Microsoft

Introduction to Windows Containers

John McCabe
Michael Friis

PUBLISHED BY
Microsoft Press
A division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Support at mspinput@microsoft.com. Please tell us what you think of this book at http://aka.ms/tellpress.

Acquisitions Editor: Kim Spilker
Developmental Editor: Bob Russell, Octal Publishing, Inc.
Editorial Production: Dianne Russell, Octal Publishing, Inc.
Copyeditor: Bob Russell

Visit us today at

microsoftpressstore.com

- **Hundreds of titles available** – Books, eBooks, and online resources from industry experts
- **Free U.S. shipping**
- **eBooks in multiple formats** – Read on your computer, tablet, mobile device, or e-reader
- **Print & eBook Best Value Packs**
- **eBook Deal of the Week** – Save up to 60% on featured titles
- **Newsletter and special offers** – Be the first to hear about new releases, specials, and more
- **Register your book** – Get additional benefits

Microsoft

Contents

Introduction

With the introduction of container support in Windows Server 2016, we open a world of opportunities that takes traditional monolithic applications on a journey to modernize them for better agility. Containers are a stepping stone that can help IT organizations understand what key items in modern IT environments, such as DevOps, Agile, Scrum, Infrastructure as Code, Continuous Integration, and Continuous Deployment, to name just a few, can do and how these organizations can adopt all of these elements and more to their enterprises.

As a result of Microsoft's strong strategic partnership with Docker—the de facto standard in container management software—enterprises can minimize the time required to onboard and run Windows Containers. Docker presents a single API surface and standardizes tooling for working across public and private container solutions as well as Linux and Windows Container deployments.

This is the next phase in IT evolution in which a direct replatform of code cannot be achieved and truly begins to bring the power of the cloud to any enterprise.

Acknowledgments

I would like to thank all of the folks at Microsoft and Docker who were involved in the development and production of this ebook. Your assistance and efforts were instrumental and greatly appreciated. In particular, I would like to thank Dee Kumar and Michael Friis for writing Chapter 2. It was a pleasure working with them, and their contribution added immeasurably to this project.

Free ebooks from Microsoft Press

From technical overviews to in-depth information on special topics, the free ebooks from Microsoft Press cover a wide range of topics. These ebooks are available in PDF, EPUB, and Mobi for Kindle formats, ready for you to download at:

http://aka.ms/mspressfree

Check back often to see what is new!

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://aka.ms/tellpress

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: http://twitter.com/MicrosoftPress.

Containers 101

We begin the journey to containers by giving you an in-depth introduction into what they are and the technology that surrounds and powers them. In addition, we provide a detailed real-world example that highlights why you should use containers today in your enterprise.

What is a container?

The most important question to ask here at the outset is a simple one: what is a container? It is important to truly understand what the technology is and how it differs from other technologies like hardware virtualization.

A container is another form of virtualization, but one that is focused at the operating system (OS) layer. It is geared toward deploying and running applications without requiring a full virtual machine (VM) for each application. In hardware virtualization, a *hypervisor* implements this "virtualization" layer, whereas in containers a *container engine* performs this.

Figure 1-1 illustrates how each application runs in an isolated memory space but shares the underlying kernel from the host machine/VM. The host machine (whether it's physical or virtual) regulates the container so that it doesn't consume all of the available resources.

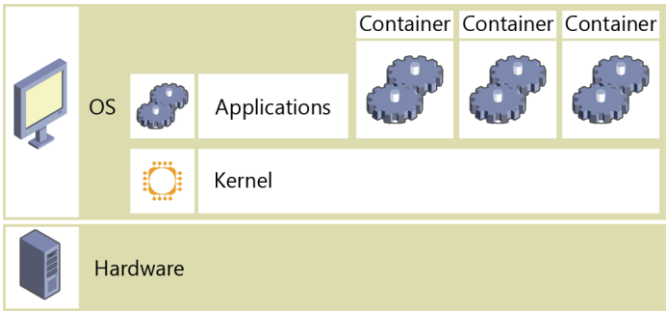

Figure 1-1: Container Virtualization Abstract

Each container has all of the necessary binaries to support the application running within it. Container hosts can run many different applications, fully isolated, at any one time. In a modern datacenter, containers can achieve a greater density per host than VMs because the footprint of a container is considerably smaller than that of a VM.

Containers versus VMs

In many ways, containers can be incorrectly thought of as a VM. VMs are independent operating systems—memory, CPU, and so on, whereas containers only appear to be. A container for example will share a kernel from the host machine, whereas a VM has its own kernel and doesn't see the host machine at all, as demonstrated in Figure 1-2.

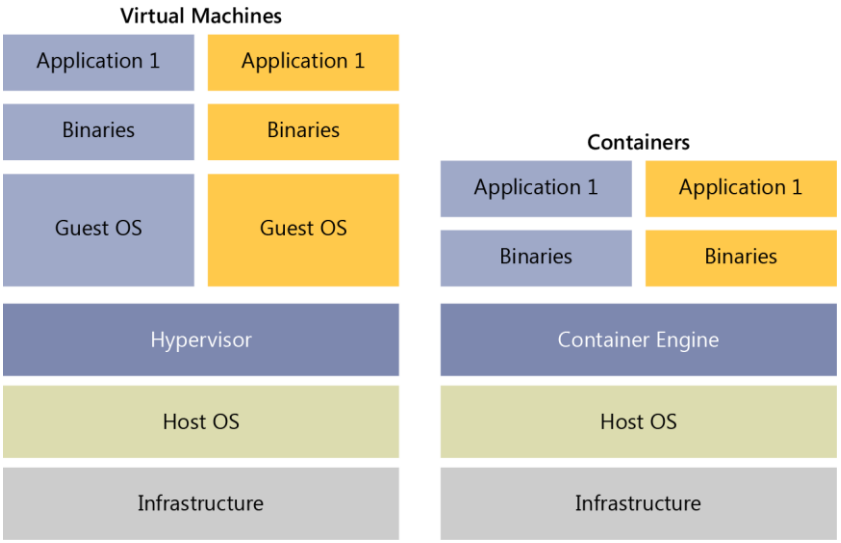

Figure 1-2: Comparing VMs and containers

Because VMs have been around for a long time, and even today we generally deploy applications to them, we are forced to ask the question: why choose a container over a VM, or vice versa?

VMs offer flexibility to an enterprise and allow it to run any type of application within them. They can be assigned resources based on the need and demand, and they tend to lend themselves well to the scale-up approach when you need to improve performance.

VMs require individual management, which generally increases the associated overhead of running an application. On the same note, a VM will take up a predefined set of resources while it is running. Thus, the number of VMs that you can store on a single host is directly related to the size of the VMs. This can have additional effects with respect to the cost of an IT environment.

Containers, conversely, are very flexible and naturally have a much smaller footprint than a VM. They don't have the same management footprint, because you no longer need to manage a full OS, thus reducing potentials costs associated to the container lifecycle. As mentioned earlier, this allows for greater density per host and makes it possible for you to scale-out far more efficiently than you can with a VM deployment.

Isolation is a key aspect for containers; it provides separation for each application that you want to run within a container. The isolation essentially gives the application its own view of the OS from the perspective of memory, CPU and file system, among other things. The application can perform any operation in this "bubble," even delete what it thinks is the OS without affecting any other containers that are running.

However, containers do lock you in to a type of OS, which in turns locks you in to the type of applications you might be able to use. This is not necessarily a bad thing, because it could simplify costs in terms of licensing and support. Containers also don't maintain state; you are required to maintain state outside of the container runtime. Again, this is not necessarily a bad thing, but an assessment on a per-application basis is required to see if this is achievable for the application.

Containers also give an enterprise a migration path toward making its applications cloud-native without a major replatform. It also facilitates an agility to migrate applications between clouds. For example, if you containerize your application by using Docker and Windows as the host OS, as long as you can find a provider that uses Docker and Windows as the host OS, you can run your application.

Containers also require applications that are intended to be containerized to have noninteractive user interface (UI) or service-based applications. This might initially put people off, but, again, this is not a bad thing; however, it will of course require some planning and understanding of how your application is deployed and operates. It is crucial to understand that containers are best suited for cloud-native scenarios. These scenarios eliminate a large amount of unnecessary code and usually break an application into lots of small functional pieces. These pieces have a very specific job and, because of this, are very performant. With this simple understanding, we can begin to understand why containers are best geared toward headless apps.

Figure 1-3 shows a simple comparison chart to help you understand when you should consider using containers.

