, state is kept inside the microservice. A stateless approach is perfectly valid and is easier to implement than stateful microservices, since the approach is similar to traditional and well-known patterns. But stateless microservices impose latency between the process and data sources. They also involve more moving pieces when you're trying to improve performance with additional cache and queues. The result is that you can end up with complex architectures that have too many tiers.

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

Stateless and stateful services are complementary. For instance, as you can see in the right diagram in Figure 4-14, a stateful service can be split into multiple partitions. To access those partitions, you might need a stateless service acting as a gateway service that knows how to address each partition based on partition keys.

Stateful services do have drawbacks. They impose a high complexity level to be scaled out. Functionality that would usually be implemented by external database systems must be

addressed for tasks such as data replication across stateful microservices and data partitioning. However, this is one of the areas where an orchestrator like Azure Service Fabric with its stateful reliable services can help the most—by simplifying the development and lifecycle of stateful microservices using the Reliable Services API and Reliable Actors.

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

Remember that Docker containers are themselves stateless. If you want to implement a stateful service, you need one of the additional prescriptive and higher-level frameworks noted earlier.

Using Azure Service Fabric Mesh

Azure Service Fabric Mesh is a fully managed service that enables developers to build and deploy mission critical applications without managing any infrastructure. Use Service Fabric Mesh to build and run secure, distributed microservices applications that scale on demand.

As shown in figure 4-15, applications hosted on Service Fabric Mesh run and scale without you worrying about the infrastructure powering it.

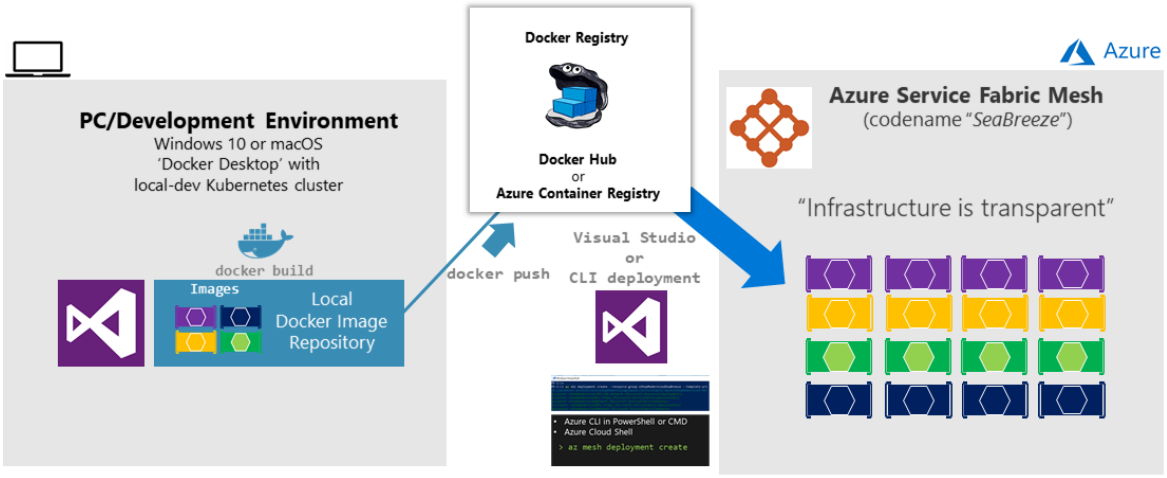

Figure 4-15. Deploying a microservice/containers application to Service Fabric Mesh

Under the covers, Service Fabric Mesh consists of clusters of thousands of machines. All cluster operations are hidden from the developer. You simply need to upload your containers and specify resources you need, availability requirements, and resource limits. Service Fabric Mesh automatically allocates the infrastructure requested by your application deployment and also handles infrastructure failures, making sure your applications are highly

available. You only need to care about the health and responsiveness of your application, not the infrastructure.

For further information, see the Service Fabric Mesh documentation.

Choosing orchestrators in Azure

The following table provides guidance on what orchestrator to use depending on workloads and OS focus.

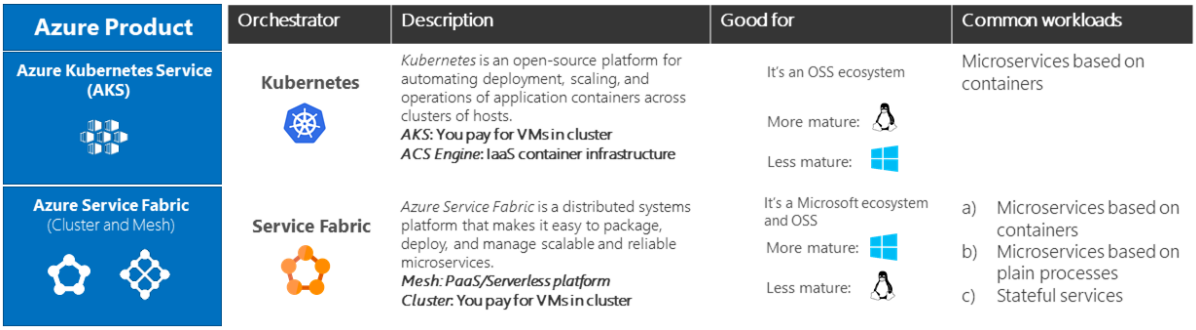

Figure 4-16. Orchestrator selection in Azure guidance

Deploy to Azure Kubernetes Service (AKS)

You can interact with AKS using your preferred client operating system (Windows, macOS, or Linux) with Azure command-line interface (Azure CLI) installed. For more details, refer Azure CLI documentation and Installation guide for the available environments.

Create the AKS environment in Azure

There are several ways to create the AKS Environment. It can be done by using Azure CLI commands or by using the Azure portal.

Here you can explore some examples using the Azure CLI to create the cluster and the Azure portal to review the results. You also need to have Kubectl and Docker in your development machine.

Create the AKS cluster

Create the AKS cluster using this command (the resource group must exist):

```
az aks create --resource-group explore-docker-aks-rg --name explore-docker-aks --node-vm-size Standard_B2s --node-count 1 --generate-ssh-keys --location westeurope
```

Note

The `--node-vm-size` and `--node-count` parameter values are good enough for a sample/dev application.

After the creation job finishes, you can see:

- The AKS cluster created in the initial resource group
- A new, related resource group, containing the resources related to the AKS cluster, as show in the following images.

The initial resource group, with the AKS cluster:

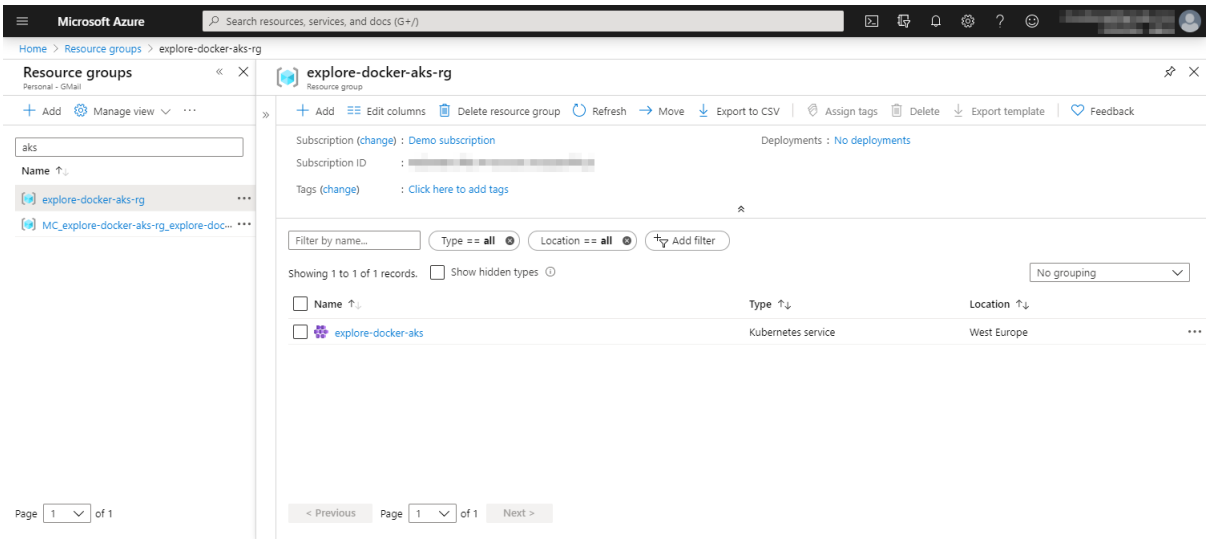

Figure 4-17. AKS Resource Group view from Azure.

The AKS cluster resource group:

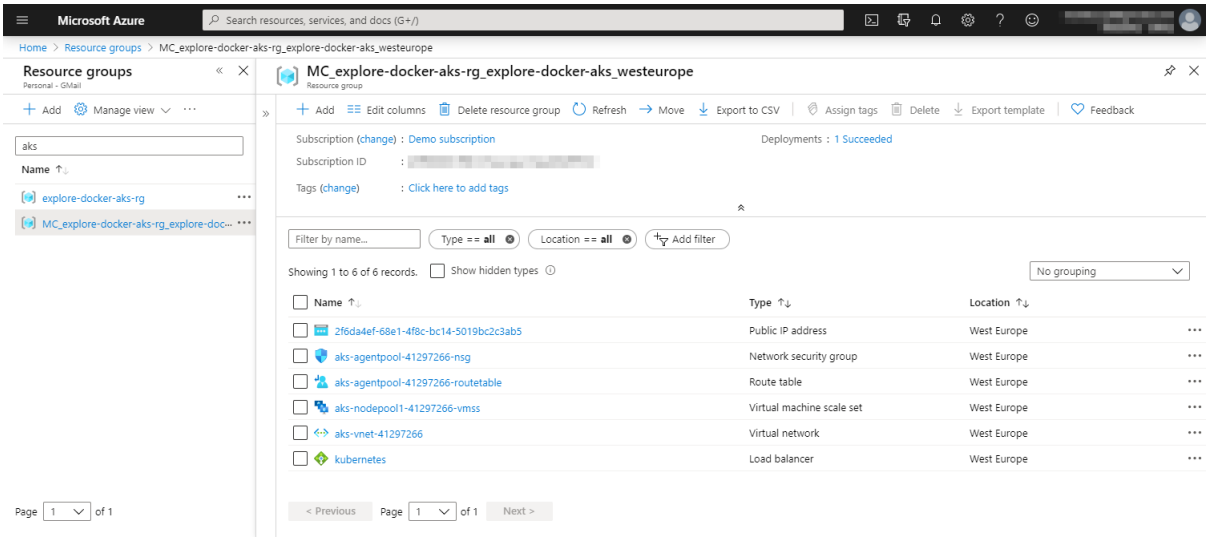

Figure 4-18. AKS view from Azure.

Important

In general, you shouldn't need to modify the resources in the AKS cluster resource group. For example, the resource group is deleted when you delete the AKS cluster.

You can also view the node created using `Azure CLI` and `Kubectl`.

First, getting the credentials:

```
az aks get-credentials --resource-group explore-docker-aks-rg --name explore-docker-aks
```

```
miguel@LAPTOP-MV:/mnt/c/Users/Miguel$ az aks get-credentials --resource-group explore-docker-aks-rg --name explore-docker-aks
Merged "explore-docker-aks" as current context in /home/miguel/.kube/config
miguel@LAPTOP-MV:/mnt/c/Users/Miguel$
```

Figure 4-19. aks get-credentials command result.

And then, getting nodes from Kubectl:

```
kubectl get nodes
```

```
miguel@LAPTOP-MV:/mnt/c/Users/Miguel$ kubectl get nodes
NAME                                STATUS   ROLES   AGE   VERSION
aks-nodepool1-41297266-vmss000000   Ready    agent   36m   v1.15.10
miguel@LAPTOP-MV:/mnt/c/Users/Miguel$
```

Figure 4-20. kubectl get nodes command result.

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, Microsoft has you covered when it comes to developing Docker applications.

Visual Studio Code and Docker CLI (cross-platform tools for Mac, Linux, and Windows)

If you prefer a lightweight, cross-platform editor supporting any development language, you can use 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 (runtime environment). Plus, Visual Studio Code supports extensions for Docker with IntelliSense for Dockerfiles and shortcut-tasks to run Docker commands from the editor.

Note

To download Visual Studio Code, go to https://code.visualstudio.com/download.

To download Docker for Mac and Windows, go to https://www.docker.com/products/docker.

Visual Studio with Docker Tools (Windows development machine)

It's recommended that you use Visual Studio 2022 or later with the built-in Docker Tools enabled. With Visual Studio, you can develop, run, and validate your applications directly in the chosen Docker environment. Press F5 to debug your application (single container or multiple containers) directly in a Docker host, or press Ctrl+F5 to edit and refresh your app without having to rebuild the container. It's the simplest and most powerful choice for Windows developers to create Docker containers for Linux or Windows.

Visual Studio for Mac (Mac development machine)

You can use Visual Studio for Mac when developing Docker-based applications. Visual Studio for Mac offers a richer IDE when compared to Visual Studio Code for Mac.

Language and framework choices

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

- .NET and ASP.NET Core
- Node.js
- Go
- 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 entire DevOps cycle, it all begins on each developer's machine, coding the app itself, using their preferred languages or platforms, and testing it locally (Figure 4-21). But in every case, you'll have an important point in common, no matter what language, framework, or platforms you choose. In this specific workflow, you're always developing and testing Docker containers in no other environments, but locally.

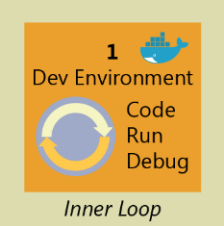

Figure 4-21. Inner-loop development context

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

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

You can set up the inner-loop development workflow that utilizes Docker as the process (described in the next section). Consider that the initial steps to set up the environment are not included, because you only need to do it once.

Building a single app within a Docker container using Visual Studio Code and Docker CLI

Apps are made up from your own services plus additional libraries (dependencies).

Figure 4-22 shows the basic steps that you usually need to carry out when building a Docker app, followed by detailed descriptions of each step.

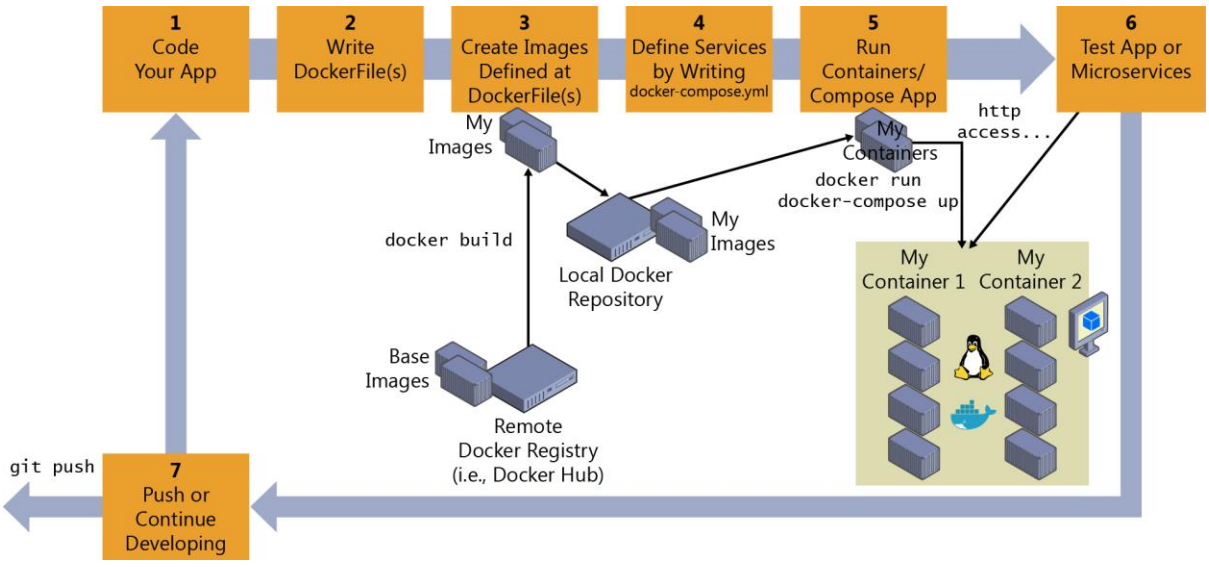

Figure 4-22. High-level workflow for the life cycle for Docker containerized applications using Docker CLI

Step 1: Start coding in Visual Studio Code and create your initial app/service baseline

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

Setting up your local environment

With the latest versions of Docker Desktop for Mac and Windows, it's easier than ever to develop Docker applications, and the setup is straightforward.

> **Tip**
>
> For instructions on setting up Docker Desktop for Windows, go to https://docs.docker.com/docker-for-windows/.
>
> For instructions on setting up Docker Desktop for Mac, go to https://docs.docker.com/docker-for-mac/.

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

Microsoft provides Visual Studio Code, which is a lightweight code editor that's supported on Windows, Linux, and macOS, and provides IntelliSense with support for many languages (JavaScript, .NET, Go, Java, Ruby, Python, and most modern languages), debugging, integration with Git and extensions support. This editor is a great fit for macOS and Linux developers. In Windows, you also can use Visual Studio.

> **Tip**
>
> For instructions on installing Visual Studio Code for Windows, Linux, or macOS, go to https://code.visualstudio.com/docs/setup/setup-overview/.
>
> For instructions on setting up Docker for Mac, go to https://docs.docker.com/docker-for-mac/.

You can work with Docker CLI and write your code using any code editor, but using Visual Studio Code with the Docker extension makes it easy to author `Dockerfile` and `docker-compose.yml` files in your workspace. You can also run tasks and scripts from the Visual Studio Code IDE to execute Docker commands using the Docker CLI underneath.

The Docker extension for VS Code provides the following features:

- Automatic `Dockerfile` and `docker-compose.yml` file generation
- Syntax highlighting and hover tips for `docker-compose.yml` and `Dockerfile` files

- IntelliSense (completions) for `Dockerfile` and `docker-compose.yml` files
- Linting (errors and warnings) for `Dockerfile` files
- Command Palette (F1) integration for the most common Docker commands
- Explorer integration for managing Images and Containers
- Deploy images from DockerHub and Azure Container Registries to Azure App Service

To install the Docker extension, press Ctrl+Shift+P, type `ext install`, and then run the Install Extension command to bring up the Marketplace extension list. Next, type **docker** to filter the results, and then select the Docker Support extension, as depicted in Figure 4-23.

Figure 4-23. Installing the Docker Extension in Visual Studio Code

Step 2: Create a DockerFile related to an existing image (plain OS or dev environments like .NET, Node.js, and Ruby)

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

The `DockerFile` is commonly placed in the root folder of your app or service and contains the required commands so that Docker knows how to set up and run that app or service. You can create your `DockerFile` and add it to your project along with your code (node.js, .NET, etc.), or, if you're new to the environment, take a look at the following Tip.

Tip

You can use the Docker extension to guide you when using the `Dockerfile` and `docker-compose.yml` files related to your Docker containers. Eventually, you'll probably write these kinds of files without this tool, but using the Docker extension is a good starting point that will accelerate your learning curve.