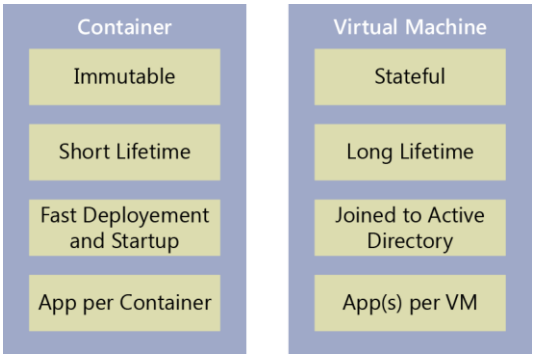

Figure 1-3: Container versus VM

So, what should you choose for an enterprise today? It very much depends on your needs. For example, if you are looking toward the future and want to be cloud-ready, containers are a better choice. They fit the DevOps model for rapid development and deployment, and allow for greater agility when moving between infrastructures, be it public or private cloud. If you want to provide additional isolation so that applications cannot interfere with one another, containers are a good fit.

Containers are a natural evolution or transition from a waterfall-based development cycle with an *N*-tier architecture that allows enterprises to get ready and shift into the mindset of cloud-based Agile architectures.

Why containerize? A real-world story

Containers provide a valuable proposition to enterprises today, giving them agility in their application estate that is not achievable easily in a VM type of deployment. Containers make it possible for enterprises to take the first steps on their cloud journey and move from a traditional monolithic architecture to a more modern *microservices architecture* without major recoding of their applications initially. Although containers are not microservices, they help the development teams to begin understanding how to break down their applications into smaller parts and containerize those components. This in turn gives those teams a chance to understand how to scale via the cloud and, when appropriate, translate those containerized components into a layer in a microservices architecture.

One such company, Tyco, began the journey to replatform its flagship and legacy applications by using containers as the stepping stone on a long-term journey toward microservices.

Tyco provides more than three million customers around the globe with the latest fire-protection and security products and services, placing it among the largest in its industry sector. A $10 billion company, Tyco has more than 57,000 employees in more than 900 locations across 50 countries, serving a range of markets, including commercial, institutional, governmental, retail, industrial, energy, residential, and small business.

Its flagship software application is very complex and contains hundreds of thousands of lines of code. When the company needs to update a certain component of an application, it must reinstall the entire application at its customer's site, which can take a significant amount of time. Some of Tyco's customers need to post guards at the doors when they are doing an upgrade, because the facilities must be protected 24/7. Major upgrades can be extremely complicated, which deter some of their customers from implementing them, even if the changes are necessary.

Tyco faced another problem with some older applications with respect to scalability. One example is the company's access-control monitoring application, C•CURE 9000, which is used by approximately 18,000 clients worldwide. This is a classic three-tier Microsoft .NET application that runs on the Windows Server OS and Microsoft SQL Server. C•CURE 9000 monitors doors, windows, card readers, and other access points in a customer's physical plant. Due to demands of the market and customer installation growth, the engineering team needed to rethink how to scale this critical application to meet increased customer demand.

Tyco found a better way to achieve the required scalability: divide and conquer. It decided to move to a microservices architecture, break apart the large legacy applications, and modernize them into bite-sized pieces. With a microservices architecture, organizations can scale and update part of an application independently of the other parts.

Using C•CURE 9000 as its first microservices test case, Tyco developers looked for natural seams in the monolithic application and separated it into smaller functional pieces that can be developed, released, updated, and scaled independently. They then migrated the components to VMs, which were easier to scale and manage than physical servers.

Not long after Tyco had C•CURE 9000 successfully running in VMs, the company learned that Microsoft was including Docker container technology in Windows Server 2016. Containers are autonomous files that contain an entire runtime environment: an application plus all its dependencies, libraries, and configuration files needed for it to run identically in any environment. By containerizing an application and its dependencies, developers can abstract differences in OS distributions and underlying infrastructure. Containers are also much lighter weight than VMs in terms of resource requirements.

The Tyco software team received a version of Windows Server 2016 and migrated C•CURE 9000 into Windows Server Containers using the Docker engine. Working alone, it took Tyco about two weeks to onboard into containers, and most of that time was spent learning how to write Docker files. There were only a couple of days of actual code changes.

By running C•CURE 9000 in Windows Server Containers, Tyco immediately gave this key legacy application elasticity and a greatly simplified architecture that is now less expensive to manage and run. By using Windows Server Containers and Docker, the company gained better consistency and control between developers, testers, and deployment teams—a full DevOps environment—without changing the application. For Tyco and its customers, it now has the ability to increase application availability and push updates faster, with minimal-to-zero downtime.

In addition to gaining DevOps build and deployment flexibility, containerizing C•CURE 9000 gave Tyco the freedom to run the application in any environment, so the company chose to move it from a Tyco datacenter to Microsoft Azure. Tyco already uses Microsoft Visual Studio and Azure Virtual Machines for development, thus it was a natural step to run this application in Azure. By using Azure, Tyco

benefits from infrastructure on demand and the ability to convert its infrastructure expenses from capital to operating costs.

Tyco is one of many enterprises today that are making the transition toward microservices. The company also is taking advantage of containerization to achieve scalability and reliability while making its long-term journey to native Platform as a Service (PaaS) applications.

Container types

There are two main types of containers available today:

- Windows Server Containers
- Windows Server Containers with Hyper-V Isolation

Although this eBook doesn't cover Linux, container technology exists for that platform and operates much like the Windows Server Containers, except with the Linux OS.

Figure 1-4 presents a comparison between Windows Server Containers and Windows Server Containers with Hyper-V Isolation.

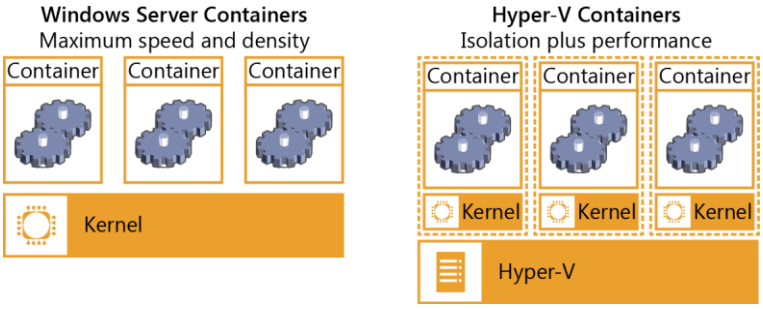

Figure 1-4: Comparing Windows Server Containers and Windows Server Containers with Hyper-V Isolation

Windows Server Containers share a kernel with the container host and all running containers. It provides isolation for the application through process and namespace isolation technology. The process and namespace isolation give the application its own view of the OS and binaries, and enforces protection from other containers.

Windows Server Containers with Hyper-V Isolation provide an isolated kernel experience through a utility VM on a host. This increases security of the container because the isolation mechanisms are enforced at the hardware level, instead.

Both container technologies operate with the same API surface and can be controlled via Docker today. This simplifies management of a containerized estate.

Container host architecture

Container hosts can be deployed in many different configurations, depending on the mix of Windows Server Containers and Windows Server Containers with Hyper-V Isolation that you want.

With Windows Server 2016, you can set up nested virtualization and run a guest Hyper-V VM to be the container host for all Windows Server Containers with Hyper-V Isolation, and similarly a guest running Windows Server 2016 to be a Windows Server Containers host.

Figure 1-5 provides a view of the architecture when using Windows Server 2016 with Hyper-V to implement both a Windows Server Container host and a Windows Server Containers with Hyper-V Isolation host.

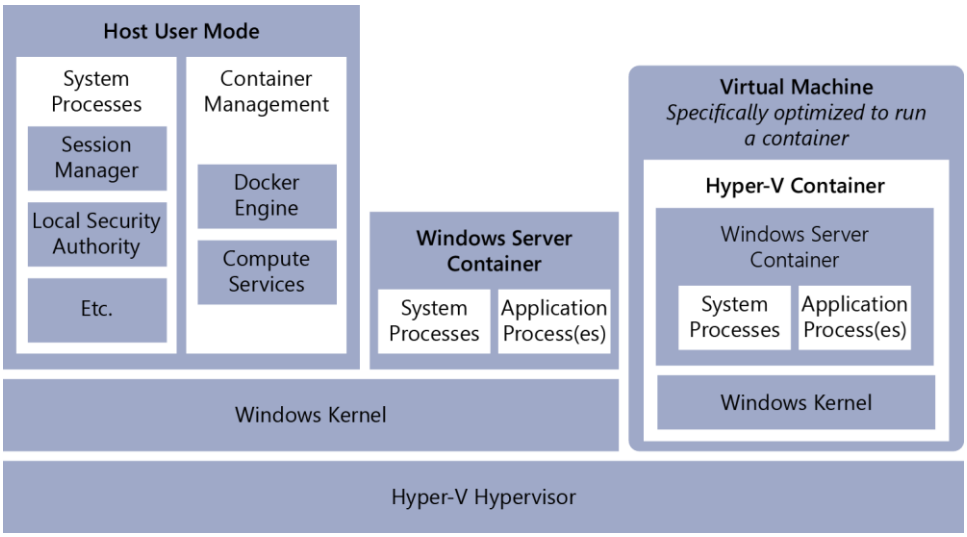

Figure 1-5: Container host mixed architecture

As seen in earlier diagrams and in more detail in Figure 1-5, we see the isolation provided by a Windows Server Containers with Hyper-V Isolation, right down to the kernel level, backed up by the hypervisor and the shared kernel for the Windows Server Container.

Container management

Container management in Windows Container host, be it a Windows Server Container or a Windows Server Containers with Hyper-V Isolation deployment, will use Docker as the main management tool for administering the lifecycle of the containers that are running.

The Docker client and engine can manage both Windows Server Containers with Hyper-V Isolation and Windows Server Containers through a standardized API. The Docker engine calls via an abstraction layer the Compute Service APIs, as shown in Figure 1-6.

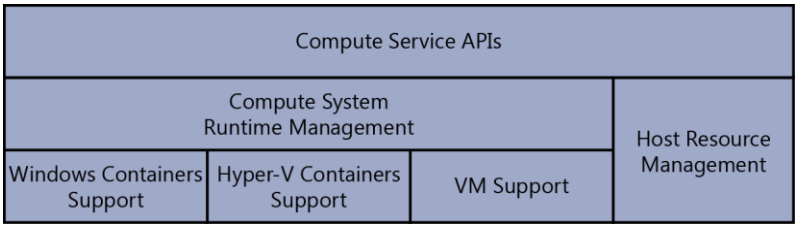

Figure 1-6: Host Compute Service and component relationships

In turn, the Compute Service API calls into the Compute System Runtime and then into the relevant area. This API allows for the management of containers, including but not limited to creating, deleting, starting, and stopping a container. These layers combine to form the Host Compute Service (HCS)

Container images

Container images are the basis of running containers. Container images are built upon a base OS layer, which is provided and maintained by Microsoft in an image registry.

The image registry is a central place where base OS images as well as custom container images can be stored.

A custom container image starts with the base OS layer being deployed to a container host. The enterprise can then install all its application dependencies and components. Following installation, because a container by default is immutable, you need to commit these changes to start a container with the deployed application.

Committing these changes creates a new layer with a dependency link to the base OS layer. This means that if an application is invoked, it will check whether the base OS layer has been loaded already into the container host before trying to invoke the application container. If the base OS layer is not running, the container engine will invoke the base OS layer container from the image repository and subsequently the application container. If you go to invoke multiple instances of that container, every additional container will need to start only the application container because the base OS layer is already running.

Figure 1-7 depicts a sample of these layers and how we have three different container images showing the base OS, Internet Information Services (IIS), and then the application framework.

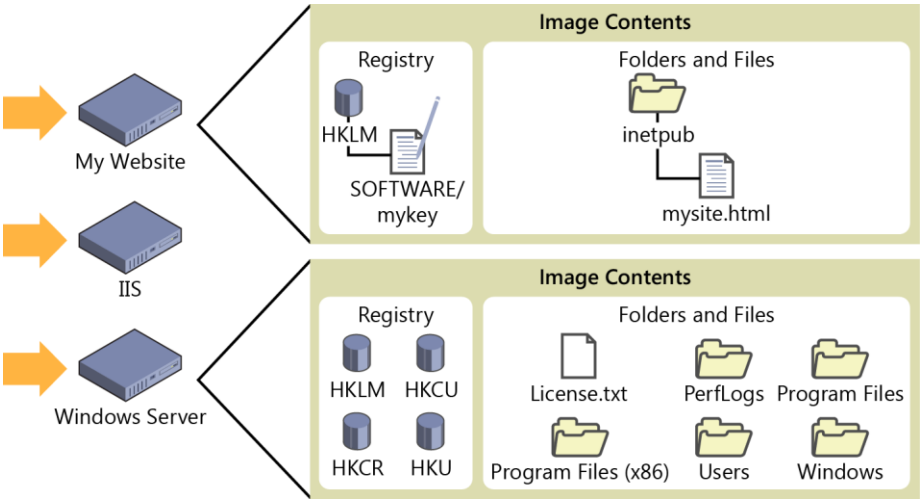

Figure 1-7: Container image layers

We could easily combine IIS and the MyWebsite layer in Figure 1-7 together into one layer; however, this might not be the best approach. When thinking about your container images, it is best to understand the dependencies and how you can create a layer that will work across many potential applications.

Currently Microsoft provides the following images for containers:

- Windows Nano Server 2016
- Windows Server 2016 Core

In addition to this, you must—depending on your host—understand what container types with what images are supported. Table 1-1 lists what is supported as of this writing.

Table 1-1: Container host image support

Host	Server core base image	Nano base image
Windows Server Core	Windows Container	Windows Server Containers with Hyper-V Isolation
Windows 10 Desktop	Windows Server Containers with Hyper-V Isolation	Windows Server Containers with Hyper-V Isolation

Container networking

Container networking follows similar principals to networking with a VM. Figure 1-8 provides a simplified view of networking for a container host.

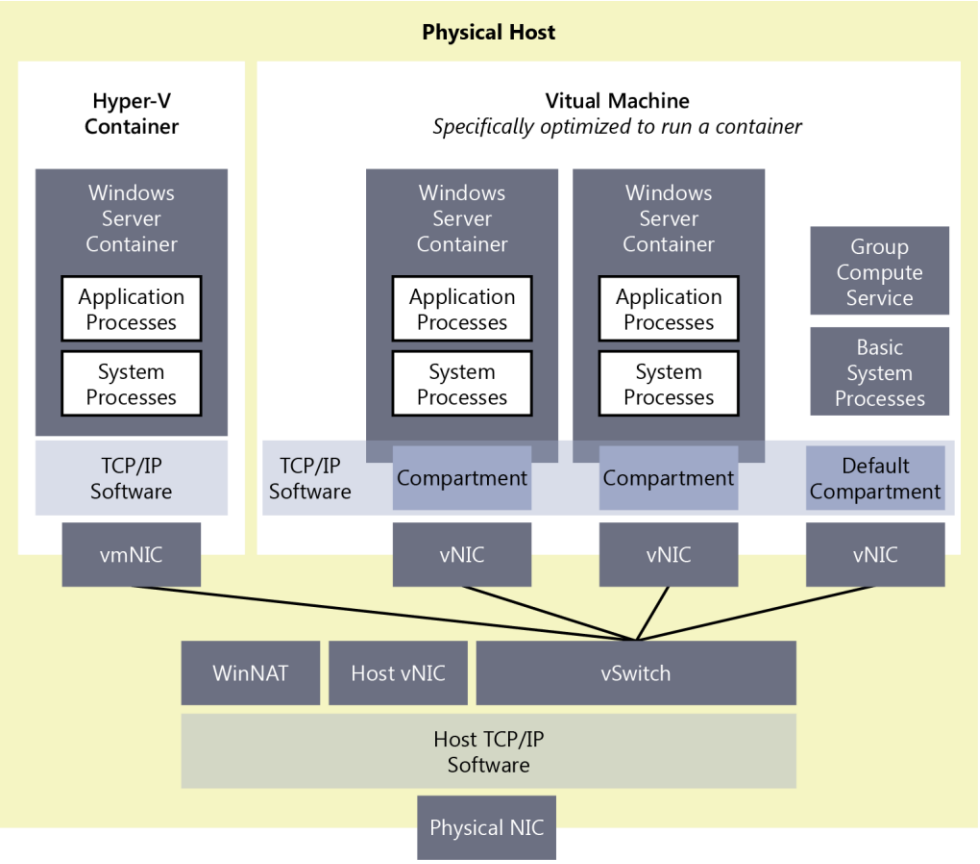

Figure 1-8: Container networking

The host, whether it's a VM or a physical host, will have a virtual switch (vSwitch). This vSwitch provides internal or external connectivity for the containers. There are different modes of external connectivity:

- **Network Address Translation (NAT) mode** Each container is connected to an internal vSwitch and uses WinNAT to connect to a private IP subnet. WinNAT performs both NAT and Port Address Translation (PAT) between the container host and the containers themselves.
- **Transparent mode** Each container is connected to an external vSwitch and is directly attached to the physical network. IP addresses can be assigned statically or dynamically by using an

external DHCP server. The raw container network traffic frames are placed directly on the physical network without any address translation

- **L2 bridge mode** Each container is connected to an external vSwitch. Network traffic between two containers in the same IP subnet and attached to the same container host is directly bridged. Network traffic between two containers on different IP subnets or attached to different container hosts is sent out through the external vSwitch. On egress, network traffic originating from the container has the source MAC address rewritten to that of the container host. On ingress, network traffic destined for a container has the destination MAC address rewritten to that of the container itself.
- **L2 tunnel mode** (This mode should be used only in a Microsoft Cloud Stack.) Similar to L2 bridge mode, each container is connected to an external vSwitch with the MAC addresses rewritten on egress and ingress. However, *all* container network traffic is forwarded to the physical host's vSwitch, regardless of Layer 2 connectivity. This allows network policy to be enforced in the physical host's vSwitch, as programmed by higher levels of the network stack (for example, network controller or network resource provider).
- **Overlay Mode** This mode is for helping to create a Docker Swarm Cluster. Essentially, you are using VXLAN technology on the host to span a Docker network across multiple hosts using our Windows Host Networking Service and the Virtual Filtering Platform Extension off the Hyper-V Switch. This network then makes it possible for you to place your workload on any host without changing configurations

Each container has its own virtual Network Interface Card (vNIC) that is isolated and connected to the vSwitch. For a Windows Server Containers with Hyper-V Isolation, because it uses a utility VM to wrap the container, the vNIC is a synthetic VM NIC exposed directly to the container itself.