In Figure 4-24, you can see the steps to add the docker files to a project by using the Docker Extension for VS Code:

1. Open the command palette, type "**docker**" and select "**Add Docker Files to Workspace**".

2. Select Application Platform (ASP.NET Core)
3. Select Operating System (Linux)
4. Include optional Docker Compose files
5. Enter ports to publish (80, 443)
6. Select the project

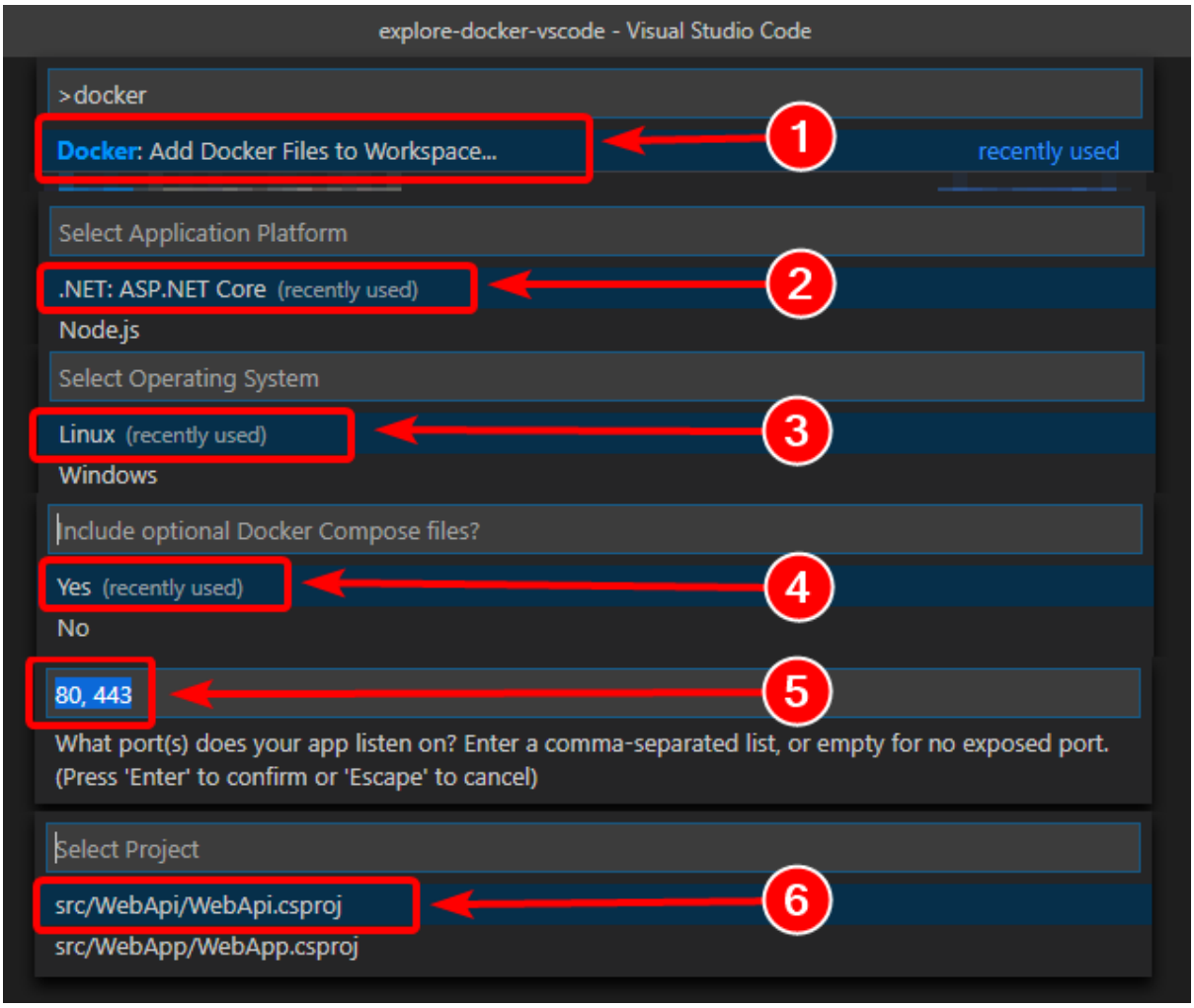

Figure 4-24. Docker files added using the Add Docker files to Workspace command

When you add a DockerFile, you specify what base Docker image you'll be using (like using `FROM mcr.microsoft.com/dotnet/aspnet`). You'll usually build your custom image on top of a base image that you get from any official repository at the Docker Hub registry (like an image for .NET or the one for Node.js).

Tip

You'll have to repeat this procedure for every project in your application. However, the extension will ask to overwrite the generated docker-compose file after the first time. You should reply to not overwrite it, so the extension creates separate docker-compose files, that you can then merge by hand, prior to running docker-compose.

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).

The following is a sample DockerFile for a .NET container:

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app
EXPOSE 80
EXPOSE 443

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY ["src/WebApi/WebApi.csproj", "src/WebApi/"]
RUN dotnet restore "src/WebApi/WebApi.csproj"
COPY . .
WORKDIR "/src/src/WebApi"
RUN dotnet build "WebApi.csproj" -c Release -o /app/build

FROM build AS publish
RUN dotnet publish "WebApi.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "WebApi.dll"]
```

In this case, the image is based on version 6.0 of the official ASP.NET Core Docker image (multi-arch for Linux and Windows), as per the line `FROM mcr.microsoft.com/dotnet/aspnet:6.0`. (For more information about this topic, see the ASP.NET Core Docker Image page and the .NET Docker Image page).

In the DockerFile, you can also instruct Docker to listen to the TCP port that you'll use at run time (such as port 80 or 443).

You can specify additional configuration settings in the Dockerfile, depending on the language and framework you're using. For instance, the `ENTRYPOINT` line with `["dotnet", "WebMvcApplication.dll"]` tells Docker to run a .NET application. If you're using the SDK and the .NET CLI (`dotnet CLI`) to build and run the .NET application, this setting would be different. The key point here is that the ENTRYPOINT line and other settings depend on the language and platform you choose for your application.

Tip

For more information about building Docker images for .NET applications, go to https://docs.microsoft.com/dotnet/core/docker/building-net-docker-images.

To learn more about building your own images, go to https://docs.docker.com/engine/tutorials/dockerimages/.

Use multi-arch image repositories

A single image name in a repo can contain platform variants, such as a Linux image and a Windows image. This feature allows vendors like Microsoft (base image creators) to create a single repo to cover multiple platforms (that is, Linux and Windows). For example, the dotnet/aspnet repository available in the Docker Hub registry provides support for Linux and Windows Nano Server by using the same image name.

Pulling the dotnet/aspnet image from a Windows host pulls the Windows variant, whereas pulling the same image name from a Linux host pulls the Linux variant.

Create your base image from scratch

You can create your own Docker base image from scratch as explained in this article from Docker. This scenario is probably not the best for you if you're just 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

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

Note

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 you consider going to that outer-loop route, you need to ensure that the Docker application is actually working properly so that they don't push code that might not work properly to the source control system (Git, etc.).

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

To create an image in your local environment and using the DockerFile, you can use the docker build command, as shown in Figure 4-25, because it already tags the image for you and builds the images for all services in the application with a simple command.

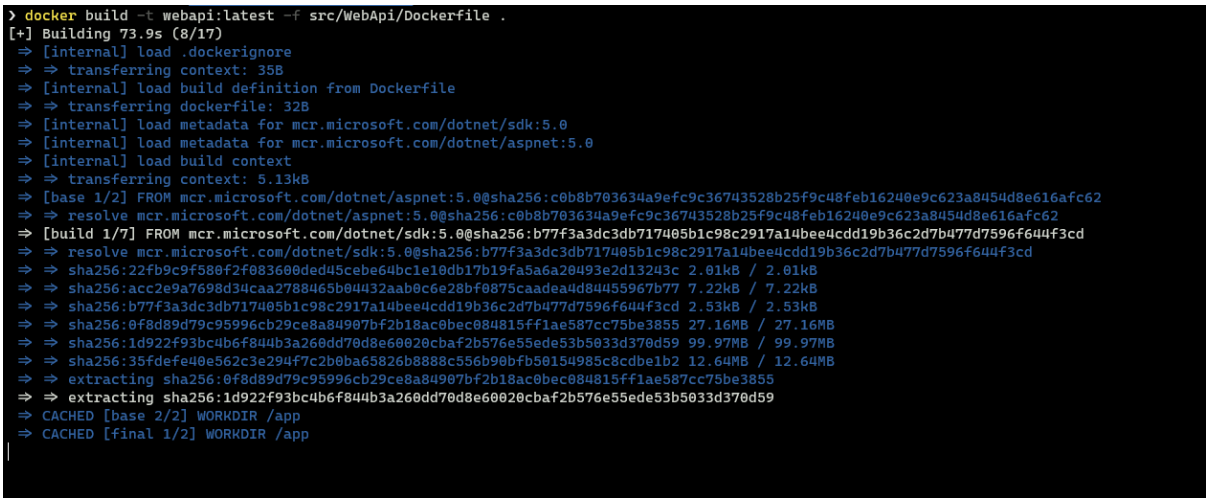

Figure 4-25. Running docker build

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

This example creates a Docker image with the name `webapi:latest` (`:latest` 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. However, we'll see in the next section that it's easier to do this using `docker-compose`.

You can find the existing images in your local repository (your development machine) by using the `docker images` command, as illustrated in Figure 4-26.

Figure 4-26. Viewing existing images using docker images

Step 4: 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.

Create that file in your main or root solution folder; it should have content similar to that shown in this `docker-compose.yml` file:

```
version: "3.4"

services:
  webapi:
    image: webapi
    build:
      context: .
      dockerfile: src/WebApi/Dockerfile
    ports:
      - 51080:80
    depends_on:
      - redis
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:80

  webapp:
    image: webapp
    build:
      context: .
      dockerfile: src/WebApp/Dockerfile
    ports:
      - 50080:80
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=http://+:80
      - WebApiBaseAddress=http://webapi

  redis:
    image: redis
```

In this particular case, this file defines three services: the web API service (your custom service), a web application, and the Redis service (a popular cache service). Each service will be deployed as a container, so you need to use a concrete Docker image for each. For this particular application:

- The web API service is built from the DockerFile in the `src/WebApi/Dockerfile` directory.
- The host port 51080 is forwarded to the exposed port 80 on the `webapi` container.
- The web API service depends on the Redis service
- The web application accesses the web API service using the internal address: `http://webapi`.
- The Redis service uses the latest public redis image pulled from the Docker Hub registry. Redis is a popular cache system for server-side applications.

Step 5: Build and run your Docker app

If your app has only 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 up of multiple services, you need to *compose it*, too. Let's see the different options.

Option A: Run a single container or service

You can run the Docker image by using the docker run command, as shown here:

```
docker run -t -d -p 50080:80 webapp:latest
```

For this particular deployment, we'll be redirecting requests sent to port 50080 on the host to the internal port 80.

Option B: Compose and run a multiple-container application

In most enterprise scenarios, a Docker application will be composed of multiple services. For these cases, you can run the `docker-compose up` command (Figure 4-27), which will use the docker-compose.yml file that you created previously. Running this command builds all custom images and deploys the composed application with all of its related containers.

```
> docker-compose up
Creating explore-docker-vscode_webapi_1  ... done
Creating explore-docker-vscode_webapp_1  ... done
Attaching to explore-docker-vscode_webapp_1, explore-docker-vscode_webapi_1
webapp_1  | warn: Microsoft.AspNetCore.DataProtection.Repositories.FileSystemXmlRepository[60]
webapp_1  |       Storing keys in a directory '/root/.aspnet/DataProtection-Keys' that may not be p
ected data will be unavailable when container is destroyed.
webapp_1  | warn: Microsoft.AspNetCore.DataProtection.KeyManagement.XmlKeyManager[35]
webapp_1  |       No XML encryptor configured. Key {19eedc7c-d3d0-4443-af35-93d4d078119c} may be pe

webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Now listening on: http://[::]:80
webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Application started. Press Ctrl+C to shut down.
webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Hosting environment: Development
webapi_1  | info: Microsoft.Hosting.Lifetime[0]
webapi_1  |       Content root path: /app
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Now listening on: http://[::]:80
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Application started. Press Ctrl+C to shut down.
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Hosting environment: Development
webapp_1  | info: Microsoft.Hosting.Lifetime[0]
webapp_1  |       Content root path: /app
```

Figure 4-27. Results of running the "docker-compose up" command

After you run `docker-compose up`, you deploy your application and its related container(s) into your Docker Host, as illustrated in Figure 4-28, in the VM representation.

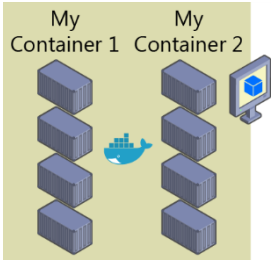

Figure 4-28. VM with Docker containers deployed

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

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

In a simple .NET Web API "Hello World" deployed as a single container or service, you'd just need to access the service by providing the TCP port specified in the DockerFile.

On the Docker host, open a browser and navigate to that site; you should see your app/service running, as demonstrated in Figure 4-29.

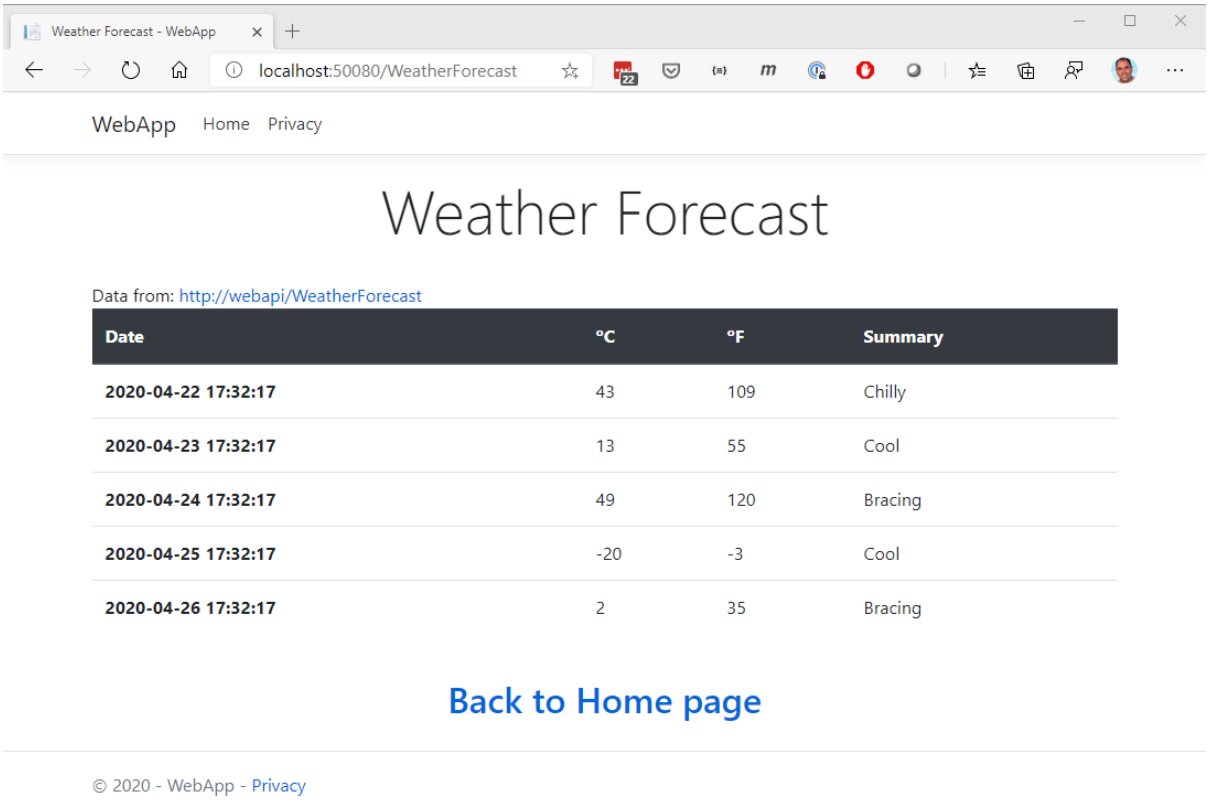

Figure 4-29. Testing your Docker application locally by using the browser

Note that it's using port 50080, but internally it's being redirected to port 80, because that's how it was deployed with `docker compose`, as explained earlier.

You can test this by using the browser using CURL from the terminal, as depicted in Figure 4-30.

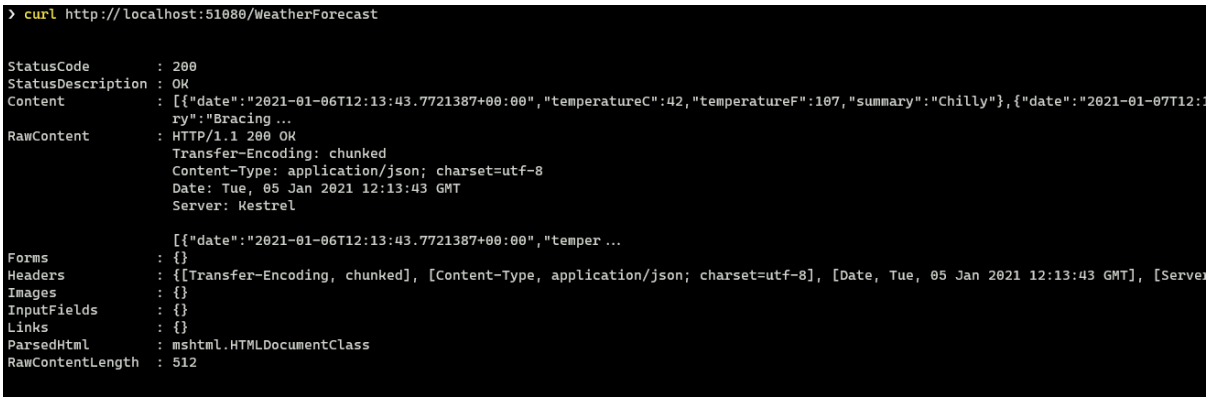

Figure 4-30. Testing a Docker application locally by using CURL

Debugging a container running on Docker

Visual Studio Code supports debugging Docker if you're using Node.js and other platforms like .NET containers.

You also can debug .NET or .NET Framework containers in Docker when using Visual Studio for Windows or Mac, as described in the next section.

Tip

To learn more about debugging Node.js Docker containers, see https://blog.docker.com/2016/07/live-debugging-docker/ and https://docs.microsoft.com/archive/blogs/user_ed/visual-studio-code-new-features-13-big-debugging-updates-rich-object-hover-conditional-breakpoints-node-js-mono-more.

Use Docker Tools in Visual Studio on Windows

The developer workflow when using the Docker Tools included in Visual Studio 2022 version 17.0 and later, is similar to using Visual Studio Code and Docker CLI (in fact, it's based on the same Docker CLI), but it's easier to get started, simplifies the process, and provides greater productivity for the build, run, and compose tasks. It can also run and debug your containers via the usual `F5` and `Ctrl+F5` keys from Visual Studio. You can even debug a whole solution if its containers are defined in the same `docker-compose.yml` file at the solution level.

Configure your local environment

With the latest versions of Docker for Windows, it's easier than ever to develop Docker applications because the setup is straightforward, as explained in the following references.

Tip

To learn more about installing Docker for Windows, go to (https://docs.docker.com/docker-for-windows/).

Docker support in Visual Studio

There are two levels of Docker support you can add to a project. In ASP.NET Core projects, you can just add a `Dockerfile` file to the project by enabling Docker support. The next level is container orchestration support, which adds a `Dockerfile` to the project (if it doesn't already exist) and a `docker-compose.yml` file at the solution level. Container orchestration support, via Docker Compose, is available in Visual Studio 2022 versions 17.0. Container orchestration support is an opt-in feature in Visual Studio 2022 versions 17.0 or later. Visual Studio 2022 also supports **Kubernetes/Helm** deployment.

The **Add > Docker Support** and **Add > Container Orchestrator Support** commands are located on the right-click menu (or context menu) of the project node for an ASP.NET Core project in **Solution Explorer**, as shown in Figure 4-31:

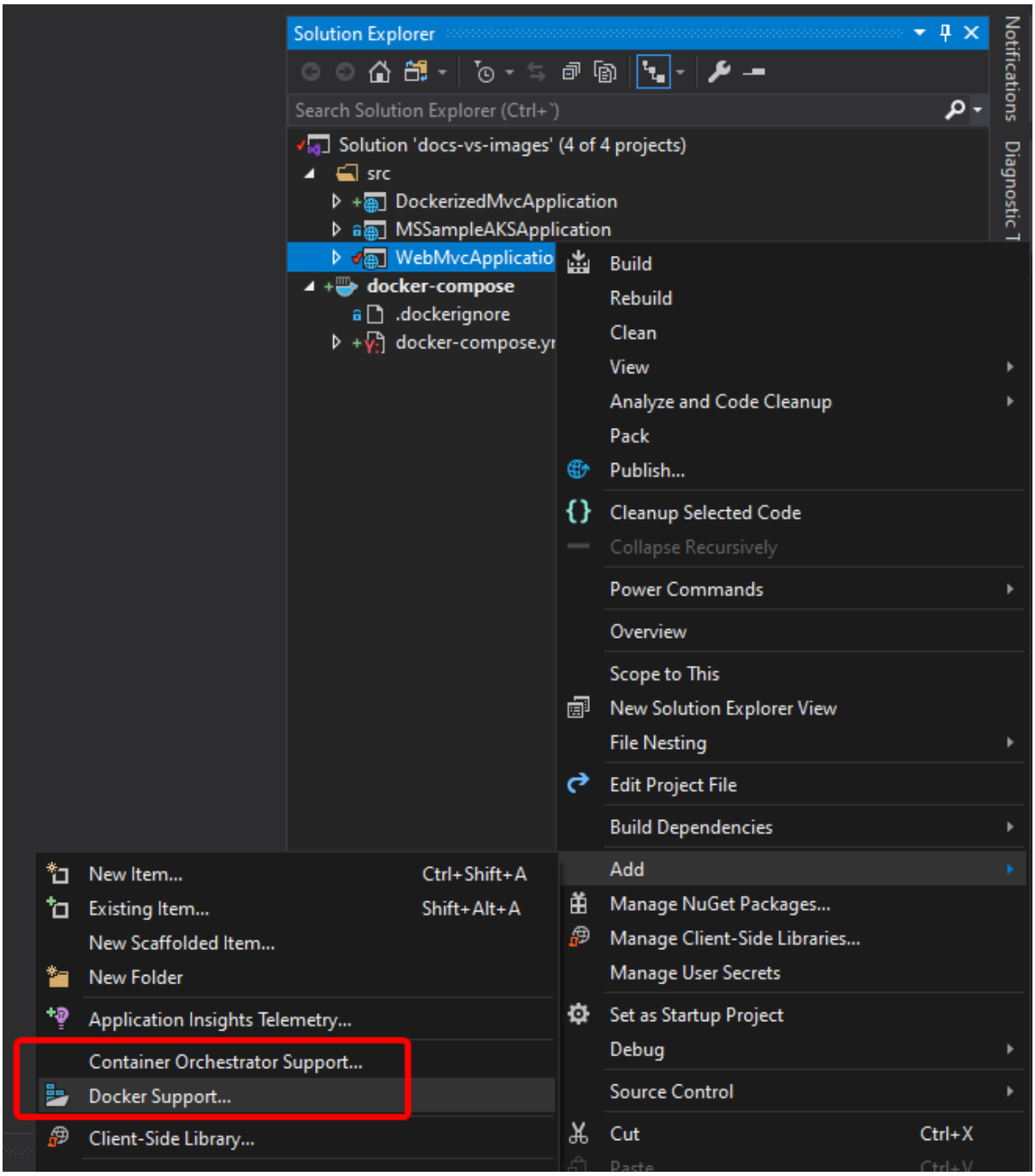

Figure 4-31. Adding Docker support to a Visual Studio project

Add Docker support

Besides the option to add Docker support to an existing application, as shown in the previous section, you can also enable Docker support during project creation by selecting **Enable Docker Support** in the **New ASP.NET Core Web Application** dialog box that opens after you click **OK** in the **New Project** dialog box, as shown in Figure 4-32.

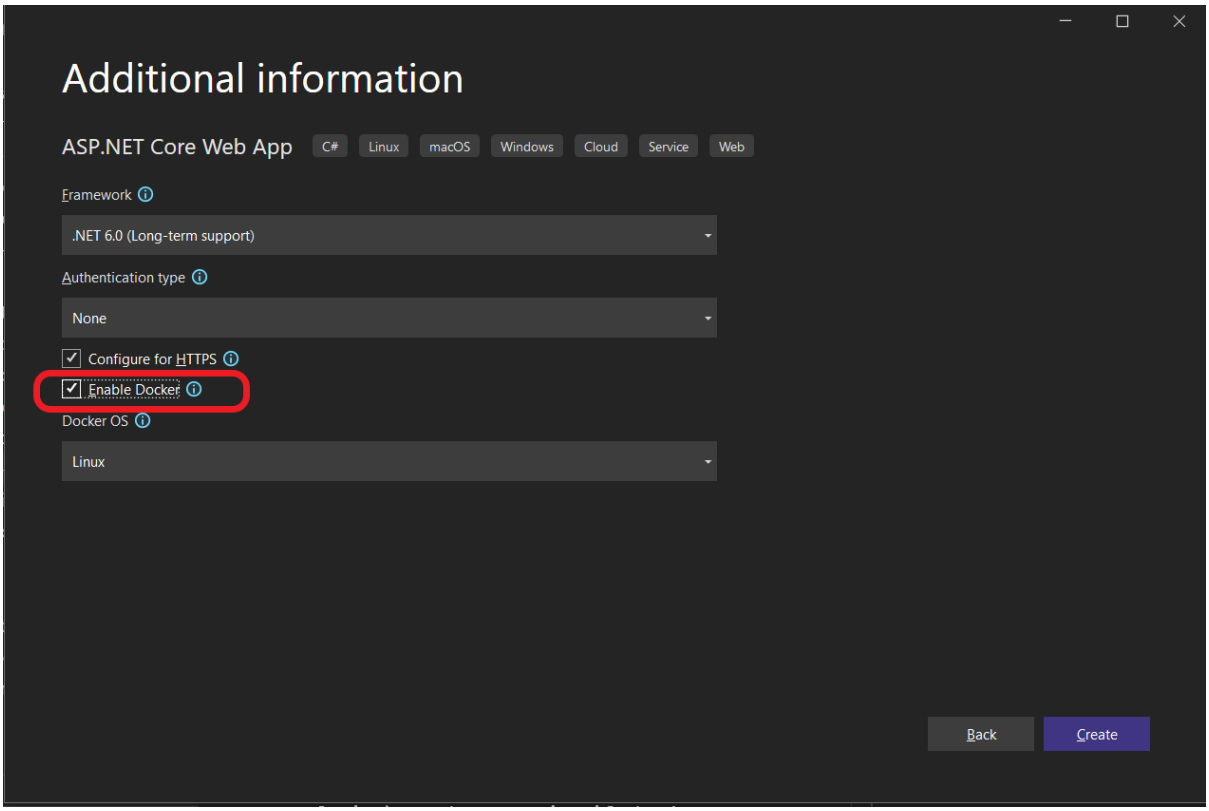

Figure 4-32. Enable Docker support during project creation in Visual Studio

When you add or enable Docker support, Visual Studio adds a *Dockerfile* file to the project, that includes references to all required project from the solution.

Add container orchestration support

When you want to compose a multi-container solution, add container orchestration support to your projects. This lets you run and debug a group of containers (a whole solution) at the same time if they're defined in the same *docker-compose.yml* file.

To add container orchestration support, right-click on the project node in **Solution Explorer**, and choose **Add > Container Orchestration Support**. Then choose **Docker Compose** to manage the containers.

After you add container orchestration support to your project, you see a Dockerfile added to the project and a **docker-compose** folder added to the solution in **Solution Explorer**, as shown in Figure 4-33:

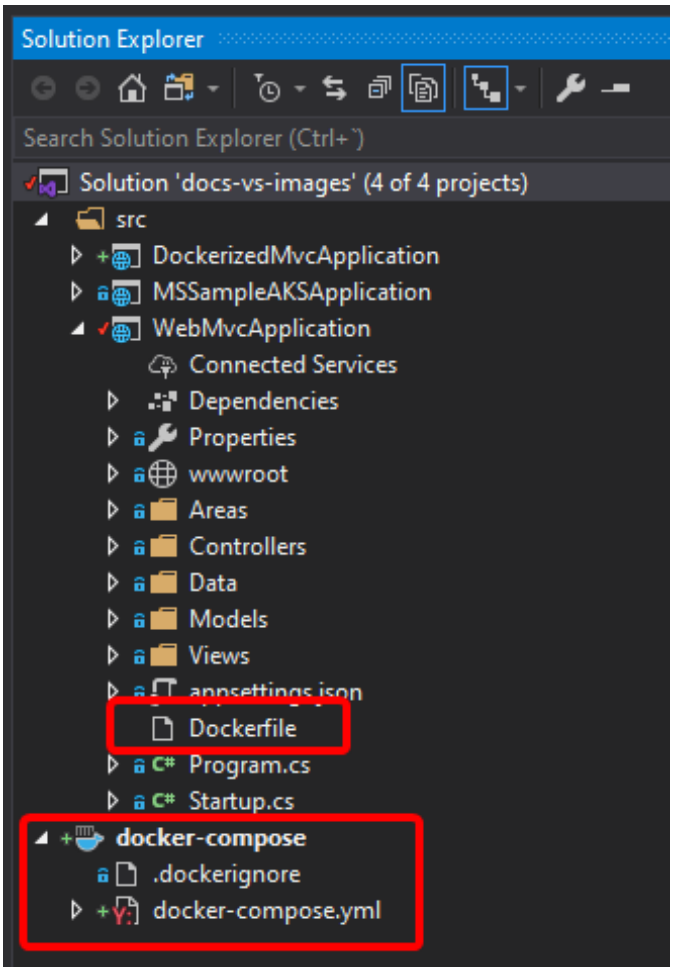

Figure 4-33. Docker files in Solution Explorer in Visual Studio

If *docker-compose.yml* already exists, Visual Studio just adds the required lines of configuration code to it.

Configure Docker tools

From the main menu, choose **Tools > Options**, and expand **Container Tools > Settings**. The container tools settings appear.

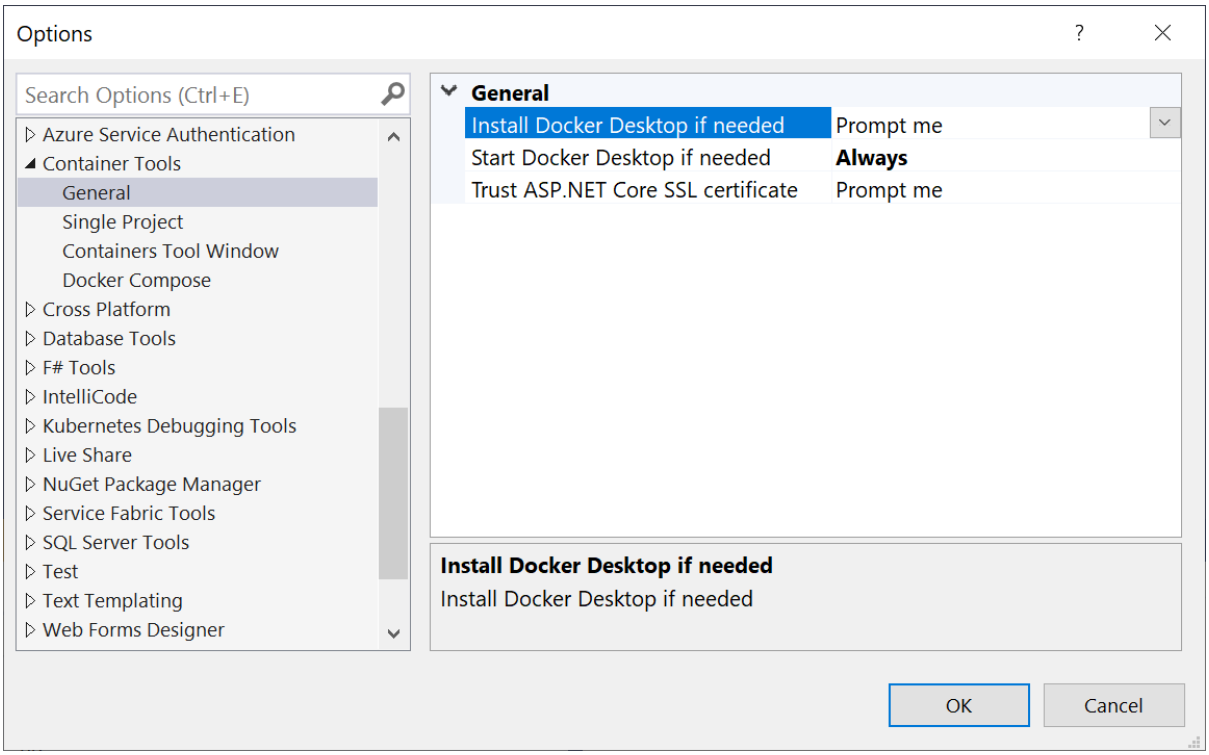

Figure 4-34. Docker Tools Options

For more detailed configurations refer to Container Tools settings

Tip

For further details on the services implementation and use of Visual Studio Tools for Docker, read the following articles:

Use the Containers tool window to view container details such as the filesystem, logs, environment, ports, and more: https://docs.microsoft.com/visualstudio/containers/view-and-diagnose-containers Debug apps in a local Docker container: https://docs.microsoft.com/visualstudio/containers/edit-and-refresh

Deploy an ASP.NET container to a container registry using Visual Studio: https://docs.microsoft.com/visualstudio/containers/hosting-web-apps-in-docker

Using Windows PowerShell commands in a DockerFile to set up Windows Containers (Docker standard based)

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

To use Windows Containers, you just need to write Windows PowerShell commands in the DockerFile, as demonstrated in the following example:

```
FROM mcr.microsoft.com/windows/servercore:ltsc2019
LABEL Description="IIS" Vendor="Microsoft" Version="10"
RUN powershell -Command Add-WindowsFeature Web-Server
CMD [ "ping", "localhost", "-t" ]
```

In this case, we're using Windows PowerShell to install a Windows Server Core base image as well as IIS.

In a similar way, you also could use Windows PowerShell commands to set up additional components like the traditional ASP.NET 4.x and .NET Framework 4.6 or any other Windows software, as shown here:

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

Build ASP.NET Core applications deployed as Linux containers into an AKS/Kubernetes orchestrator

Azure Kubernetes Services (AKS) is Azure's managed Kubernetes orchestrations services that simplify container deployment and management.

AKS main features are:

- An Azure-hosted control plane
- Automated upgrades
- Self-healing
- User-configurable scaling
- Simpler user experience for both developers and cluster operators.

The following examples explore the creation of an ASP.NET Core 6.0 application that runs on Linux and deploys to an AKS Cluster in Azure, while development is done using Visual Studio 2022 version 17.0.

Creating the ASP.NET Core Project using Visual Studio 2022

ASP.NET Core is a general-purpose development platform maintained by Microsoft and the .NET community on GitHub. It's cross-platform, supporting Windows, macOS and Linux, and can be used in device, cloud, and embedded/IoT scenarios.

This example uses a couple of simple projects based on Visual Studio templates, so you don't need much additional knowledge to create the sample. You only have to create the project using a standard template that includes all the elements to run a small project with a REST API and a Web App with Razor pages, using ASP.NET Core 6.0 technology.

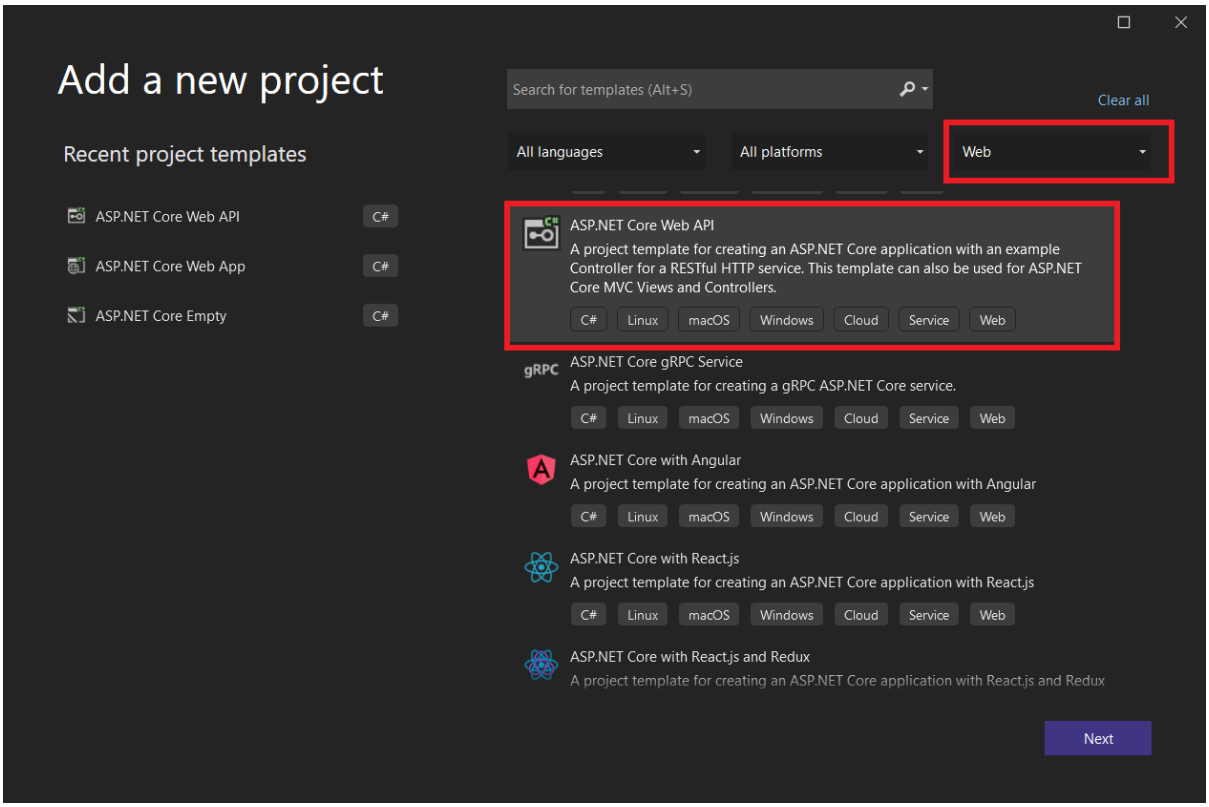

Figure 4-35. Creating an ASP.NET Core Web Application in Visual Studio 2022.

To create the sample project in Visual Studio, select **File** > **New** > **Project**, select the **Web** project type and then the **ASP.NET Core Web Application** template. You can also search for the template if you need it.

Then enter the application name and location as shown in the next image.

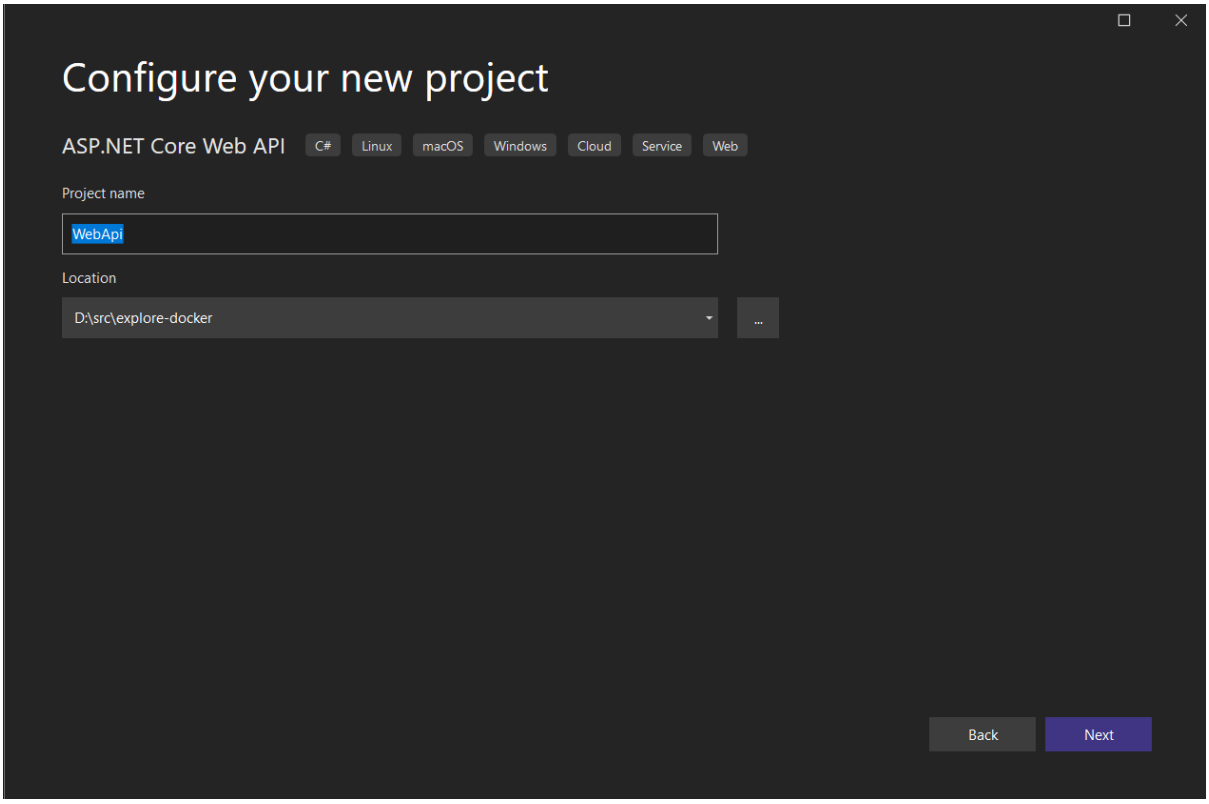

Figure 4-36. Enter the project name and location in Visual Studio 2022.

Verify that you've selected ASP.NET Core 6.0 as the framework. .NET 6 is included in the latest release of Visual Studio 2022 and is automatically installed and configured for you when you install Visual Studio.

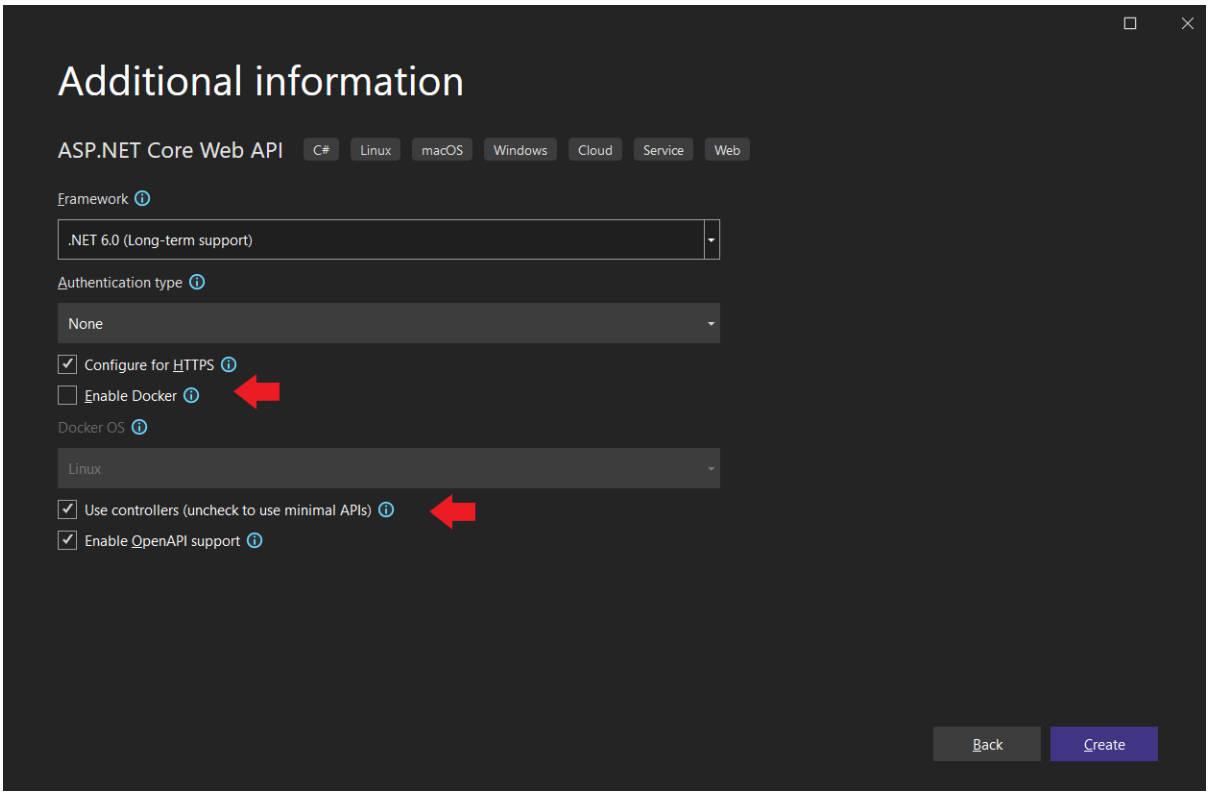

Figure 4-37. Selecting ASP.NET CORE 6.0 and Web API project type

Notice Docker support is not enabled now. You'll do that in the next step after the project creation. You'll also notice that by default controller option is checked. You can uncheck that if you want to Create a minimal web API with ASP.NET Core.

To show you can "Dockerize" your project at any time, you'll add Docker support now. So right-click on the project node in Solution Explorer and select **Add** > **Docker support** on the context menu.

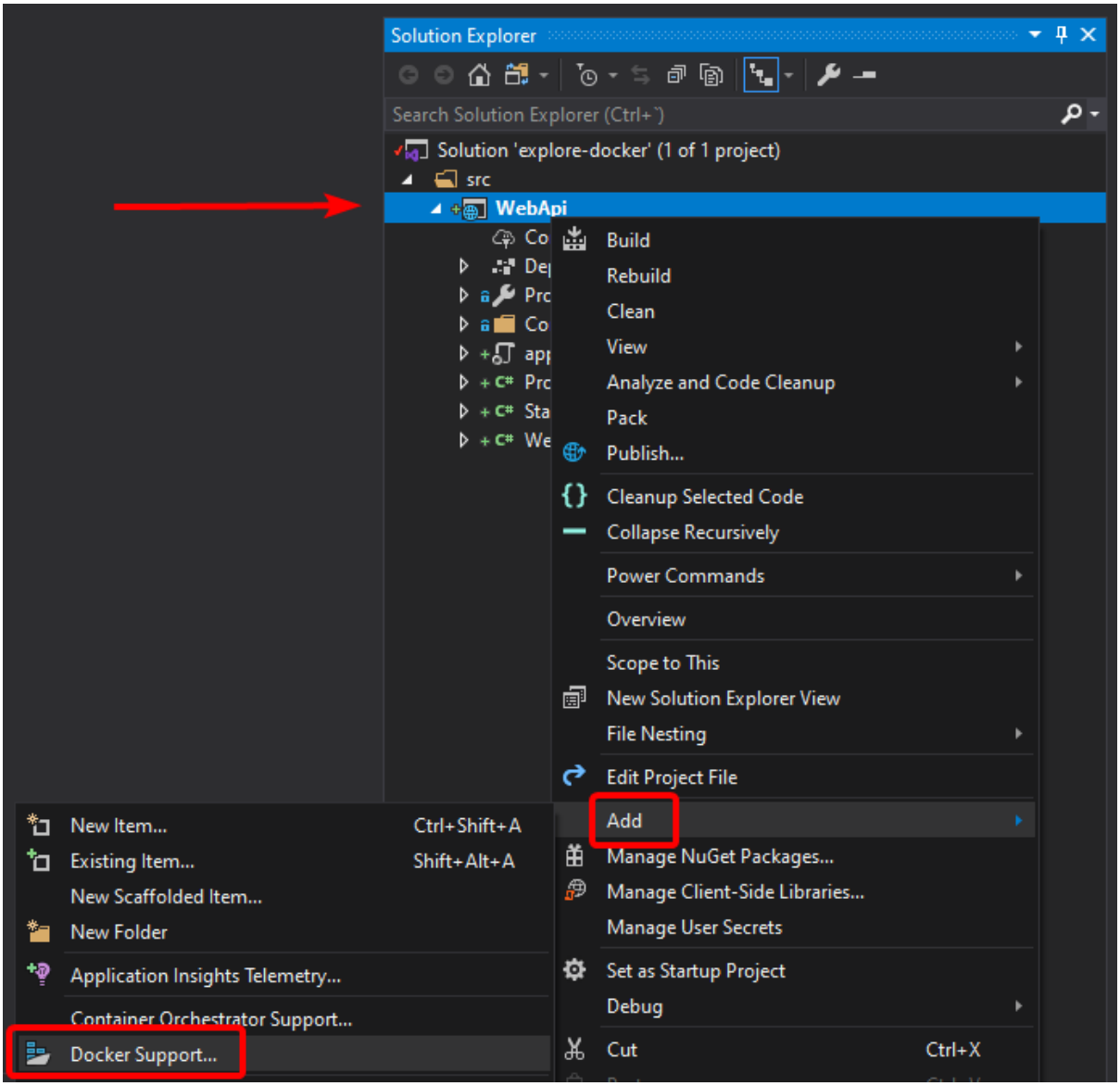

Figure 4-38. Adding Docker support to an existing project

To complete adding Docker support, you can choose Windows or Linux. In this case, select **Linux**.

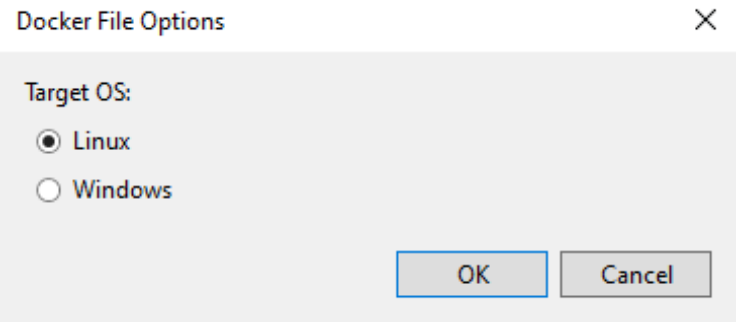

Figure 4-39. Selecting Linux containers.

With these simple steps, you have your ASP.NET Core 6.0 application running on a Linux container.

In a similar way, you can also add a very simple **WebApp** project (Figure 4-40) to consume the web API endpoint, although the details are not discussed here.

After that, you add orchestrator support for your **WebApi** project as shown next, in image 4-40.

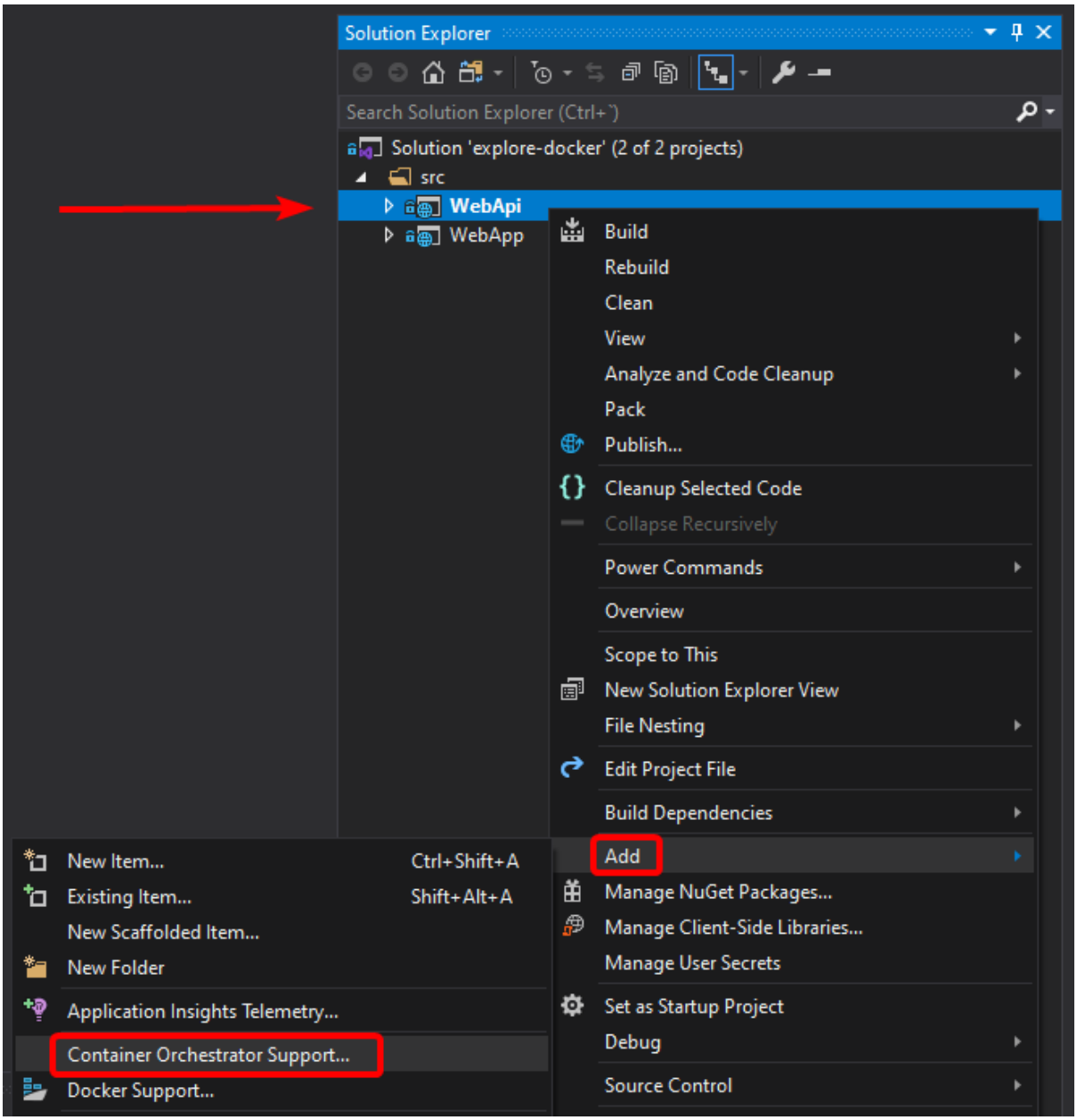

Figure 4-40. Adding orchestrator support to WebApi project.

When you choose the `Docker Compose` option, which is fine for local development, Visual Studio adds the docker-compose project, with the docker-compose files as shown in image 4-41.

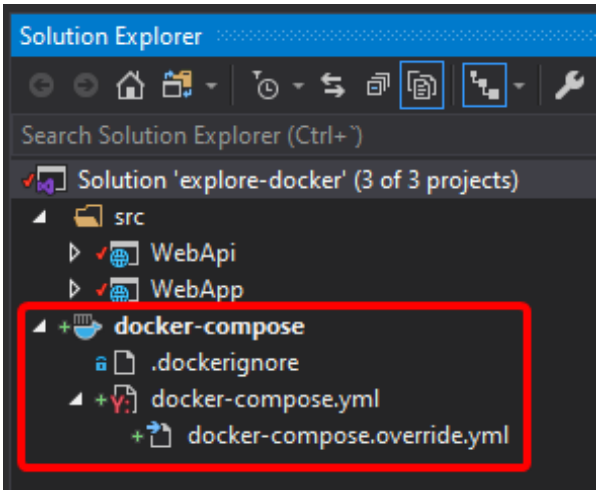

Figure 4-41. Adding orchestrator support to WebApi project.

The initial files added are similar to these ones:

`docker-compose.yml`

```
version: "3.4"

services:
  webapi:
    image: ${DOCKER_REGISTRY-}webapi
    build:
      context: .
      dockerfile: WebApi/Dockerfile

  webapp:
    image: ${DOCKER_REGISTRY-}webapp
    build:
      context: .
      dockerfile: WebApp/Dockerfile
```

`docker-compose.override.yml`

```
version: "3.4"

services:
  webapi:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
    ports:
      - "80"
      - "443"
    volumes:
      - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
      - ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro
  webapp:
    environment:
      - ASPNETCORE_ENVIRONMENT=Development
      - ASPNETCORE_URLS=https://+:443;http://+:80
    ports:
      - "80"
```

```
      - "443"
    volumes:
      - ${APPDATA}/Microsoft/UserSecrets:/root/.microsoft/usersecrets:ro
      - ${APPDATA}/ASP.NET/Https:/root/.aspnet/https:ro
```

To have you app running with Docker Compose you just have to make a few tweaks to `docker-compose.override.yml`

```
services:
  webapi:
    #...
    ports:
      - "51080:80"
      - "51443:443"
    #...
  webapp:
    environment:
      #...
      - WebApiBaseAddress=http://webapi
    ports:
      - "50080:80"
      - "50443:443"
    #...
```

Now you can run your application with the **F5** key, or by using the **Play** button, or the **Ctrl+F5** key, selecting the docker-compose project, as shown in image 4-42.

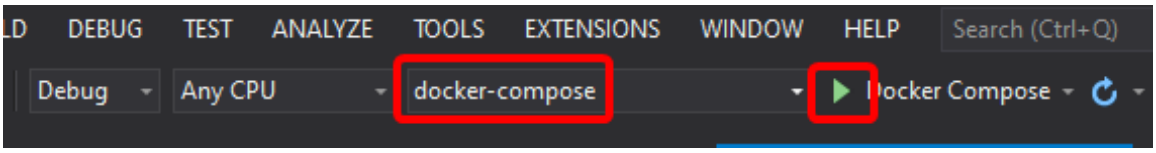

Figure 4-42. Adding orchestrator support to WebApi project.

When running the docker-compose application as explained, you get:

1. The images built and containers created as per the docker-compose file.
2. The browser open in the address configured in the "Properties" dialog for the `docker-compose` project.
3. The **Container** window open (in Visual Studio 2022 version 17.0 and later).
4. Debugger support for all projects in the solution, as shown in the following images.

Browser opened:

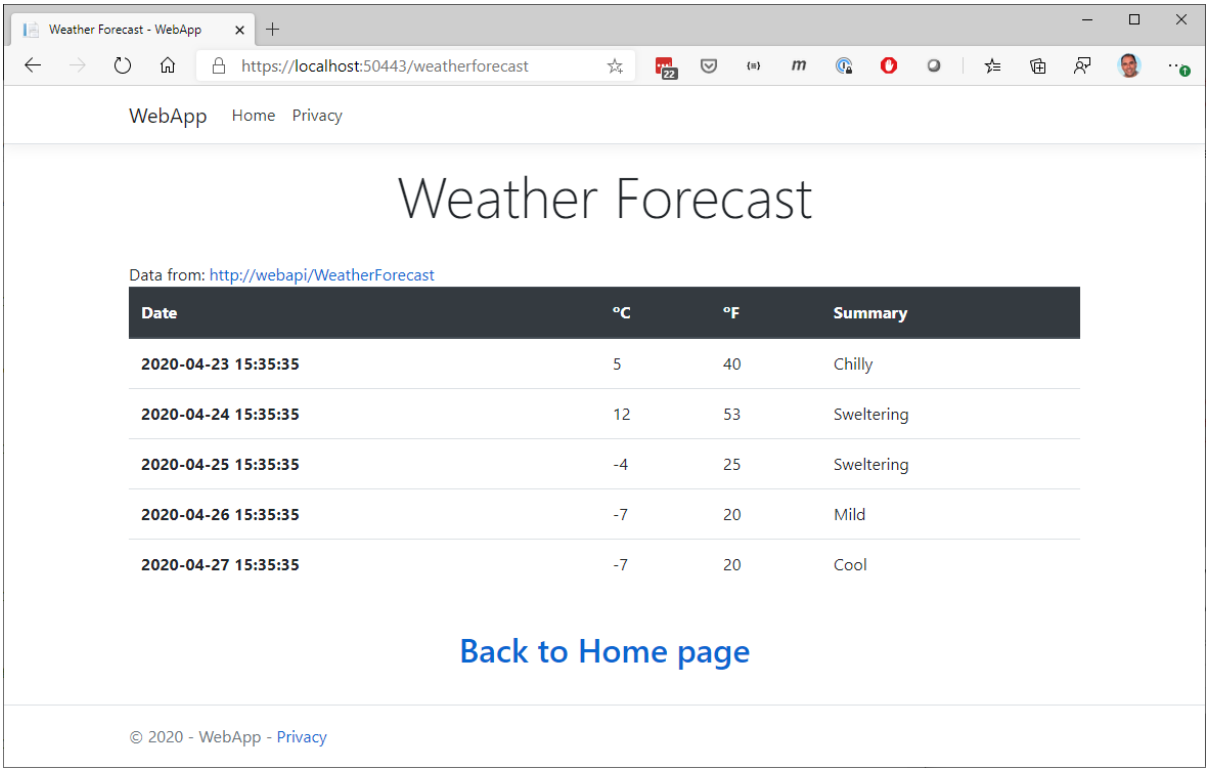

Figure 4-43. Browser window with an application running on multiple containers.

Containers window:

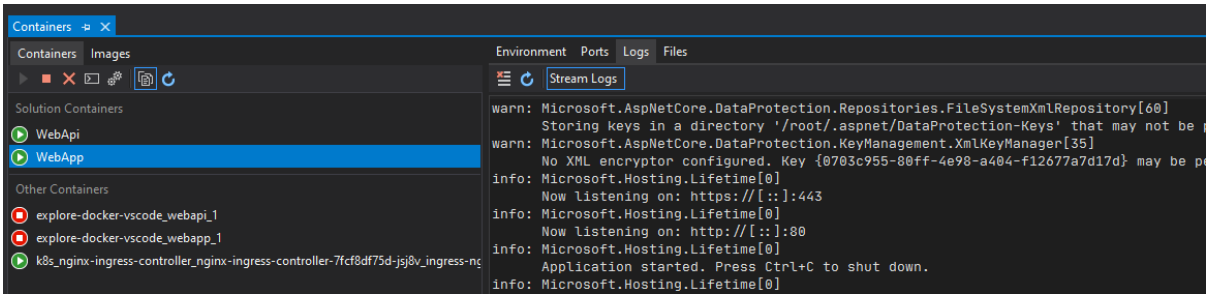

Figure 4-44. Visual Studio "Containers" window

The **Containers** window lets you view running containers, browse available images, view environment variables, logs, and port mappings, inspect the filesystem, attach a debugger, or open a terminal window inside the container environment.

As you can see, the integration between Visual Studio 2022 and Docker is completely oriented to the developer's productivity.

Of course, you can also list the images using the `docker images` command. You should see the `webapi` and `webapp` images with the `dev` tags created by the automatic deployment of our project with Visual Studio 2022.

```
docker images
```

```
$ docker images
REPOSITORY                          TAG                 IMAGE ID            CREATED             SIZE
webapi                              latest              301ede061a1e        5 minutes ago       210MB
webapp                              latest              413e9355ed78        5 minutes ago       210MB
mcr.microsoft.com/dotnet/sdk        5.0                 acc2e9a7698d        25 hours ago        616MB
mcr.microsoft.com/dotnet/aspnet     5.0                 66792fe28528        25 hours ago        205MB
```

Figure 4-45. View of Docker images

Register the Solution in an Azure Container Registry (ACR)

You can upload the images to the Azure Container Registry (ACR), but you could also use Docker Hub or any other registry, so the images can be deployed to the AKS cluster from that registry.

Create an ACR instance

Run the following command from the **az cli**:

```
az acr create --name exploredocker --resource-group explore-docker-aks-rg --sku basic --
admin-enabled
```

> **Note**
>
> The container registry name (e.g `exploredocker`) must be unique within Azure, and contain 5-50 alphanumeric characters. For more details, refer Create a container registry

Create the image in Release mode

You'll now create the image in **Release** mode (ready for production) by changing to **Release**, as shown in Figure 4-46, and running the application as you did before.

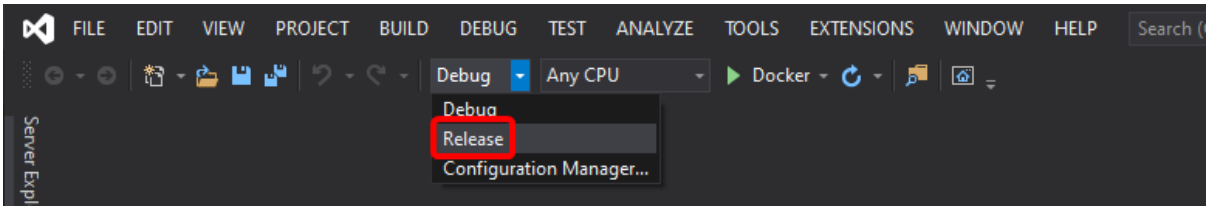

Figure 4-46. Selecting Release Mode

If you execute the `docker images` command, you'll see both images created, one for `debug` (**dev**) and the other for `release` (**latest**) mode.

Create a new Tag for the Image

Each container image needs to be tagged with the `loginServer` name of the registry. This tag is used for routing when pushing container images to an image registry.

You can view the `loginServer` name from the Azure portal, taking the information from the Azure Container Registry

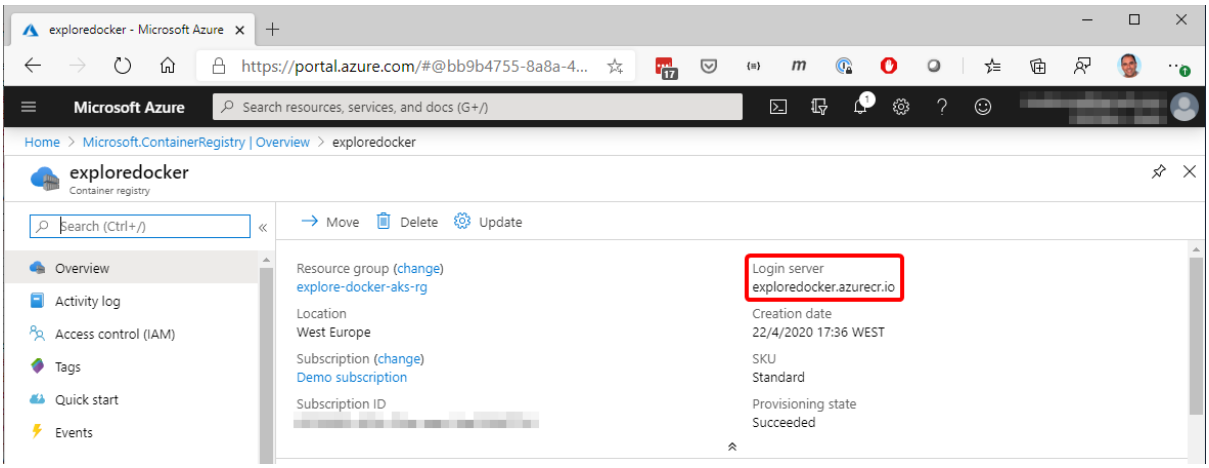

Figure 4-47. View of the name of the Registry

Or by running the following command:

```
az acr list --resource-group <resource-group-name> --query
"[].{acrLoginServer:loginServer}" --output table
```

```
PS: [Miguel]>az acr list --resource-group explore-docker-aks-rg --query "[].{acrLoginServer:loginServer}" --output table
AcrLoginServer
------------------------
exploredocker.azurecr.io
```

Figure 4-48. Get the name of the registry using az cli

In both cases, you'll obtain the name. In our example, `exploredocker.azurecr.io`.

Now you can tag the image, taking the latest image (the Release image), with the command:

```
docker tag <image-name>:latest <login-server-name>/<image-name>:v1
```

After running the `docker tag` command, list the images with the `docker images` command, and you should see the image with the new tag.

```
PS: [Miguel]>docker images
REPOSITORY                          TAG       IMAGE ID       CREATED             SIZE
webapp                              latest    11f2a7d7718c   4 minutes ago       213MB
exploredocker.azurecr.io/webapp     v1        11f2a7d7718c   4 minutes ago       213MB
webapi                              latest    db9979d4cfef   4 minutes ago       208MB
exploredocker.azurecr.io/webapi     v1        db9979d4cfef   4 minutes ago       208MB
webapp                              dev       cba194e20cb9   About an hour ago   207MB
webapi                              dev       ed6ea5466cb4   About an hour ago   207MB
```

Figure 4-49. View of tagged images

Push the image into the Azure ACR

Log in to the Azure Container Registry

```
az acr login --name exploredocker
```

Push the image into the Azure ACR, using the following command:

```
docker push <login-server-name>/<image-name>:v1
```

This command takes a while uploading the images but gives you feedback in the process. In the following image, you can see the output from one image completed and another in progress.

```
PS: [Miguel]>docker push exploredocker.azurecr.io/webapi:v1
The push refers to repository [exploredocker.azurecr.io/webapi]
35e6bc6eeb8d: Pushed
1754f2ec3a1a: Pushed
8957624a4d9d: Pushed
b21b0b2df887: Pushed
77362d5ab4dc: Pushed
974ff534c026: Pushed
b60e5c3bcef2: Pushed
v1: digest: sha256:0f9415358734702a89084b8461eed3c50047c5c347093c311eb8d32d15ad0fda size: 1791
PS: [Miguel]>docker push exploredocker.azurecr.io/webapp:v1
The push refers to repository [exploredocker.azurecr.io/webapp]
33e7f8019853: Pushing [================================================>]  5.322MB
1754f2ec3a1a: Preparing
8957624a4d9d: Preparing
b21b0b2df887: Preparing
77362d5ab4dc: Preparing
974ff534c026: Waitingng
b60e5c3bcef2: Waiting
```

Figure 4-50. Console output from the push command.

To deploy your multi-container app into your AKS cluster you need some manifest `.yaml` files that have, most of the properties taken from the `docker-compose.yml` and `docker-compose.override.yml` files.

deploy-webapi.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapi
  labels:
    app: weather-forecast
spec:
  replicas: 1
  selector:
    matchLabels:
      service: webapi
  template:
    metadata:
      labels:
        app: weather-forecast
        service: webapi
    spec:
      containers:
        - name: webapi
          image: exploredocker.azurecr.io/webapi:v1
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
              protocol: TCP
```

```
          env:
            - name: ASPNETCORE_URLS
              value: http://+:80
---
apiVersion: v1
kind: Service
metadata:
  name: webapi
  labels:
    app: weather-forecast
    service: webapi
spec:
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  selector:
    service: webapi
```

deploy-webapp.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: webapp
  labels:
    app: weather-forecast
spec:
  replicas: 1
  selector:
    matchLabels:
      service: webapp
  template:
    metadata:
      labels:
        app: weather-forecast
        service: webapp
    spec:
      containers:
        - name: webapp
          image: exploredocker.azurecr.io/webapp:v1
          imagePullPolicy: IfNotPresent
          ports:
            - containerPort: 80
              protocol: TCP
          env:
            - name: ASPNETCORE_URLS
              value: http://+:80
            - name: WebApiBaseAddress
              value: http://webapi
---
apiVersion: v1
kind: Service
metadata:
  name: webapp
  labels:
    app: weather-forecast
    service: webapp
spec:
  type: LoadBalancer
```

```
  ports:
    - port: 80
      targetPort: 80
      protocol: TCP
  selector:
    service: webapp
```

Note

The previous `.yml` files only enable the `HTTP` ports, using the `ASPNETCORE_URLS` parameter, to avoid issues with the missing certificate in the sample app.

Tip

You can see how to create the AKS Cluster for this sample in section **Deploy to Azure Kubernetes Service (AKS)** on this guide.

Now you're almost ready to deploy using **kubectl**, but first you must get the credentials from the AKS Cluster with this command:

```
az aks get-credentials --resource-group explore-docker-aks-rg --name explore-docker-aks
```

```
PS: [Miguel]>az aks get-credentials --resource-group explore-docker-aks-rg --name explore-docker-aks
Merged "explore-docker-aks" as current context in C:\Users\Miguel\.kube\config
```

Figure 4-51. Getting credentials from AKS into the kubectl environment.

You also have to allow the AKS cluster to pull images from the ACR, using this command:

```
az aks update --name explore-docker-aks --resource-group explore-docker-aks-rg --attach-acr
exploredocker
```

The previous command might take a couple of minutes to complete. Then, use the `kubectl apply` command to launch the deployments, and then `kubectl get all` get list the cluster objects.

```
kubectl apply -f deploy-webapi.yml
kubectl apply -f deploy-webapp.yml

kubectl get all
```

```
PS: [Miguel]>kubectl apply -f deploy-webapi.yml
deployment.apps/webapi created
service/webapi created
PS: [Miguel]>kubectl apply -f deploy-webapp.yml
deployment.apps/webapp created
service/webapp created
PS: [Miguel]>kubectl get all
NAME                         READY   STATUS              RESTARTS   AGE
pod/webapi-676467cfc8-wqg5s  0/1     ContainerCreating   0          15s
pod/webapp-56cfc7fdb5-cnxsd  0/1     ContainerCreating   0          7s


NAME                 TYPE           CLUSTER-IP    EXTERNAL-IP   PORT(S)        AGE
service/kubernetes   ClusterIP      10.0.0.1      <none>        443/TCP        4h33m
service/webapi       ClusterIP      10.0.97.189   <none>        80/TCP         15s
service/webapp       LoadBalancer   10.0.74.46    <pending>     80:31349/TCP   7s


NAME                    READY   UP-TO-DATE   AVAILABLE   AGE
deployment.apps/webapi  0/1     1            0           15s
deployment.apps/webapp  0/1     1            0           7s

NAME                               DESIRED   CURRENT   READY   AGE
replicaset.apps/webapi-676467cfc8  1         1         0       15s
replicaset.apps/webapp-56cfc7fdb5  1         1         0       7s
```

Figure 4-52. Deployment to Kubernetes

You'll have to wait a while until the load balancer gets the external IP, checking with `kubectl get services`, and then the application should be available at that address, as shown in the next image:

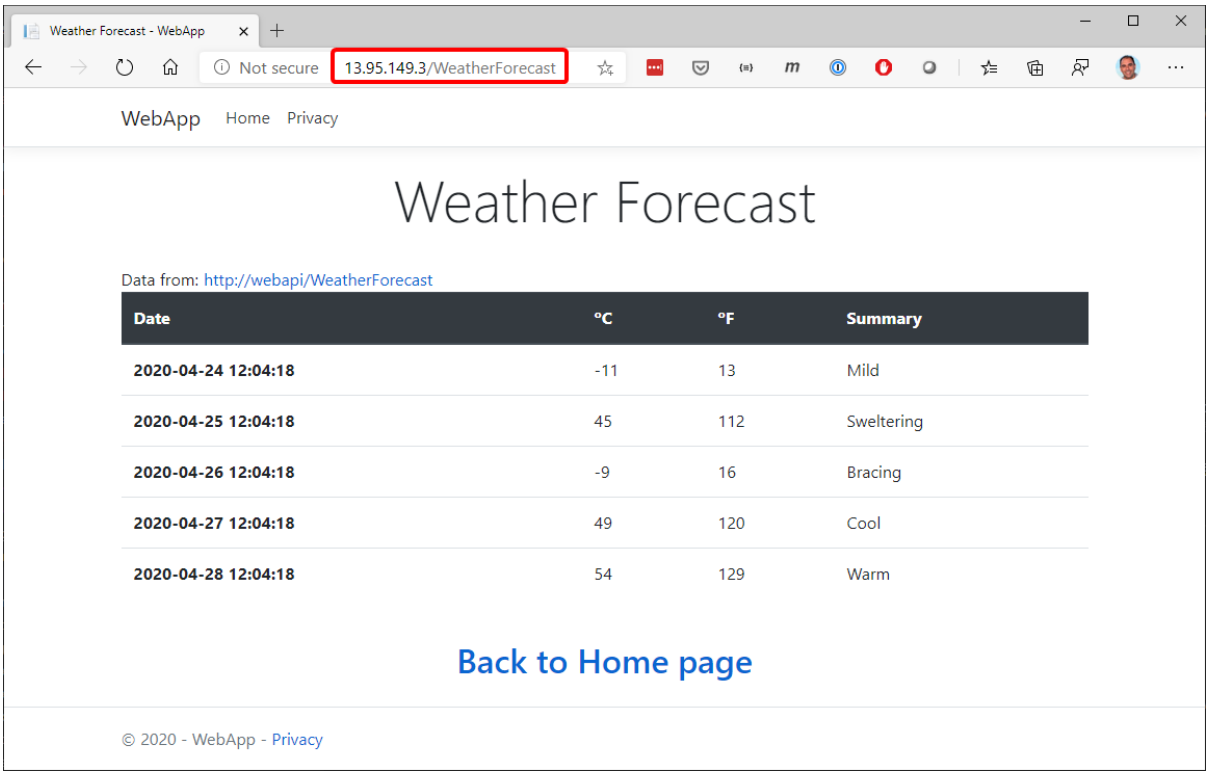

Figure 4-53. Deployment to Kubernetes

When the deployment completes, you can access the Kubernetes Web UI with a local proxy, using an ssh tunnel.

First you must create a ClusterRoleBinding with the following command:

```
kubectl create clusterrolebinding kubernetes-dashboard --clusterrole=cluster-admin --
serviceaccount=kube-system:kubernetes-dashboard
```

And then this command to start the proxy:

```
az aks browse --resource-group exploredocker-aks-rg --name explore-docker-aks
```

A browser window should open at `http://127.0.0.1:8001` with a view similar to this one:

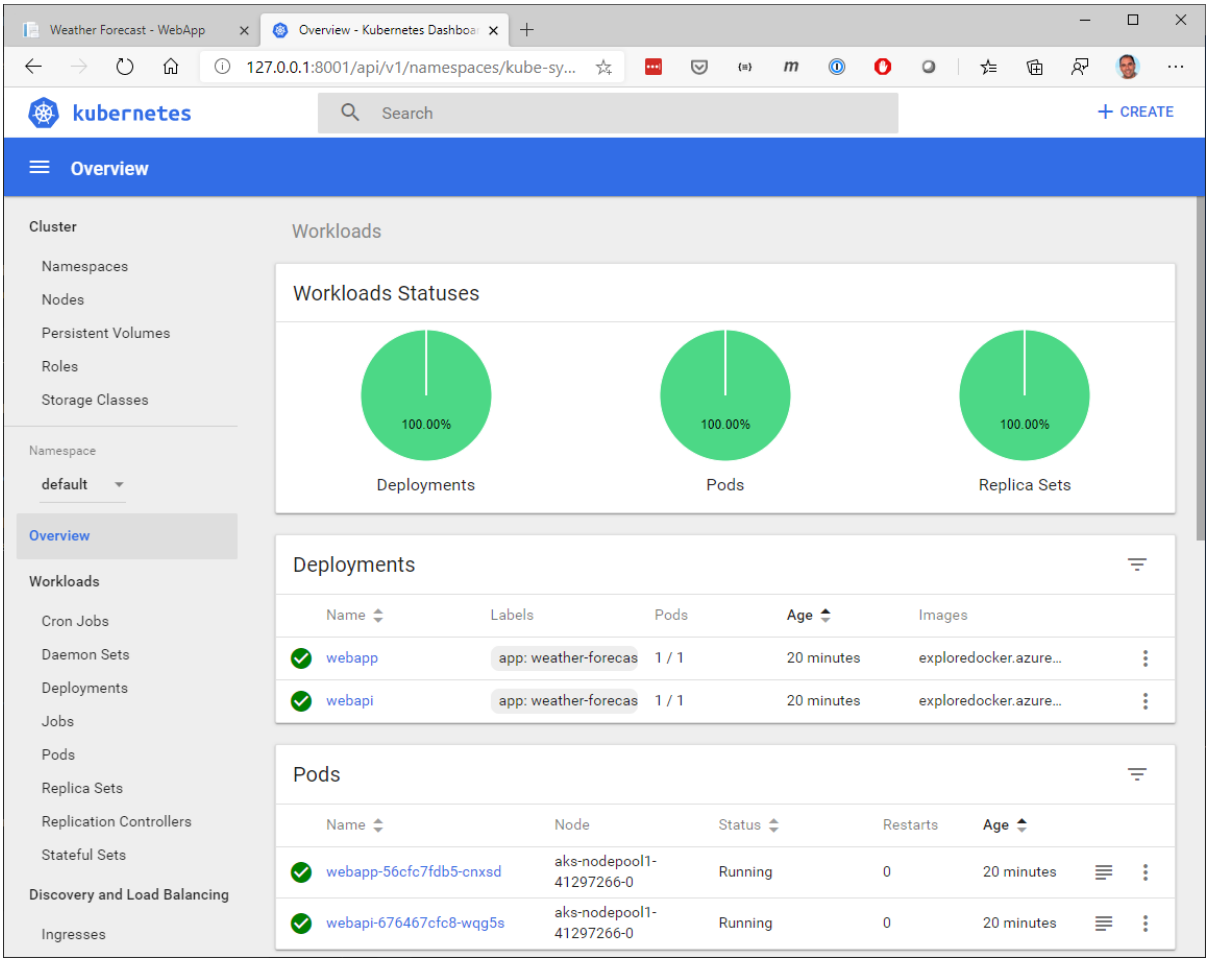

Figure 4-54. View Kubernetes cluster information

Now you have your ASP.NET Core application, running in Linux containers, and deployed to an AKS cluster on Azure.

Note

For more information on deployment with Kubernetes see: https://kubernetes.io/docs/reference/kubectl/cheatsheet/

CHAPTER 5

Docker application DevOps workflow with Microsoft tools

Microsoft Visual Studio, Azure DevOps Services and/or GitHub, Team Foundation Server, and Azure Monitor provide a comprehensive ecosystem for development and IT operations that give your team the tools to manage projects and rapidly build, test, and deploy containerized applications.

Teams can choose which tools and platforms they want to use for end to end DevOps. With Visual Studio and Azure DevOps Services in the cloud, along with Team Foundation Server on-premises, development teams can productively build, test, and release containerized applications that target either Windows or Linux. Alternatively, teams can also use Visual Studio Code and GitHub. Teams can even use combinations: for example, storing source code in GitHub and using Azure Boards for work item tracking and Azure Pipelines for CI/CD.

Microsoft tools can automate the pipeline for specific implementations of containerized applications—Docker, .NET, or any combination with other platforms—from global builds and Continuous Integration (CI) and tests with Azure DevOps Services, Team Foundation Server or GitHub, to Continuous Deployment (CD) to Docker environments (Development, Staging, Production), and to transmit analytics information about the services to the development team through Azure Monitor. Every code commit can initiate a build (CI) and automatically deploy the services to specific containerized environments (CD).

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

The complexity of containerized application development increases steadily depending on the business complexity and scalability needs. A good example of this complexity are applications based on microservices architectures. To succeed in such an environment, your project must automate the entire life cycle—not only the build and deployment, but it also must manage versions along with the collection of telemetry. Azure DevOps Services, GitHub and Azure offer the following capabilities:

- Azure DevOps Services/Team Foundation Server source code management (based on Git or Team Foundation Version Control), Agile planning (Agile, Scrum, and CMMI are supported), CI, release management, and other tools for Agile teams.
- Azure DevOps Services and Team Foundation Server include a powerful and growing ecosystem of first and third-party extensions with which you easily can construct a CI, build, test, delivery, and release management pipeline for microservices.
- GitHub or GitHub Enterprise Server offer similar capabilities, with source control based on Git, Projects and Issues for project tracking, GitHub Actions for automating workflows including CI/CD, and GitHub Advanced Security for dependency, secret and vulnerability scanning.
- Run automated tests as part of your build pipeline in Azure DevOps Services or through GitHub Actions
- Azure DevOps Services/GitHub can tighten the DevOps life cycle with delivery to multiple environments, not just for production environments, but also for testing, including A/B experimentation, canary releases, and so on.
- Organizations easily can 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 templates with tools they're already comfortable with.

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

Figure 5-1 presents an end-to-end depiction of the steps comprising the DevOps outer-loop workflow. It shows the "outer loop" of DevOps. When code is pushed to the repo, a CI pipeline is started, then begins the CD pipeline, where the application gets deployed. Metrics collected from deployed applications are fed back into the development workload, where the "inner loop" occurs, so development teams have actual data to respond to user and business needs.

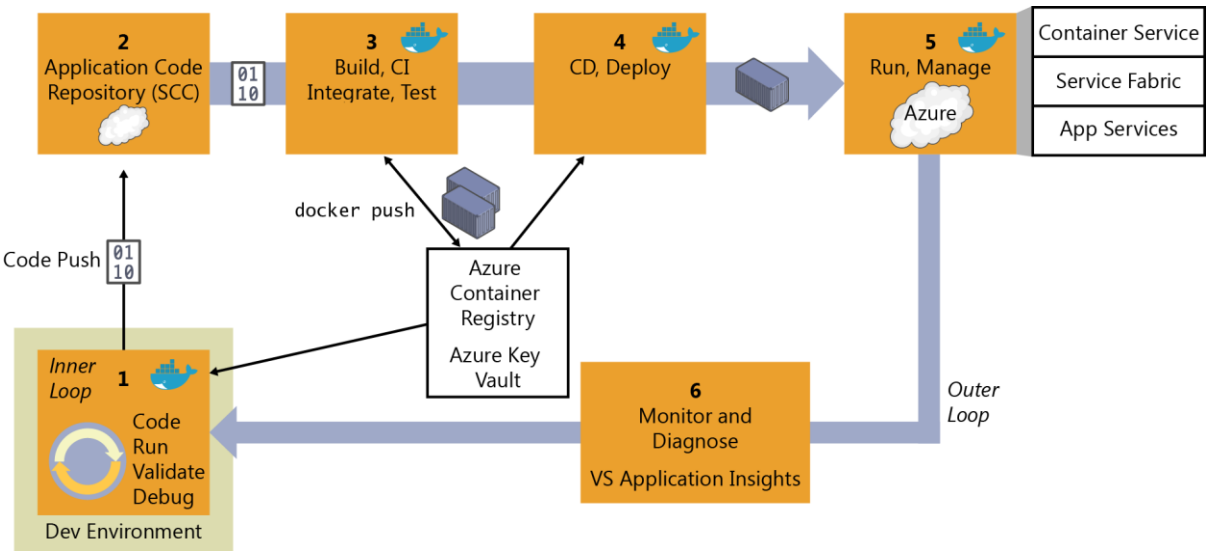

Figure 5-1. DevOps outer-loop workflow for Docker applications with Microsoft tools

Now, let's examine each of these steps in greater detail.

Step 1: Inner-loop development workflow

This step is explained in detail in Chapter 4, but, to recap, here is where the outer-loop begins, the moment at which a developer pushes code to the source control management system (like Git) initiating CI pipeline actions.

Step 2: Source-Code Control integration and management with Azure DevOps 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 though source-code control (SCC) and source-code management might seem second-nature to most developers, when creating Docker applications in a DevOps life cycle, it's critical to emphasize that you must not submit the Docker images with the application 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 must be created solely on the source code that's being integrated in your global build or CI pipeline based on your source-code repository (like Git).

The local images, generated by developers, should just be used by them when testing within their own machines. That's why it's critical to have the DevOps pipeline activated from the SCC code.

Azure DevOps Services and Team Foundation Server 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 repositories, or Subversion) and still be able to connect to it and get the code as the starting point for your DevOps CI pipeline. You can also use GitHub Actions for CI/CD pipelines.

Step 3: Build, CI, Integrate, and Test with Azure DevOps Services/GitHub and Docker

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 between what's developed, tested through CI, and run in production. Docker Engine deployed across the developer laptops and test infrastructure makes the containers portable across environments.

At this point, after you have a version-control system with the correct 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 (Azure DevOps Services/GitHub), Docker Engine, and a Docker Registry.

You can use Azure DevOps Services as the foundation for building your applications and setting your CI pipeline, and for publishing the built "artifacts" to an "artifacts repository," which is explained in the next step. Alternatively, you can use GitHub to implement the same workflow.

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

Here is the basic concept: The CI pipeline will be kicked-off by a commit to an SCC repository like Git. The commit will cause Azure DevOps Services/GitHub to run a build job within a Docker container and, upon successful completion of that job, push a Docker image to the Docker Registry, as illustrated in Figure 5-2. The first part of the outer loop involves steps 1 to 3, from code, run, debug and validate, then the code repo up to the build and test CI step.

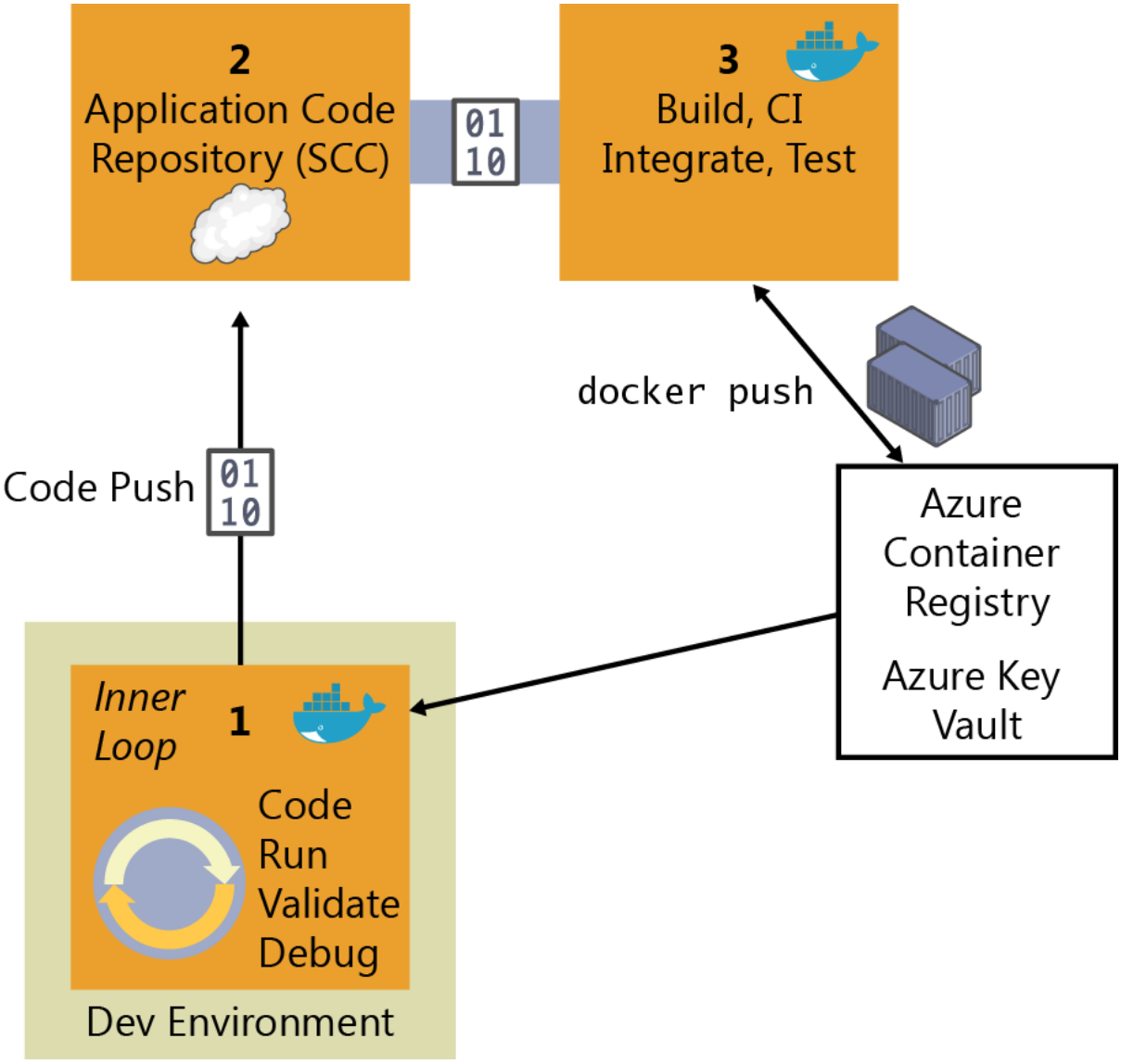

Figure 5-2. The steps involved in CI

Here are the basic CI workflow steps with Docker and Azure DevOps Services:

1. The developer pushes a commit to an SCC repository (Git/Azure DevOps Services, GitHub, etc.).

2. If you're using Azure DevOps Services or Git, CI is built in, which means that it's as simple as selecting a check box in Azure DevOps Services. If you're using an external SCC (like GitHub), a `webhook` will notify Azure DevOps Services of the update or push to Git/GitHub.

3. Azure DevOps Services pulls the SCC repository, including the Dockerfile describing the image, as well as the application and test code.

4. Azure DevOps Services builds a Docker image and labels it with a build number.

5. Azure DevOps Services instantiates the Docker container within the provisioned Docker Host, and runs the appropriate tests.

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

Here are the basic CI workflow steps with Docker and GitHub:

1. The developer pushes a commit to a GitHub repo.
2. CI is built in, so Actions will trigger base on the event filters.
3. GitHub pulls the SCC repository, including the *Dockerfile* describing the image, as well as the application and test code.
4. GitHub builds a Docker image and labels it with a build number.
5. GitHub instantiates the Docker container within the provisioned Docker Host, and runs the appropriate tests.
6. If the tests are successful, the image is first relabeled to a meaningful name so that you know it's a "blessed build" (like "/1.0.0" or any other label), and then pushed up to your Docker Registry (Docker Hub, Azure Container Registry, DTR, etc.)

Implement a CI pipeline with Azure DevOps Services and the Docker extension for Azure DevOps Services

Visual Studio Azure DevOps Services contains Build & Release Templates that you can use in your CI/CD pipeline with which you can build Docker images, push Docker images to an authenticated Docker registry, run Docker images, or run other operations offered by the Docker CLI. It also adds a Docker Compose task that you can use to build, push, and run multi-container Docker applications, or run other operations offered by the Docker Compose CLI, as shown in Figure 5-3.

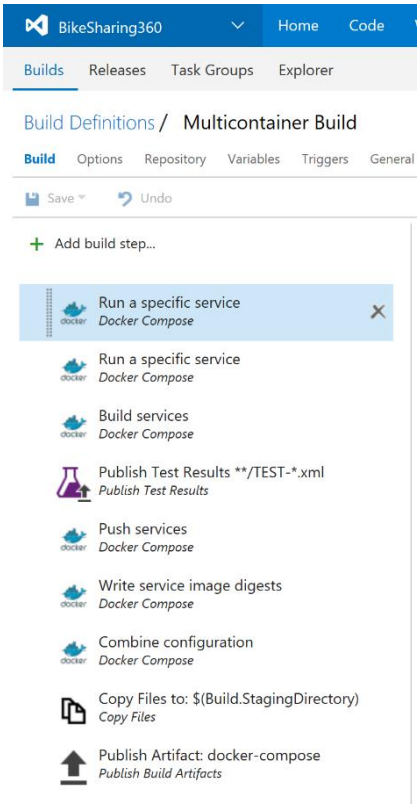

Figure 5-3. The Docker CI pipeline in Azure DevOps Services including Build & Release Templates and associated tasks.

You can use these templates and tasks to construct your CI/CD artifacts to Build / Test and Deploy in Azure Service Fabric, Azure Kubernetes Service, and similar offerings.

With these Visual Studio Team Services 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:

- Azure DevOps Services, or for on-premises installations, Team Foundation Server 2015 Update 3 or later.
- An Azure DevOps Services agent that has the Docker binaries.

An easy way to create one of these agents is to use Docker to run a container based on the Azure DevOps Services agent Docker image.

Tip

To read more about assembling an Azure DevOps Services Docker CI pipeline and view the walkthroughs, visit these sites:

- Running a Visual Studio Team Services (Now Azure DevOps Services) agent as a Docker container: https://hub.docker.com/_/microsoft-azure-pipelines-vsts-agent
- Building .NET Linux Docker images with Azure DevOps Services: https://docs.microsoft.com/archive/blogs/stevelasker/building-net-core-linux-docker-images-with-visual-studio-team-services