Container security

As with all new technology, especially in relation to the cloud, security becomes a crucial topic to understand. Comprehending certain elements of security when it comes to containers will facilitate the seamless adoption of containers as a technology for the enterprise. In this section, we discuss some of the most common items with respect to container security.

Identity

Containers are designed to be standalone, nondomain-joined entities. This leads to a simple question of how do you allow containers access to domain-joined resources. In general, users want to ensure that a service can be accessed only by authorized individuals and applications. For example, they will want the appropriate database administrator to manage a SQL database, and only authorized apps to write to it.

When users want to give consent for an app to access it, they follow one of two general approaches:

- Create a shared secret (password, certificate, or otherwise) and control who has access to it in lieu of an account
- Create a service account in Active Directory and ensure that only authorized administrators and servers have access to use it
  - Any user account could be used as a service account with manual password rollover
  - Active Directory Managed Service Accounts offer password rollover, but are tied to a single computer account

- Active Directory Group-Managed Service Accounts (gMSA) offer automatic rollover and can be included as part of an Access Control List to a group of machines and other accounts such as users

Figure 1-9 demonstrates how a container uses the service account to authenticate to a SQL database.

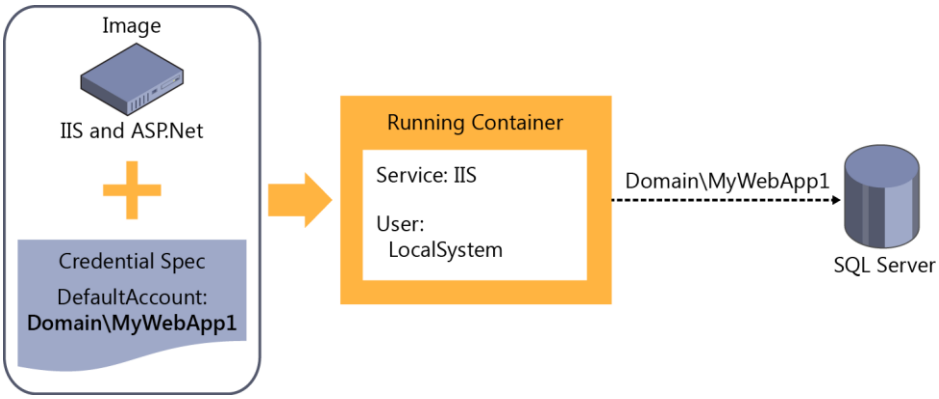

Figure 1-9: Managed Service Accounts and containers

With service accounts, it's important to manage and audit who can control the server. Because all server administrators effectively have access to the service account, customers include them in the scope when doing audits on the attack surface of service accounts. Because the server usually has a single purpose, users are satisfied with this boundary.

Isolation

The security model for containers is *effectively* the same as VMs. This means that if you want to achieve the same hardware-backed isolation enjoyed today, you must use Windows Server Containers with Hyper-V Isolation. This becomes crucial in multitenant environments where there are governance policies enforcing this.

From a host perspective, administrators have full management rights, and the container host itself is completely trusted. The Docker daemon and Host Compute Service (see Figure 1-6) can start all containers.

- It can be assumed that the host also has access to processes running and data stored in a container
- Container administrators can manage all containers on the host
  - Including all management actions—start/stop/modify
  - Including the ability to read files and create processes in the container
- Container administrators could mistakenly start a container with the wrong network or storage settings

Containers are different in that each one should generally represent one app, and the host is implicitly trusted. Container host administrators have full control of all containers, and there is nothing preventing a host administrator from starting a process in a container while its running. It's also easy to modify an existing container and redeploy it. Therefore, it is much easier for an administrator to identify what app is running in a container as well as to tamper with it or leak data from it. Many

customers are accepting of this risk today, but security is an area that Docker and other companies are continuing to improve.

Code integrity

The Trusted Installer service on the container host is used to validate code integrity of the Windows Server Core and Nano Server base images as they're installed on the host today. If the base image is tampered with, it will be detected and repaired by the existing code integrity features.

Code identification and vulnerability scanning

Another question that arises with respect to reusing containers is identifying where the code came from. If you download a container image from a repository, how do you truly know that the code has not been modified maliciously.

For example, the immutability of a container would infer that if you recycle the container, it will start with the application in a known good state, but if that state is tainted while building the container image, it obviously can introduce security problems into the environment. An additional example would be if the code in the container were modified but the container was not recycled. The rogue code could lead to potential problems and security holes within the environment.

Enterprises can use tools like Docker Security Scanner to validate an image from a private repository from known vulnerabilities and exposures.

More info To learn more, go to https://docs.docker.com/docker-cloud/builds/image-scan/.

High availability with containers and container hosts

Containers being a cloud technology are designed to be scalable. A simple example would be the use containers for servicing as a web portal for an ecommerce website. When customer demand increases, additional container instances are started and the requests are served via these instances. When demand decreases, the container instances are stopped.

Traditionally, the use of load balancers and VMs would be configured and deployed to meet these requirements, including some orchestration technology, in order to handle the start/stop/deployment of the VMs.

For container hosts and containers, we can utilize technology built in to the Docker engine to configure a Docker swarm cluster. The Docker swarm cluster orchestrates the deployment of new containers, including load balancing of the incoming traffic to the respective containers.

More info To learn more, go to https://docs.docker.com/engine/swarm/key-concepts/.

Antivirus programs

Containers can have antivirus programs installed within the image. However, it introduces some challenges that each antivirus vendor will need to overcome to avoid affecting the performance of the container. For example, because containers share a view of the same data from a container host, it is possible to perform redundant virus scans on this data.

To install antivirus capabilities in a container, Microsoft has published guidance available to configure the solution and avoid redundant scans.

> **More info** To learn more, go to https://msdn.microsoft.com/windows/hardware/drivers/ifs/anti-virus-optimization-for-windows-containers.

Patching containers and container hosts

In traditional operating systems, a vendor usually releases software patches on a regular cadence to fix bugs, improve security and stability, and increase performance. Customers would choose when to download and install these patches to suit their maintenance windows. For large organizations, this is a time-consuming process and generally one that requires a lot of post reporting to ensure that the systems have successfully been patched.

When you think about patching and container technology, the traditional rules change. With a traditional OS, you would either run Windows Update or use a tool like System Center Configuration Manager to target and patch the system.

Because the container image is immutable and any state changes you perform on the container are lost after restart, you need to understand how to keep these images up to date.

Container OS image

As we have previously mentioned, a container can have many layers, beginning with the base OS image, the framework layer, and then possibly the application layer. You might therefore think that by simply updating the base layer and then the framework layer, the application layer will be automatically updated because we have a dependency on the base OS image. In fact, this is not the case. Indeed, if you update the base OS image either via a patch or by downloading the latest version available from the image repository, it will break the framework layer and the application layer.

Because containers are a mix of an infrastructure technology and a development technology, you need to examine the possibility of introducing a DevOps model to update the base image and provide subsequent build tasks, which will deploy the necessary framework and application into their respective layer. Figure 1-10 shows you the basic layers and the re-creation of them when we apply the patch to the base image.

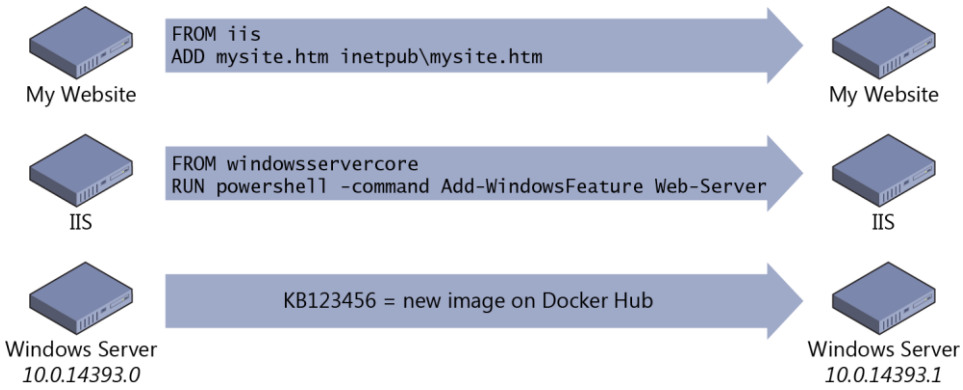

Figure 1-10: Patching and rebuilding container layers

To roll this new image into production, an enterprise can maintain the old images, start the new patched containers, redirect traffic to the new containers, and then phase-out the old container images and remove them from your image repository.