- Building a Linux-based Visual Studio Team Service build machine with Docker support: https://www.donovanbrown.com/post/Building-a-Linux-Based-Visual-Studio-Team-Service-Build-Machine-with-Docker-Support

Implement a CI pipeline with GitHub Actions

GitHub Actions allow you to create automation scripts that can build Docker images, push Docker images to an authenticated Docker registry, run Docker images, or run other operations offered by the Docker CLI.

You can use public Actions (such as Azure Login) and `run` (shell) commands to construct your CI/CD artifacts to Build / Test and Deploy in Azure Service Fabric, Azure Kubernetes Service, and similar offerings.

With these Actions, 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.

Integrate, test, and validate multi-container Docker applications

Typically, most Docker applications are composed of multiple containers rather than a single container. A good example is a microservices-oriented application for which you would have one container per microservice. But, even without strictly following the microservices approach patterns, it's probable that your Docker application would be composed of multiple containers or 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 of its containers within an integration Docker host or even into a test cluster to which your containers are distributed.

If you're using a single host, you can use Docker commands such as docker-compose to build and deploy related containers to test and validate the Docker environment in a single VM. But, if you're working with an orchestrator cluster like DC/OS, Kubernetes, or Docker Swarm, 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 you can run against Docker containers:

- Unit tests for Docker containers
- Testing groups of interrelated applications or microservices
- Test in production and "canary" releases

The important point is that when running integration and functional tests, you must run those tests from outside of the containers. Tests are not contained or run in the containers you're deploying, because the containers are based on static images that should be exactly like the ones you'll be deploying to production.

A practical option when testing more advanced scenarios, like including several clusters (test cluster, staging cluster, and production cluster) is to publish the images to a registry, so it can be tested in various clusters.

Push the custom application Docker image into your global Docker Registry

After the Docker images have been tested and validated, you'll want to tag and publish them to your Docker registry. The Docker registry is a critical piece in the Docker application life cycle because it's the central place where you store your custom test (also known as "blessed images") to be deployed into QA and production environments.

Similar to how the application code stored in your SCC repository (Git, etc.) is your "source of truth," the Docker registry is your "source of truth" for your binary application or bits to be deployed to the QA or 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, 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 method you use is similar and is based on the `docker push` command, as shown in Figure 5-4.

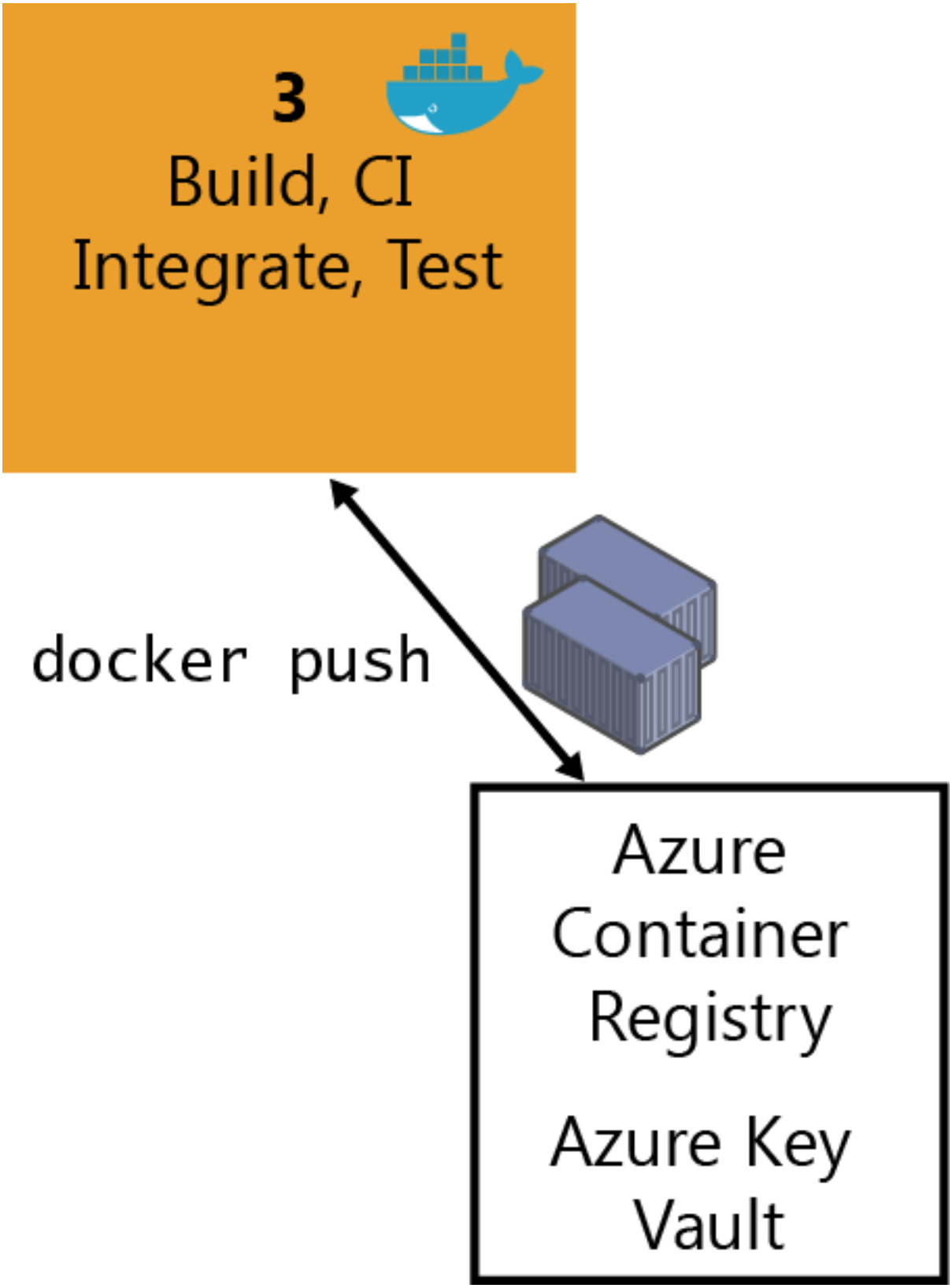

Figure 5-4. Publishing custom images to Docker Registry

In step 3, for building integration and testing (CI) you might publish the resulting docker images to a private or public registry. There are multiple offerings of Docker registries from cloud vendors like Azure Container Registry, Amazon Web Services Container Registry, Google Container Registry, GitHub Container Registry, Quay Registry, and so on.

Using the Docker tasks, you can 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 Figure 5-5.

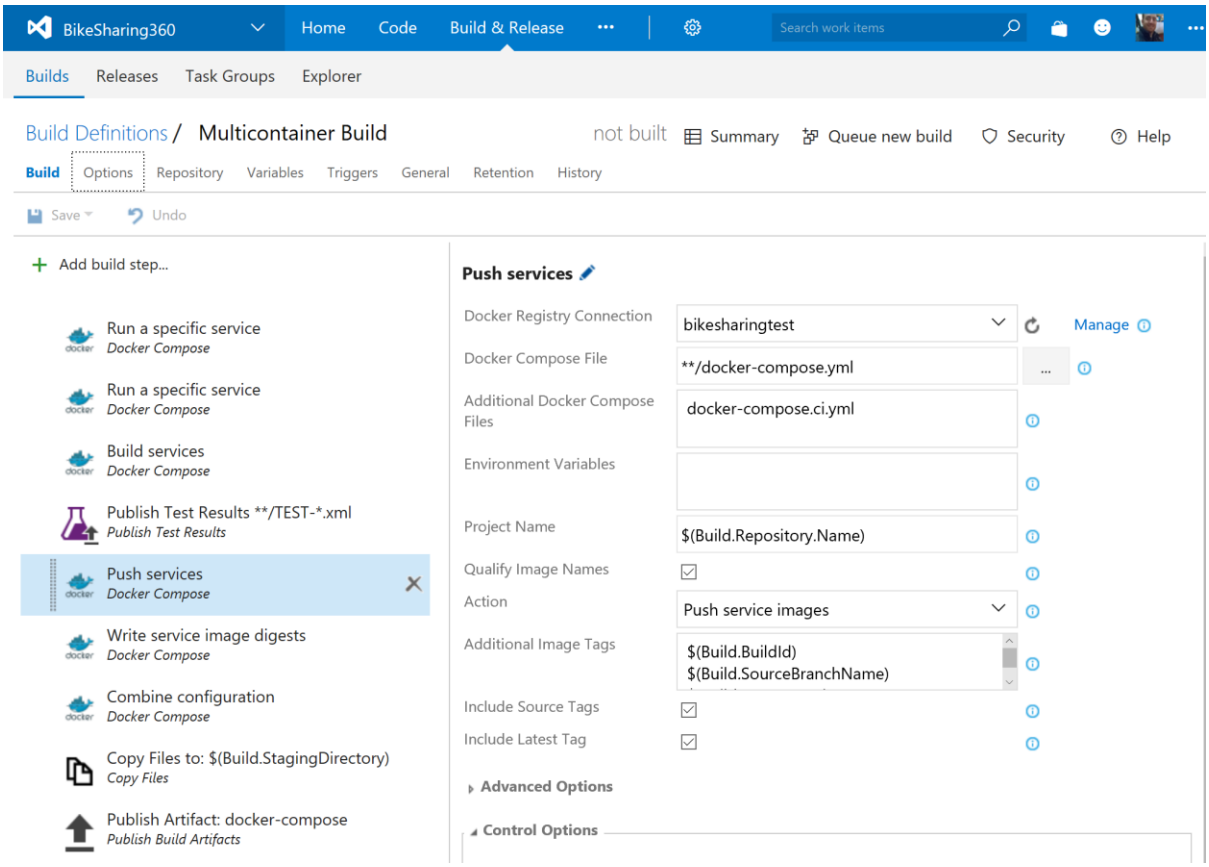

Figure 5-5. Using Azure DevOps Services to publishing custom images to a Docker Registry

Tip

For more information about Azure Container Registry, see https://aka.ms/azurecontainerregistry.

Step 4: CD, Deploy

The immutability of Docker images ensures a repeatable deployment with what's developed, tested through CI, and run in production. After you have the application Docker images published in your Docker registry (either private or public), you can deploy them to the several environments that you might have (production, QA, staging, etc.) from your CD

pipeline by using Azure DevOps Services pipeline tasks, Azure DevOps Services Release Management or GitHub Actions.

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

Deploying composed Docker applications to multiple Docker environments

Let's look first at the less-complex scenario: deploying to simple Docker hosts (VMs or servers) in a single environment or multiple environments (QA, staging, and production). In this scenario, internally your CD pipeline can use docker-compose (from your Azure DevOps Services deployment tasks) to deploy the Docker applications with its related set of containers or services, as illustrated in Figure 5-6.

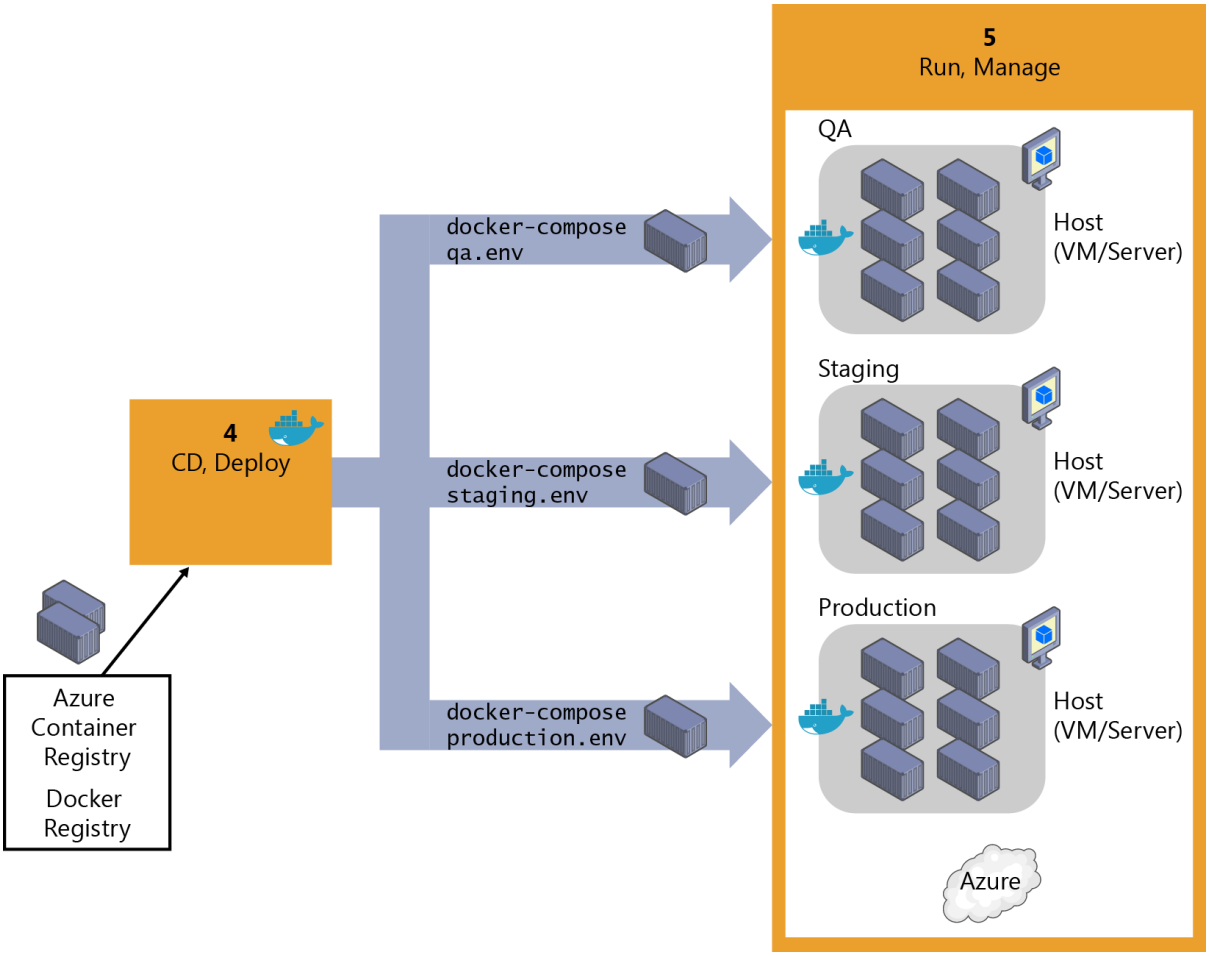

Figure 5-6. Deploying application containers to simple Docker host environments registry

Figure 5-7 highlights how you can connect your build CI to QA/test environments via Azure DevOps Services by clicking Docker Compose in the Add Task dialog box. However, when deploying to staging or production environments, you would usually use Release Management features handling multiple environments (like QA, staging, and production). If you're deploying to single Docker hosts, it is using the Azure DevOps Services "Docker Compose" task (which is invoking the `docker-compose up` command under the hood). If you're deploying to Azure Kubernetes Service (AKS), it uses the Docker Deployment task, as explained in the section that follows. Similar steps can be built for deployment using GitHub Actions.

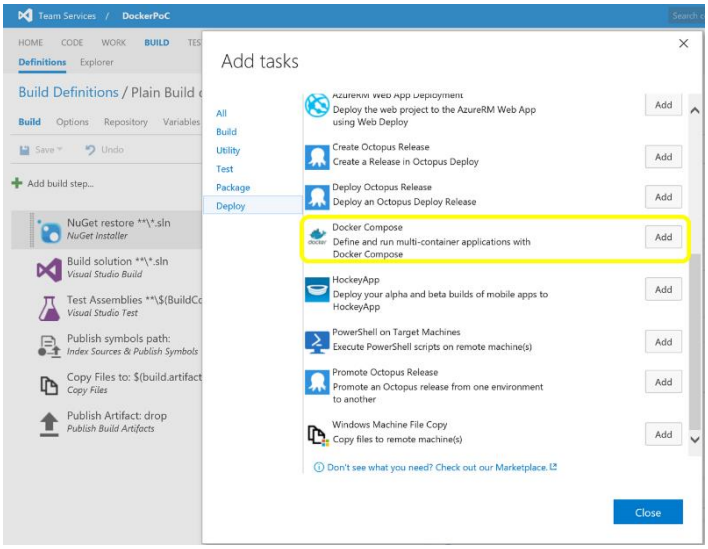

Figure 5-7. Adding a Docker Compose task in an Azure DevOps Services pipeline or GitHub workflow

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

The Azure DevOps Services templates give you the ability 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. You can invoke `docker-compose` in a `run` step inside GitHub Actions to accomplish the same goal.

Managing releases to Docker environments by using Azure DevOps Services Release Management or GitHub Actions

Through the Azure DevOps Services templates, 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 an entire application, all through the Azure DevOps

Services Release Management capabilities intended for multiple environments, as shown in Figure 5-8.

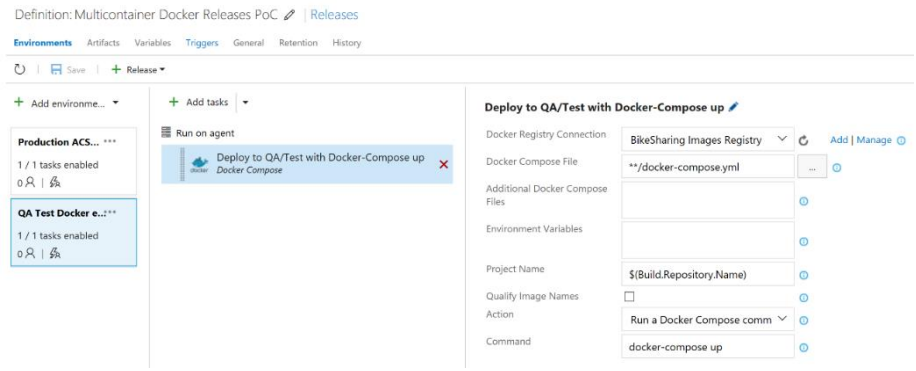

Figure 5-8. Configuring Azure DevOps Services Docker Compose tasks from Azure DevOps Services Release Management

However, keep in mind that the scenario shown in Figure 5-6 and implemented in Figure 5-8 is a simple one (it's deploying to single Docker hosts and VMs, and there will be a single container or instance per image) and probably should be used only for development or test scenarios. In most enterprise production scenarios, you would want to have High Availability (HA) and easy-to-manage scalability by load balancing across multiple nodes, servers, and VMs, plus "intelligent failovers" so if a server or node fails, its services and containers will be moved to another host server or VM. In that case, you need more advanced technologies such as container clusters, orchestrators, and schedulers. Thus, the way to deploy to those clusters is by handling the advanced scenarios explained in the next section.

GitHub Actions can be used in the same manner, including the use of environments for approvals.

Deploying Docker applications to Docker clusters

The nature of distributed applications requires compute resources that are also distributed. To have production-scale capabilities, you need to have clustering capabilities that provide high scalability and high availability based on pooled resources.

You could deploy containers manually to those clusters from a CLI tool or a web UI, but you should reserve that kind of manual work to spot deployment testing or management purposes like scaling-out or monitoring.

From a CD point of view, you can use Azure DevOps Services or GitHub Actions to run specially made deployment tasks from your environments that will deploy your containerized applications to distributed clusters in Container Service, as illustrated in Figure 5-9.

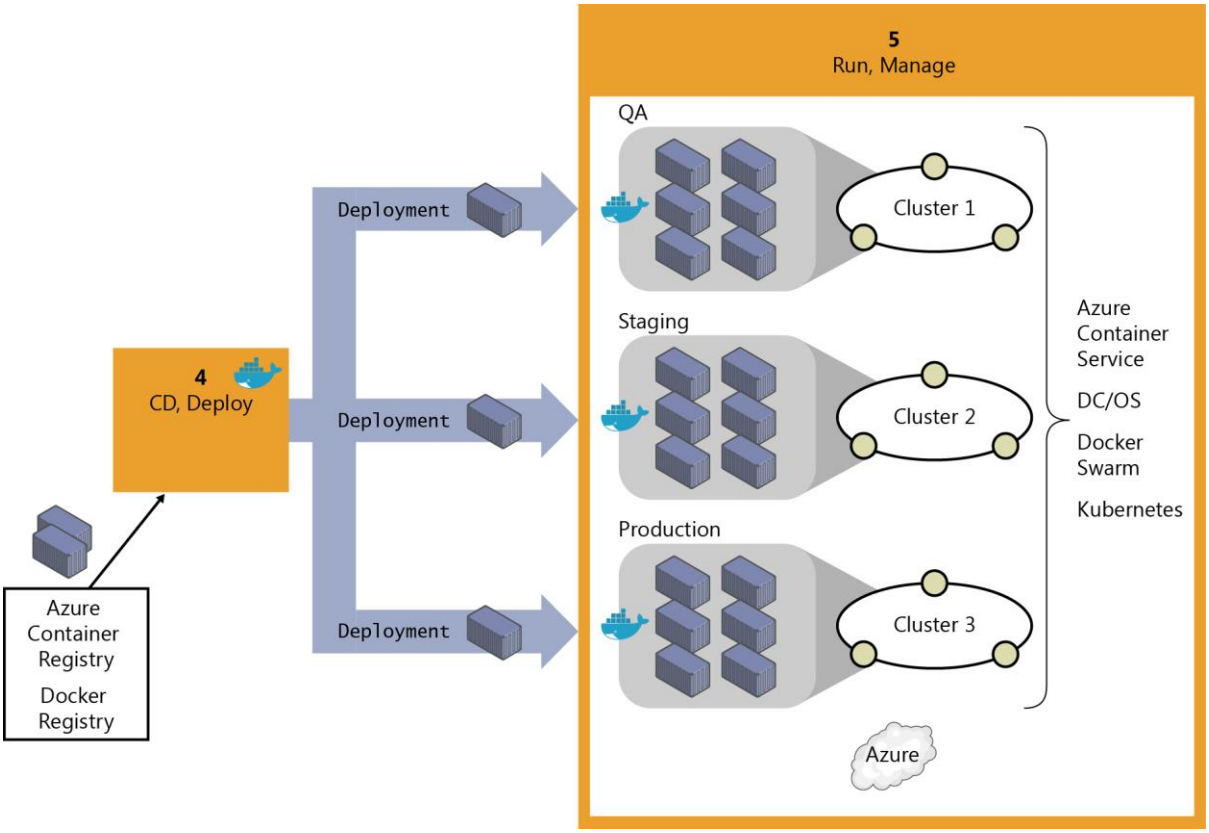

Figure 5-9. Deploying distributed applications to Container Service

Initially, when deploying to certain clusters or orchestrators, you would traditionally use specific deployment scripts and mechanisms per each orchestrator (that is, Kubernetes and Service Fabric have different deployment mechanisms) instead of the simpler and easy-to-use `docker-compose` tool based on the `docker-compose.yml` definition file. However, thanks to the Azure DevOps Services Docker Deploy task, shown in Figure 5-10, now you can also deploy to the supported orchestrators by just using your familiar `docker-compose.yml` file because the tool performs that "translation" for you (from your `docker-compose.yml` file to the format needed by the orchestrator).

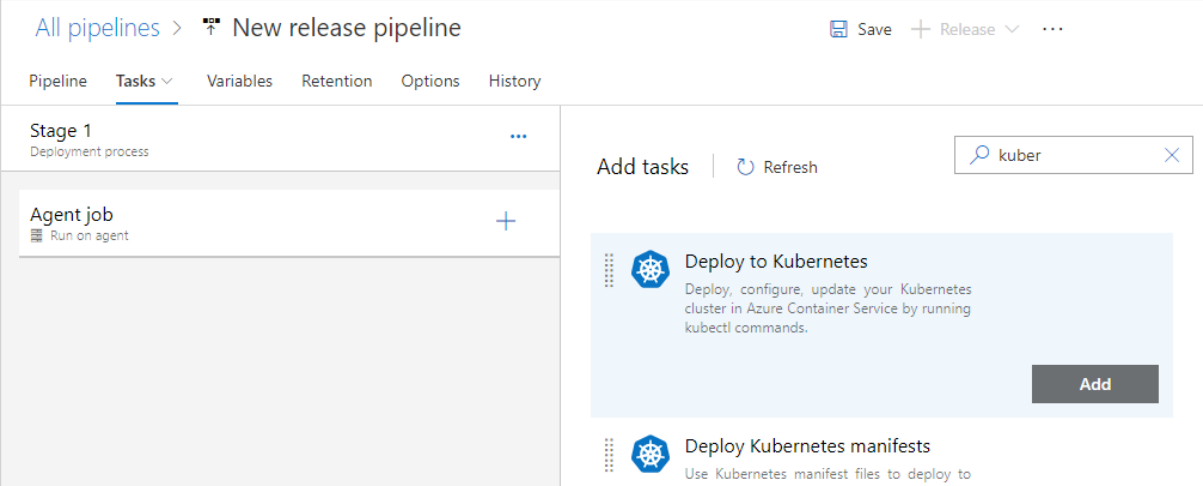

Figure 5-10. Adding the Deploy to Kubernetes task to your Environment

Figure 5-11 demonstrates how you can edit the Deploy to Kubernetes task with the sections available for configuration. This is the task that will retrieve your ready-to-use custom Docker images to be deployed as containers in the cluster.

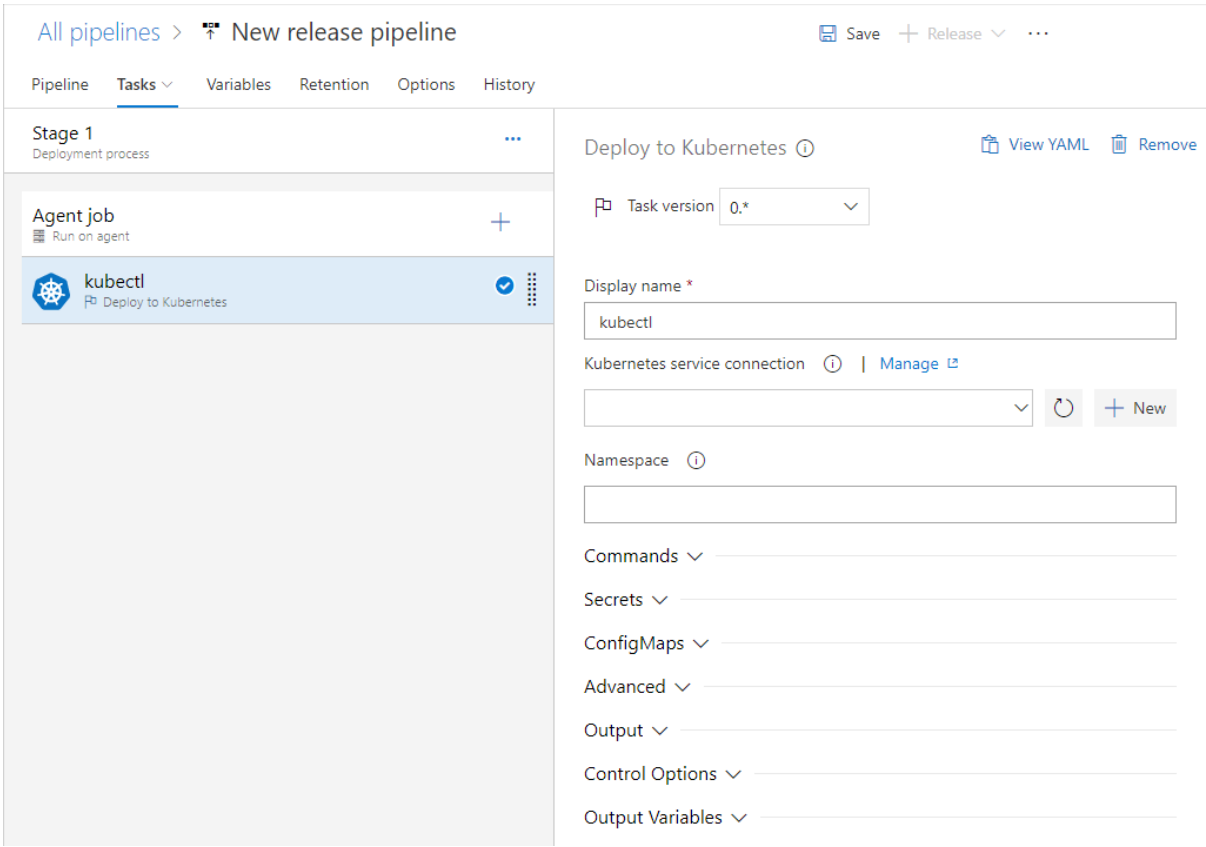

Figure 5-11. Docker Deploy task definition deploying to ACS DC/OS

Tip

To read more about the CD pipeline with Azure DevOps Services and Docker, visit https://azure.microsoft.com/services/devops/pipelines

Tip

To see GitHub Actions workflows for CI, visit https://github.com/dotnet-architecture/eShopOnContainers/wiki/GitHub-Actions. For a walkthrough of GitHub Actions performing deployment to an Azure Kubernetes environment, visit https://github.com/dotnet-architecture/eShopOnContainers/wiki/Deployment-With-GitHub-Actions.

Step 5: Run and manage

Because running and managing applications at enterprise-production level is a major subject in and of itself, and due to the type of operations and people working at that level (IT operations) as well as the large scope of this area, the entire next chapter is devoted to explaining it.

Step 6: Monitor and diagnose