[Less optimal] Patching a container as a new layer

Another approach to patching a container, is simply to patch the container application image layer and commit the changes. Figure 1-11 illustrates how to maintain existing layers and create this new patched layer.

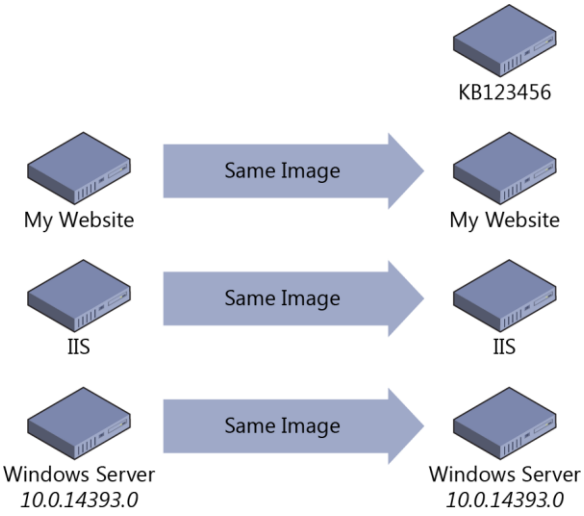

Figure 1-11: Patching a container image as a new layer

Selecting which method to patch depends on which stage an enterprise is at with respect to DevOps. For example, if the build cycle is not automated, achieving the first patching option might be difficult, whereas the second option would be very easy to achieve.

> **Note** This option is less recommended because of the unnecessary bloat it creates; it will not be supported when using a Nano server image from RS3 release onward, because we are removing the servicing stack from Nano Server. To learn more, go to https://docs.microsoft.com/en-us/windows-server/get-started/nano-in-semi-annual-channel.

Docker 101

In this chapter, we give a primer on Docker containers. We discuss the partnership between Docker and Microsoft and introduce all the tools available in the Docker ecosystem.

What is Docker?

Docker is the leading containerization platform for automating and managing the deployment of applications as portable, self-sufficient containers that can run on any cloud or on-premises. Docker also is the name of the company that is developing this technology, in tight collaboration with the cloud community, Linux, and Windows vendors, like Microsoft. Docker is becoming the standard unit of deployment and is emerging as the de facto standard implementation for containers on developer desktops, datacenters, and in the cloud. But what are containers? Chapter 1 presents a solid description, but let's revisit this question again here.

To give the computer science definition, containers are an operating system–level isolation method for running multiple applications on a single control host. With developers building, and then packaging their applications into containers, and providing them to IT to run on a standardized platform, containers reduce the overall effort to deploy applications and can streamline the entire development and test cycle, ultimately reducing costs. Because containers can run on a host operating system (OS), which itself could be physical or virtual, it provides IT with flexibility and the opportunity to drive an increased level of server consolidation, all while maintaining a level of isolation that allows many containers to share the same host OS. Figure 2-1 illustrates the breadth of what can potentially run in a container.

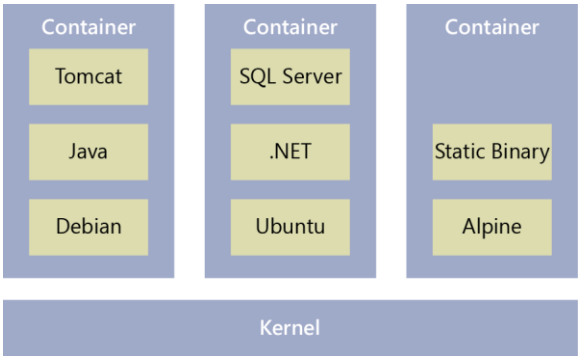

Figure 2-1: Potential container workload

Let's take a look at some of the benefits that you can derive from utilizing containers.

Lightweight

Docker containers running on a single machine share that machine's OS kernel; they start instantly and use less compute and RAM. Images are constructed from filesystem layers and share common files. This minimizes drive usage, and image downloads are much faster.

Standard

Docker containers are based on open standards and run on all major Linux distributions, Windows, and on any infrastructure including VMs, bare-metal, and in the cloud.

Secure

Docker containers isolate applications from one another and from the underlying infrastructure. Docker provides the strongest default isolation to limit app issues to a single container instead of the entire machine.

Docker Enterprise Edition

Docker Enterprise Edition (Docker EE) is designed for enterprise development and IT teams who build, ship, and run business-critical applications in production at scale. Docker EE is integrated, certified, and supported to provide enterprises with the most secure container platform in the industry, to modernize all applications. An application-centric platform, Docker EE is designed to accelerate and secure the entire software supply chain, from development to production running on any infrastructure. Docker EE is fully supported on the Microsoft Platform through Windows Server 2016 and Microsoft Azure. Docker EE also supports CentOS, Red Hat Enterprise Linux, Ubuntu, SUSE Linux Enterprise Server, Oracle Linux, and Amazon Web Services (AWS).

Certified Infrastructure, Containers, and Plug-ins

Docker EE is optimized for operating systems and cloud providers by Docker as Certified Infrastructure. Additionally, the Docker Certification Program will certify containers and plug-ins from Docker's ecosystem partners; for example, Independent Software Vendors (ISV) containers that run on top of the Docker platform and networking and storage plug-ins that extend the Docker platform. Docker and Microsoft provide cooperative support so that you can confidently use these products in production. Certified Containers and Certified Plugins are available on the Docker Store for you to download and install. Here are the benefits you can realize with each:

- **Certified Infrastructure** This provides an integrated environment for enterprise Linux (CentOS, Oracle Linux, RHEL, SLES, Ubuntu), Windows Server 2016, and cloud providers like AWS and Azure.
- **Certified Container** Certified Containers provide trusted ISV products, packaged and distributed as Docker containers—built with secure best practices cooperative support.
- **Certified Plugin** This provides networking and volume plug-ins and easy-to-download-and-install containers to the Docker EE environment.

Figure 2-2 illustrates the Certified Plugins, Certified Containers, and Certified Infrastructure that Docker EE brings to an Enterprise.

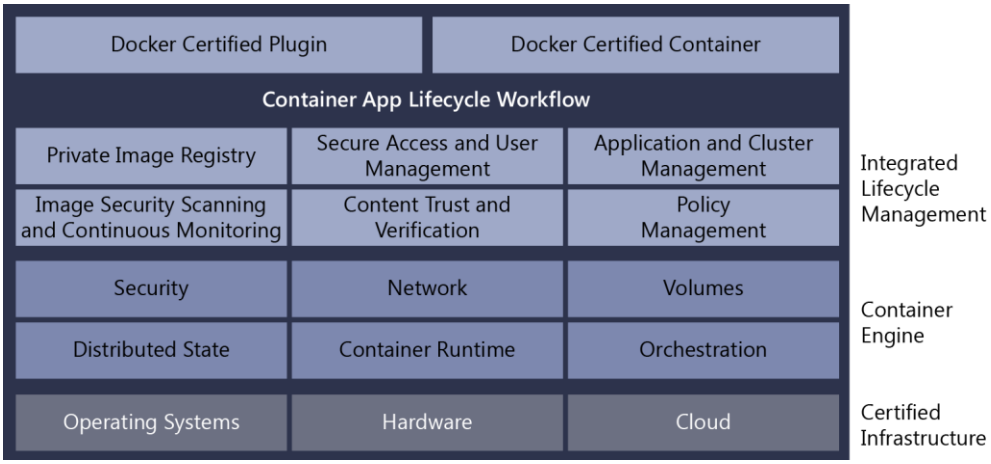

Figure 2-2: Docker Certified Infrastructure, Containers, and Plugins

Integrated container management with Docker Datacenter

Docker Datacenter, part of Docker EE, provides integrated container management (see Figure 2-3) and security from development to production. Enterprise-ready capabilities such as multitenancy, security, and full support for the Docker API give IT teams the ability to scale operations efficiently without breaking the developer experience. Open interfaces allow for easy integration into existing systems and the flexibility to support any range of business processes. Docker EE provides a unified software supply chain for all apps—from commercial off-the-shelf, to homegrown monoliths, to modern microservices written for Windows or Linux environments, on any server, VM, or cloud.

Here are just some of the benefits you enjoy with Docker EE:

- Integrated management of all app resources from a single web admin UI
- Frictionless deployment of apps and Compose files to production with just a few clicks
- Multitenant system with granular Role-Based Access Control (RBAC) and Lightweight Directory Access Protocol (LDAP)/Active Directory integration
- Self-healing application deployments with the ability to apply rolling application updates
- End-to-end security model with secrets management, image signing, and image security scanning
- Open and extensible to existing enterprise systems and processes

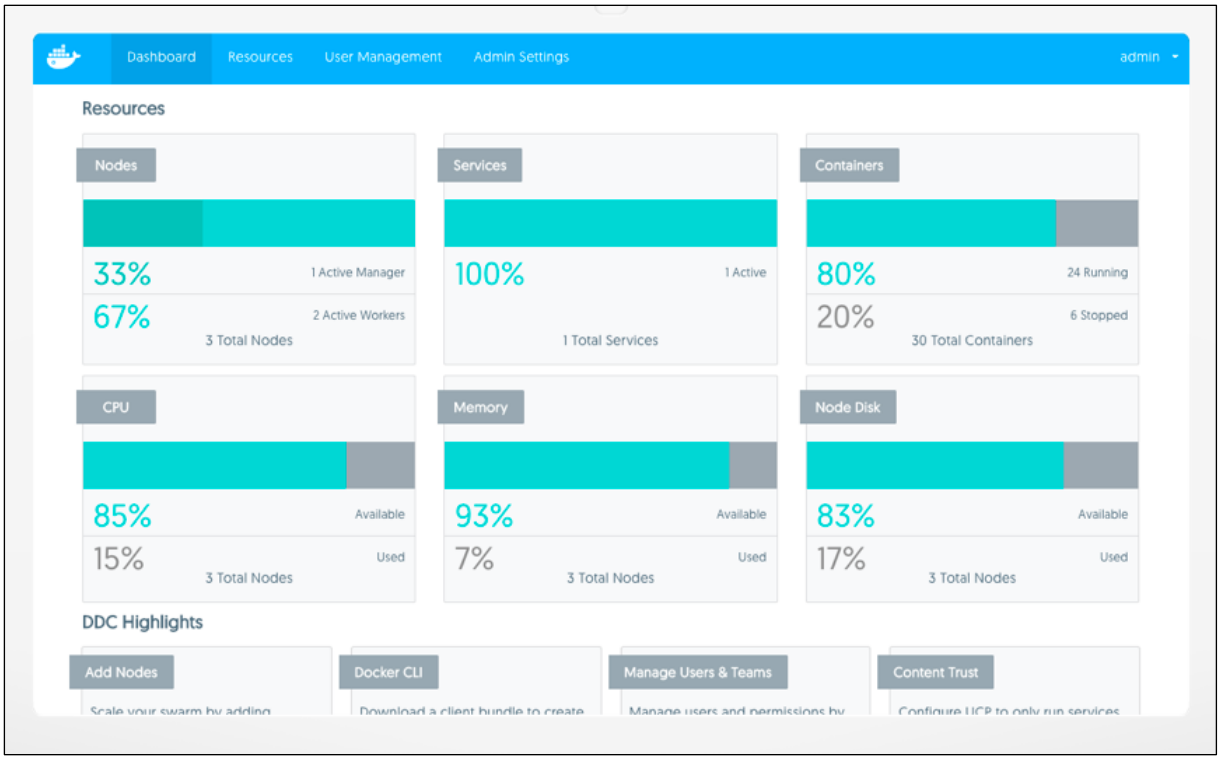

Figure 2-3: Integrated container management

Docker EE is available as a free trial and for purchase from Docker Sales, online via the Docker Store (supported by Microsoft), and by Docker's network of regional partners.

Docker EE is available in three tiers:

- **Basic** The Docker Platform for Certified Infrastructure, with support from Docker, Inc. and Certified Containers and Plugins from the Docker Store
- **Standard** Adds Docker Datacenter capabilities of advanced image and container management, LDAP/Active Directory user integration, and RBAC
- **Advanced** Adds Docker Security Scanning and continuous vulnerability monitoring

Figure 2-4 shows more detail of what is available in each tier.

	COMMUNITY EDITION	ENTERPRISE EDITION BASIC	ENTERPRISE EDITION STANDARD	ENTERPRISE EDITION ADVANCED
Container engine and built in orchestration, networking, security	✓	✓	✓	✓
Docker Certified Infrastructure, Plugins and ISV Containers		✓	✓	✓
Image Management (private registry, caching)	Cloud hosted repos		✓	✓
Docker Datacenter Integrated container app management			✓	✓
Docker Datacenter Multi-tenancy with RBAC, LDAP/AD support			✓	✓
Integrated secrets mgmt, image signing policy			✓	✓
Image security scanning	Preview			✓
Support	Community Support	Business Day or Business Critical	Business Day or Business Critical	Business Day or Business Critical

Figure 2-4: What's available in each Docker EE tier

For the developer and "do it yourself" ops community, Docker has renamed its free software to Docker Community Edition (CE). It is available for Mac and Windows, for Azure, and for CentOS, Debian, Fedora and Ubuntu, all of which are you can download from the Docker Store.

What is the Docker Universal Control Plane?

The Docker Universal Control Plane (UCP) is the enterprise-grade cluster management solution from Docker that is available in Docker EE. You install it behind your firewall, and it helps you manage your entire cluster from a single place.

You can install Docker UCP on-premises, or in a virtual private cloud. With it, you can manage thousands of nodes as if they were a single one. You can monitor and manage your cluster using a graphical UI.

More info To learn more about Docker UCP, go to https://docs.docker.com/ucp/overview/.

What is Docker Trusted Registry?

Docker Trusted Registry (DTR) is the enterprise-grade image storage solution from Docker. You install it behind your firewall so that you can securely store and manage the Docker images that you use in your applications.

DTR architecture

DTR is a containerized application that runs on a Docker UCP cluster, as shown in Figure 2-5.

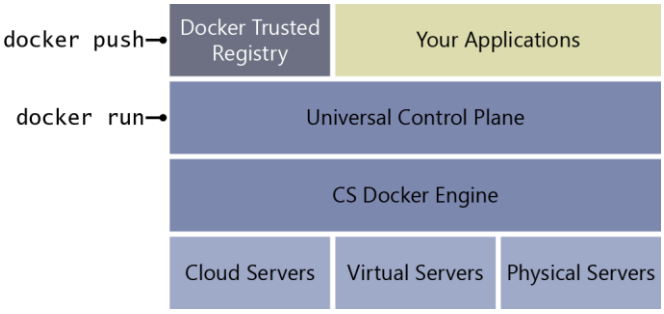

Figure 2-5: Docker Trusted Registry

After you have deployed DTR, you use your Docker command-line interface (CLI) client to sign in, push, and pull images.

What is the Docker partnership?

In 2014, Microsoft and Docker announced a partnership in which they would release a fully supported version of Docker EE to run on Windows Server 2016. This partnership makes it possible for customers of Microsoft to receive enterprise support from Microsoft, backed by Docker, for running containerized workloads on Windows Server 2016.

The commercially supported Docker engine was made available to Windows Server 2016 customers at no additional charge. Using this, customers can build, manage, and run containerized applications in a Windows Server 2016 production environment with a full support ecosystem of 450 registered partners backing the Docker engine release.

Microsoft had observed Docker being an industry leader with containers on Linux for many years. With many of Docker's tools having a vibrant open source ecosystem, these tools have become the staple choice of many enterprises who have already embarked on the containerization journey.

These compelling reasons, among a variety of others, including Azure support for Linux Containers on VMs, drove the two companies together to form a strong collaborative partnership. With Windows Server Containers standardizing on management toolsets and building on top of the already vast Docker ecosystem, a winning formula for the partnership has emerged.

The extensive partnership integrates across the Microsoft portfolio of developer tools, operating systems, and cloud infrastructure, including the following:

- Windows Server 2016
- Hyper-V
- Microsoft Visual Studio
- Azure

Microsoft and Docker aim to provide a modern platform for developers and IT pros to build, ship, and run distributed applications on-premises, in the cloud, or through service providers across both Windows and Linux operating systems. Together, the two companies are bringing container applications across platforms, integrating across Docker's developer tools, the OS, and cloud infrastructure to provide a seamless experience that spans the application environment from development to test and production.

One platform, one journey for all applications

Every version of Windows Server 2016 grants access to Docker EE, making possible the use of containers in the Windows Server app development and management ecosystem. Organizations can now securely build, ship, and run any app, across any infrastructure—from desktop to datacenter to public cloud. Figure 2-6 highlights some trends about application modernization in an enterprise.

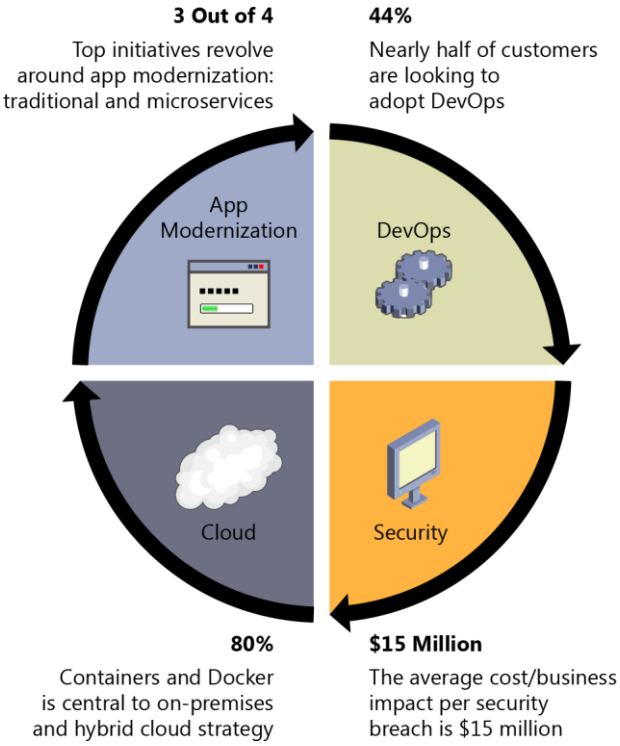

Figure 2-6: Top drivers for customer application modernization

With Docker—a platform for running applications in lightweight containers—and Windows Server 2016, you can give traditional apps a new lease on life, adding features, increasing security and performance, and moving toward continuous deployment, without a lengthy and expensive rebuild project. The partnership provides the agility, portability, and control of the Docker platform to Windows developers and IT pros.