This topic also is covered in the next chapter as part of the tasks that IT performs in production systems; however, is important to highlight that the insights obtained in this step must feed back to the development team so that the application is constantly improved. From that point of view, it's also part of DevOps, although the tasks and operations are commonly performed by IT.

Only when monitoring and diagnostics are 100% within the realm of DevOps are the monitoring processes and analytics performed by the development team against testing or beta environments. This is done either by performing load testing or by monitoring beta or QA environments, where beta testers are trying the new versions.

Create CI/CD pipelines in Azure DevOps Services for a .NET application on Containers and deploying to a Kubernetes cluster

In Figure 5-12 you can see the end-to-end DevOps scenario covering the code management, code compilation, Docker images build, Docker images push to a Docker registry and finally the deployment to a Kubernetes cluster in Azure.

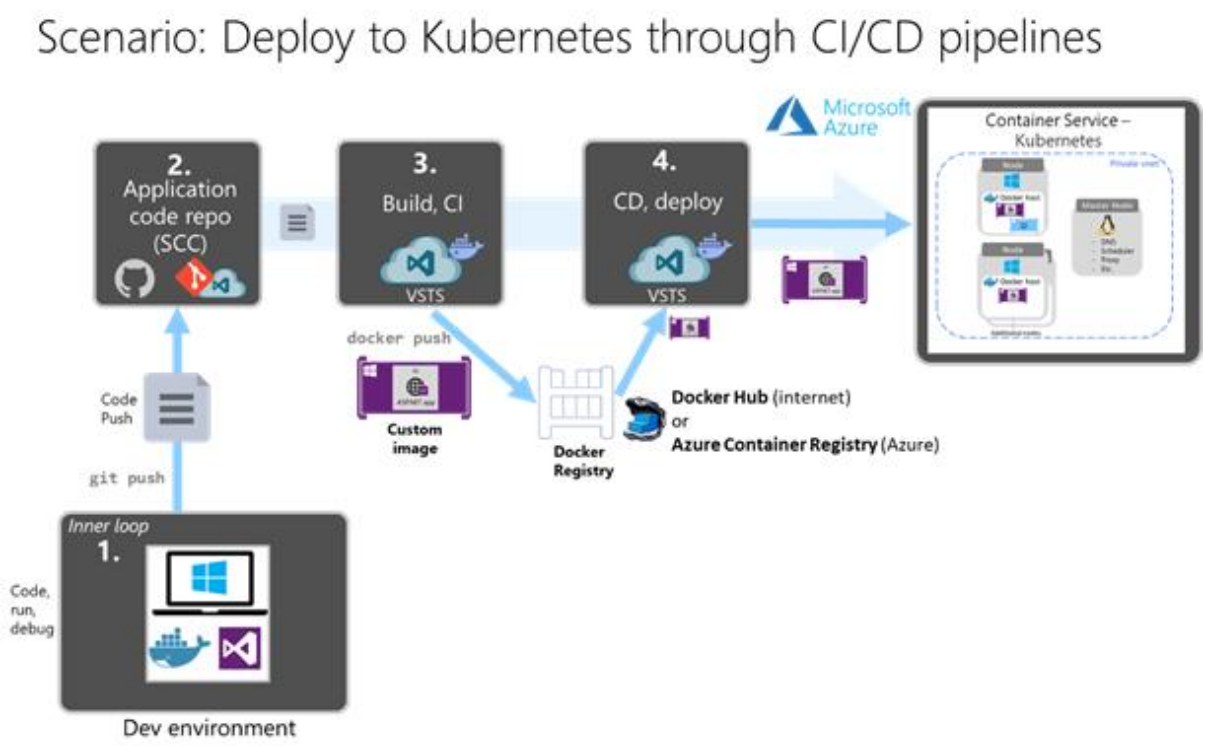

Figure 5-12. CI/CD scenario creating Docker images and deploying to a Kubernetes cluster in Azure

It is important to highlight that the two pipelines, build/CI, and release/CD, are connected through the Docker Registry (such as Docker Hub or Azure Container Registry). The Docker registry is one of the main differences compared to a traditional CI/CD process without Docker.

As shown in Figure 5-13, the first phase is the build/CI pipeline. In Azure DevOps Services you can create build/CI pipelines that will compile the code, create the Docker images, and push them to a Docker Registry like Docker Hub or Azure Container Registry.

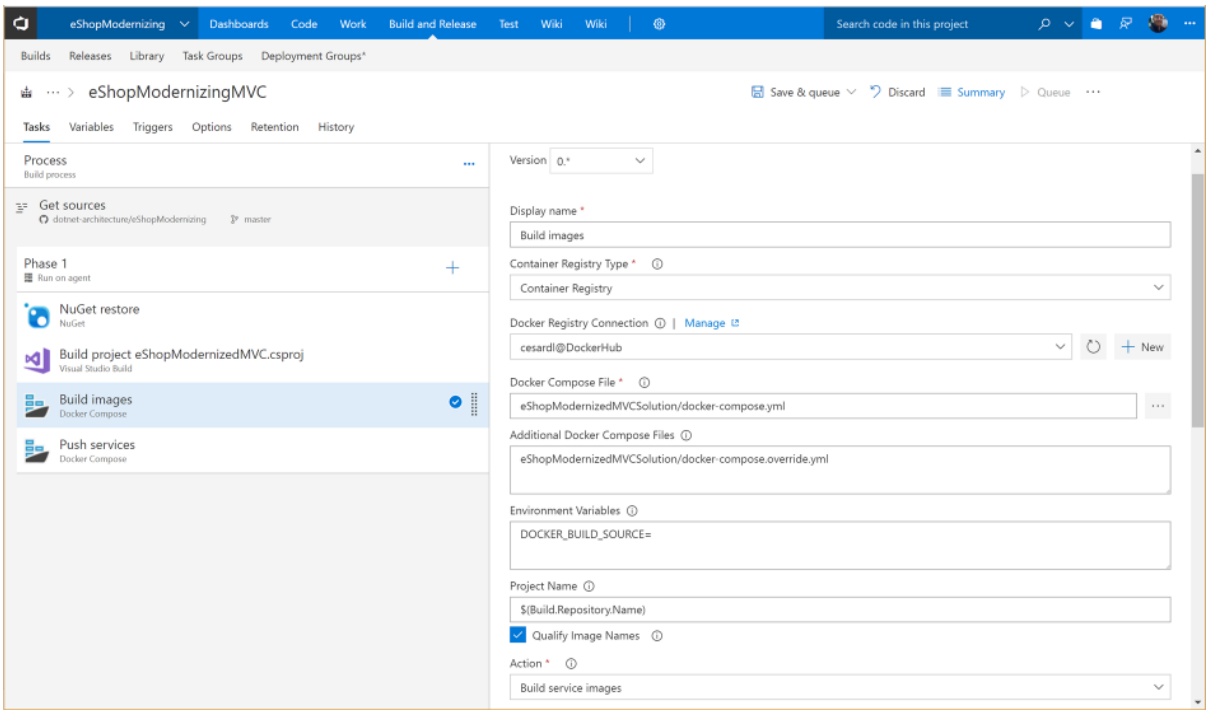

Figure 5-13. Build/CI pipeline in Azure DevOps building Docker images and pushing images to a Docker registry

The second phase is to create a deployment/release pipeline. In Azure DevOps Services, you can easily create a deployment pipeline targeting a Kubernetes cluster by using the Kubernetes tasks for Azure DevOps Services, as shown in Figure 5-14.

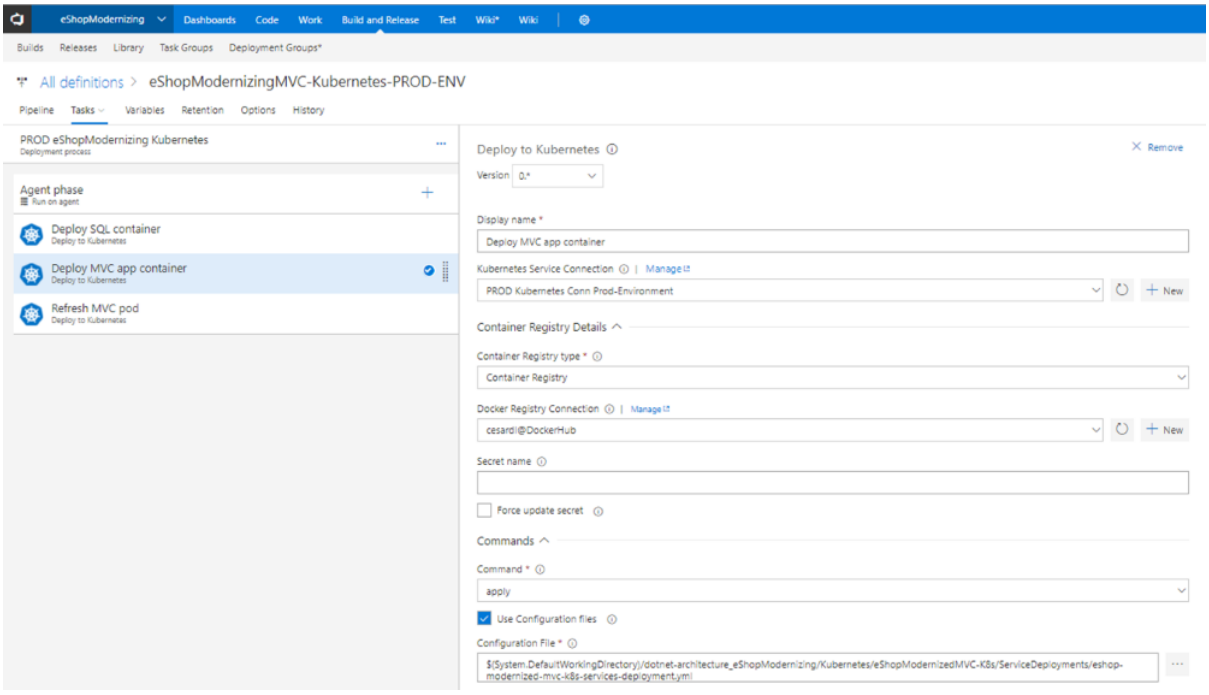

Figure 5-14. Release/CD pipeline in Azure DevOps Services deploying to a Kubernetes cluster

[!Walkthrough] Deploying eShopModernized to Kubernetes:

For a detailed walkthrough of Azure DevOps CI/CD pipelines deploying to Kubernetes, see this post:
https://github.com/dotnet-architecture/eShopModernizing/wiki/04.-How-to-deploy-your-Windows-Containers-based-apps-into-Kubernetes-in-Azure-Container-Service-(Including-CI-CD)

CHAPTER 6

Run, manage, and monitor Docker production environments

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

This last pillar in the containerized Docker applications life cycle is centered on how you can run, manage, and monitor your applications in scalable, high availability (HA) production environments.

The way you run your containerized applications in production (infrastructure architecture and platform technologies) is very much related and based on the chosen architecture and development platforms discussed in Chapter 1 of this e-book.

This chapter examines specific products and technologies from Microsoft and other vendors that you can use to effectively run scalable, HA distributed applications plus how you can manage and monitor them from the IT perspective.

Run composed and microservices-based applications in production environments

Applications composed by multiple microservices do need to be deployed into orchestrator clusters in order to simplify the complexity of deployment and make it viable from an IT point of view. Without an orchestrator cluster, it would be difficult to deploy and scale out a complex microservices application.

Introduction to orchestrators, schedulers, and container clusters

Earlier in this e-book, *clusters* and *schedulers* were introduced as part of the discussion on software architecture and development. Kubernetes and Service Fabric are examples of

Docker clusters. Both of these orchestrators can run as a part of the infrastructure provided by Microsoft Azure Kubernetes 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's precisely what orchestrators and schedulers provide. Let's take a brief look at them here:

- **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. Although 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 leave them in a pending state, having a degree of "fairness," being robust to errors, and always be available.

- **Orchestrators**. Platforms extend life-cycle management capabilities to complex, multi-container workloads deployed on a cluster of hosts. By abstracting the host infrastructure, orchestration tools give users a way to treat the entire cluster as a single deployment target.

The process of orchestration involves tooling and a platform that can automate all aspects of application management from initial placement or 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.

The capabilities provided by orchestrators and schedulers are complex to develop and create from scratch, therefore you usually would want to use orchestration solutions offered by vendors.

Manage production Docker environments

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.

Container Service and management tools

Container Service provides rapid deployment of popular open-source container clustering and orchestration solutions. It uses Docker images to ensure that your application containers are fully portable. By using 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 you can scale these applications 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 Container Service, you need an Azure subscription. With Container Service, you can take advantage of the enterprise-grade features of Azure while still maintaining application portability, including at the orchestration layers.

Table 6-1 lists common management tools related to their orchestrators, schedulers, and clustering platform.

Table 6-1. Docker management tools

Management tools	Description	Related orchestrators
Azure Monitor for Containers	Azure dedicated Kubernetes management tool	Azure Kubernetes Services (AKS)
Kubernetes Web UI (dashboard)	Kubernetes management tool, can monitor and manage local Kubernetes cluster	Azure Kubernetes Service (AKS) Local Kubernetes
Azure portal for Service Fabric Azure Service Fabric Explorer	Online and desktop version for managing Service Fabric clusters, on Azure, on premises, local development, and other clouds	Azure Service Fabric
Container Monitoring (Azure Monitor)	General container management y monitoring solution. Can manage Kubernetes clusters through Azure Monitor for Containers.	Azure Service Fabric Azure Kubernetes Service (AKS) Mesosphere DC/OS and others.

Azure Service Fabric

Another choice for cluster-deployment and management is Azure Service Fabric. Service Fabric is a Microsoft microservices platform that includes container orchestration as well as developer programming models to build highly scalable microservices applications. Service

Fabric supports Docker in Linux and Windows Containers and can run in Windows and Linux servers.

The following are Service Fabric management tools:

- 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 and desktop multi-platform 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.

Monitor containerized application services

It's critical for applications split into multiple containers and microservices to have a way to monitor and analyze the behavior of the whole application.

Azure Monitor

Azure Monitor is an extensible analytics service that monitors your live application. It helps you to detect and diagnose performance issues and to understand what users actually do with your app. It's designed for developers, with the intent of helping you to continuously improve the performance and usability of your services or applications. Azure Monitor works with both web/services and standalone apps on a wide variety of platforms like .NET, Java, Node.js and many other platforms, hosted on-premises or in the cloud.

Additional resources

- **Overview of Azure Monitor**
  https://docs.microsoft.com/azure/azure-monitor/overview
- **What is Application Insights?**
  https://docs.microsoft.com/azure/azure-monitor/app/app-insights-overview
- **What is Azure Monitor Metrics?**
  https://docs.microsoft.com/azure/azure-monitor/platform/data-platform-metrics
- **Container Monitoring solution in Azure Monitor**
  https://docs.microsoft.com/azure/azure-monitor/insights/containers

Security and backup services

There are many support chores with lots of details that you have to handle to ensure your applications and infrastructure are in top notch condition to support business needs, and the situation becomes more complicated in the microservices realm, so you need a way to have both high-level and detailed views when you need to take action.

Azure has the tools to manage and provide a unified view of four critical aspects of both your cloud and on-premises resources:

- **Security**. With Azure Security Center.
- Get full visibility and control over the security of your virtual machines, apps, and workloads.
- Centralize the management of your security policies and integrate existing processes and tools.
- Detect real threats with advanced analytics.
- **Backup**. With Azure Backup.
- Avoid costly business disruptions, meet compliance goals, and protect your data against ransomware and human errors.
- Keep your backup data encrypted in transit and at rest.
- Ensure access based on multifactor authentication to prevent unauthorized use.
- **On-premises resources**. With hybrid cloud solutions.

Containerized Docker Application Lifecycle key takeaways

- Container-based solutions provide important cost-saving benefits because containers solve deployment problems caused by dependency failures in production environments, thereby improving DevOps and production operations significantly.
- Docker has become the de facto standard in the container industry and is 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 Kubernetes Service (AKS) and Azure Service Fabric are fundamental and indispensable for any microservices-based or multi-container applications that have significant complexity and scalability needs.
- An end-to-end DevOps environment that supports Continuous Integration/Continuous Deployment (CI/CD) and connects to the production Docker environments can provide agility and ultimately improve the time to market of your applications.
- Azure DevOps Services greatly simplifies your DevOps environment by deploying to Docker environments from your CI/CD pipelines. This statement applies to simple Docker environments as well as to advanced microservice and container orchestrators based on Azure.