Together, Docker and Microsoft address 98 percent of enterprise app requirements. Windows Server Containers help secure and modernize existing enterprise .NET and line-of-business server applications with little or no code changes. You can package existing apps in containers to realize the benefit of a more agile DevOps model, and then deploy on-premises, to any cloud, or in a hybrid model. And you can reduce infrastructure and management costs for those applications, as well. Figure 2-7 highlights this relationship in more detail.

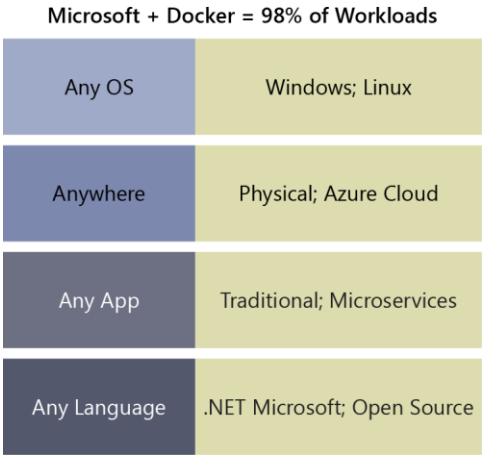

Figure 2-7: Containers anywhere, any app

Windows Server Containers are isolated behind their own network compartment. This can be provided by a NAT DHCP or Static IP. Each container has an independent session namespace, which helps to provide isolation and security. The kernel object namespace is isolated per container. Windows Server Containers with Hyper-V Isolation take a slightly different approach to containerization.

To create more isolation, Windows Server Containers with Hyper-V Isolation each have their own copy of the Windows kernel and have memory assigned directly to them, a key requirement of strong isolation. Windows Server Containers and Windows Server Containers with Hyper-V Isolation are powered by Docker.

Developers and IT pros

Container-based solutions provide important benefits of cost savings. Containers are a solution to deployment problems caused by app dependencies on libraries and the OS that make transitioning the app from one environment to the next (e.g., from QA to production) so problematic. The process can be streamlined for the entire development and test cycle, ultimately reducing costs. Because containers can run on a physical or virtual host OS, you gain the flexibility to increase server consolidation.

For developers, Windows Server 2016 containers unlock huge gains in productivity. You can build an application, package the app within a container, and deploy the container, knowing that it will run without modification, on-premises, in a service provider's datacenter, or in the public cloud, using services such as Azure. You can distribute multitier apps across Infrastructure as a Service (IaaS) models and deliver apps more rapidly than ever.

At the same time, IT pros gain even higher levels of consolidation for apps and workloads. You can secure and modernize existing enterprise .NET and line-of-business server apps with little or no code changes, all on a platform that can rapidly scale to meet changing business needs.

Modernizing traditional applications

For any enterprise, the journey that you must take to modernize applications needs to be defined. Figure 2-8 outlines the entire process. Enterprises today can begin to containerize legacy applications and begin to see the benefits in terms of cost and efficiency almost immediately, it also prompts wider conversations on transforming the application portfolio for an enterprise.

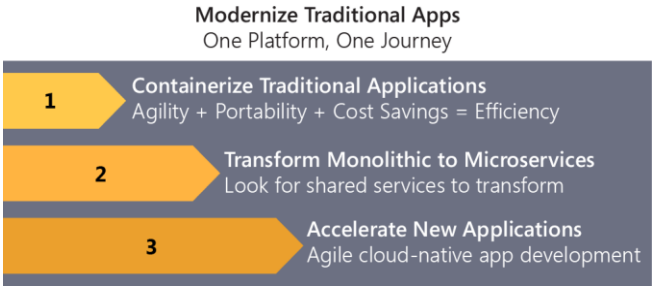

Figure 2-8: The journey for modernizing traditional applications

Deploying monolithic applications 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. Although VM Scale Sets are a great feature to scale VMs (which are required to host your Docker containers), they take time to instance. When deployed as app instances, the configuration of the app is managed as part of the VM.

Deploying updates as Docker images are far faster and more network efficient. You can instance the Vn (Docker Container) instances on the same hosts as your Vn-1 instances, eliminating additional costs of additional VMs. Docker images typically start in seconds, speeding rollouts.

Tearing down a Docker instance is as easy as using the `docker stop` command, typically completing in less than a second.

Because containers are inherently immutable, by design, you never need to worry about corrupt VMs as a result of an update script forgetting to account for some specific configuration or file left on a drive.

Even though monolithic apps can benefit from Docker, we're touching only on the tips of the benefits. The larger benefits of managing containers comes from deploying the various instances and lifecycle of each container instance.

Breaking up the monolithic application into subsystems that can be scaled, developed, and deployed individually is the entry point into the realm of microservices.

Docker commands

Docker has three primary container build functions to begin working with a container.

The `docker build` command is based on a declarative model in which a Dockerfile represents how to build a container, and the command runs on this file. The file contains at the start a `from` command that represents the base image to start from, and then it contains a series of commands that represent configuring the container and the underlying images. This model is repeatable and can provide enhanced benefits like the speed of the build using caching of previous layers that are the same.

The Docker `compose` command represents a declarative service model in which multiple containers or build files represent a service; for example, a website in one container and a data store in another, which are always required.

The most basic command is the Docker `run` command, which simply starts a container that is already built by either pulling from a central repository or locally.

A very specific trait of containers—and an important factor related to these three commands—is that they are designed to do something. "Doing something" is interesting in the Windows world, in which Windows services are not considered to be a container command. "Do something" will come up in

different models in which a container needs to start and run a command, and when the command completes the container stops. Essentially, what this means is that hosting a web application in the w3wp Windows service will not keep a container running, and you need to take special care in this instance.

In that case, you could run a custom script or custom executable file that checks that the web service is running. If that custom script or executable file finds that the service is not running, it can then log the error and stop, at which point the container stops.

Figure 2-9 presents a simplified view of some of the Docker ecosystem and how it layers together.

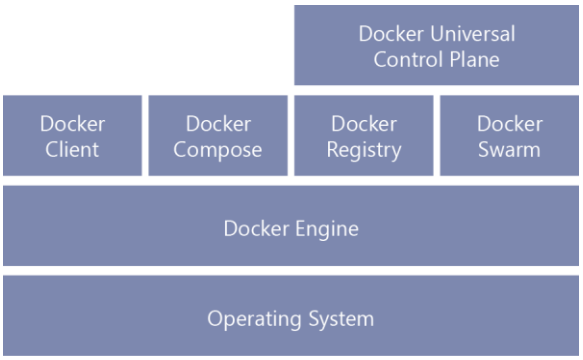

Figure 2-9: A simplified view of Docker's ecosystem

The Docker engine itself provides a REST-based management API between the Windows Server Containers or Linux Containers runtime and the management tools above it. In Windows Server Containers, the Docker engine works with the Host Compute Service in Windows, which allows it to manage containers on the host OS.

The Docker engine provides the API later to tools like the Docker client, which connects to the Docker engine to help manage the lifecycle of a container. If you need to start a container, the client will issue the `docker run` command against the Docker engine and, in turn, will contact the container runtime of the underlying OS.

What is the Docker client?

The Docker client is a command-line tool that controls the lifecycle of the containers deployed to the container host. Throughout this book, there will be plenty of examples of using the Docker client to manage containers.

The client docker.exe is located at C:\Program Files\Docker\, and when you install it, it will update the environment variables on the client machine to support calling the client from any path.

If a you run docker.exe with no additional parameters, the Docker client will issue help, describing the available parameters and switches that you can call.

Figure 2-10 depicts a Docker client showing options and commands.

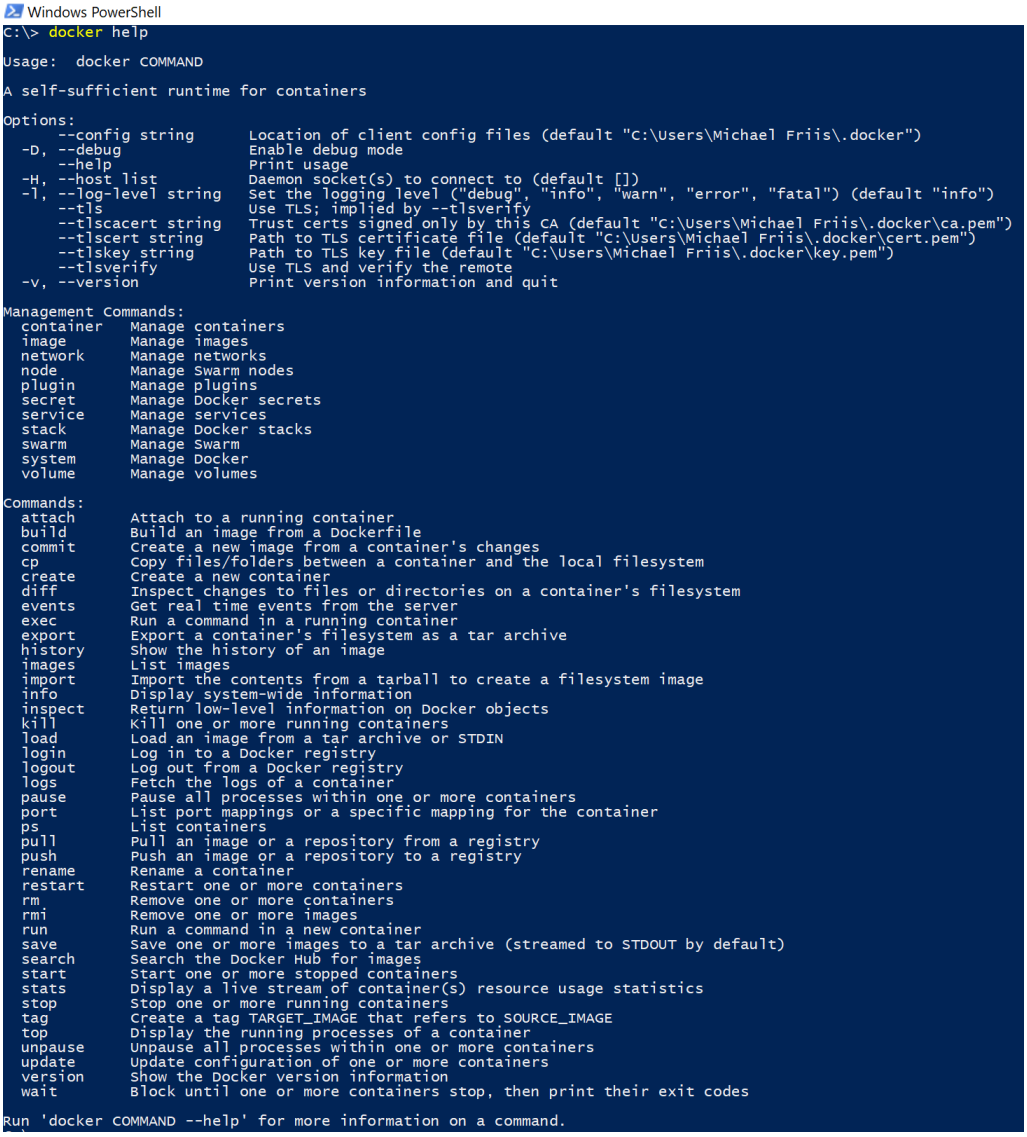

Figure 2-10: Docker client output

To obtain help on a command, you can call the `--help` parameter, as demonstrated in following code snippet:

```
docker run --help
docker image --help
```

As previously mentioned, working examples are shown through the entire book.

> **More info** To learn more about the Docker client, go to https://docs.docker.com/engine/reference/commandline/cli/.

What is a Dockerfile?

A Dockerfile is a recipe that describes how to build an application container. In this respect, it is similar to a Windows PowerShell script or an ARM template. You can use a Dockerfile to automate and simplify the packaging process for container images. Dockerfiles implement the Infrastructure as Code design pattern. You must save a Dockerfile in Windows with no file extension.

A Dockerfile looks something like the following code block:

```
# Sample Dockerfile
# Indicates that the windowsservercore image will be used as the base image.
FROM microsoft/windowsservercore
# Metadata indicating an image maintainer.
MAINTAINER joebloggs@microsoft.com
# Uses dism.exe to install the IIS role.
RUN dism.exe /online /enable-feature /all /featurename:iis-webserver /NoRestart
# Creates an HTML file and adds content to this file.
RUN echo "Hello World - Dockerfile" > c:\inetpub\wwwroot\index.html
# Sets a command or process that will run each time a container is run from the new image.
CMD [ "cmd" ]
```

In the preceding example, we use `dism` to install software; however, there are a number of different methods by which you can deploy software into a container.

Dockerfiles become important when you need to think about a patching strategy or build management in general for containerized applications.

More info To learn more about Docker files, go to https://docs.microsoft.com/virtualization/windowscontainers/manage-docker/manage-windows-dockerfile.

What is Docker Compose?

Docker Compose brings together the ability to describe multiple application containers, including all their dependencies, and treat them as a single unit of code to manage their lifecycle as if it were a single container.

You can use Docker Compose to aid mobility between development, testing, and staging environments. It also underpins workflows in Continuous Integration.

Docker Compose has three steps to building the environment. First, you define each individual container by using a Dockerfile. Then, you bring these together into a single docker-compose.yml file. Finally, you use Docker Compose to run the system.

Figure 2-11 shows the layout of a docker-compose.yml file and individual container components.

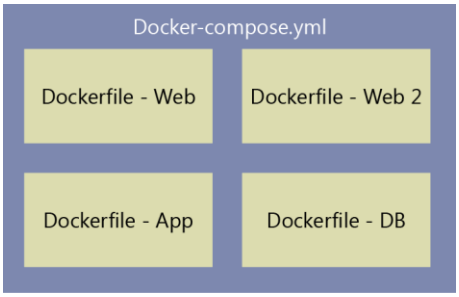

Figure 2-11: Docker Compose layout

Next, let's take a quick look at a simple docker-compose.yml file:

```
Web:
    build: .
dockerfile: web
Web2:
    build: .
dockerfile: web2
App:
    build: .
dockerfile: app
DB:
    Image: example/DB:latest
```

This calls each layer from the latest image available. We are building what could be considered the components that might be part of a build-and-release cycle. Using predefined Dockerfiles, we have defined what the images are, and then use the compose service to build it out as necessary.

More info To learn more about Docker Compose, go to https://docs.docker.com/compose/.

Getting started: modernize your apps today

Are you ready to begin containerizing traditional apps? Docker and Microsoft provide plenty of tools and best practices to help. Great choices for starter apps include Microsoft Windows Internet Information Server (IIS) websites, .Net apps, mid-tier business logic apps, and Apache Web Server. Or, you can use the Microsoft Nerd Dinner test app.

This extensive partnership between Microsoft and Docker spans the Microsoft portfolio of developer tools, operating systems, and cloud infrastructure.

Language and framework choices

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

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

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

- **Visual Studio Tools for Containers** Use a single integrated Visual Studio toolset to build, debug, and deploy apps in locally or Azure-hosted containers. Developers also gain multiproject debugging for single and multicontainer scenarios.
- **Docker for Azure** Get started building, assembling, and shipping containerized applications on Azure. The native Azure application provides an integrated, easy-to-deploy Docker environment, optimized to use the underlying Azure IaaS services.

- **Docker Datacenter in Azure Marketplace** Use prebuilt cloud templates for Docker Datacenter to develop and run containerized apps directly in the Azure cloud. Docker Datacenter delivers efficiency of computing and operations resources through Docker-supported container management and orchestration.
- **Azure Container Service** Start building, assembling, and shipping applications on Azure—no additional software installation required. This native Azure app provides an integrated, easy-to-deploy environment that uses the underlying Azure IaaS and a modern Docker platform to deploy portable apps. Standard Docker tooling and API support are included.
- **.NET Core Tools** Create a seamless experience for Windows, Linux, and Mac OS developers. Optimized for high-scale, high-performance microservices, these tools make building containerized .NET apps a breeze.
- **Image2Docker** Point this Windows PowerShell module at a virtual hard drive image, scan for common Windows components, and suggest a Docker le. The tool supports VHD, VHDK, and WIM, with a conversion tool for VMDK.

Get the latest news from Microsoft Press sent to your inbox.

- New and upcoming books
- Special offers
- Free eBooks
- How-to articles

Sign up today at MicrosoftPressStore.com/Newsletters

Microsoft

Deep dive: host deployment

In this chapter, we take a step-by-step walk-through of the different scenarios for deploying a container host within your enterprise. We also look closely at getting an initial container image ready for deployment and making it fit to serve as a foundation upon which you can build. Finally, we discuss how you can set up an existing private cloud environment to support container technology.

Deploying a container host/virtual machine (Nano, Core, Windows 10)

In this section, we explore how you can deploy a container host (physical or virtual machine) with the currently supported major container platforms.

Hardware

The hardware that an enterprise must have for deploying a container host will depend on the deployment scenario that you choose.

Figure 3-1 illustrates the various deployment scenarios that you should keep in mind when considering containers.

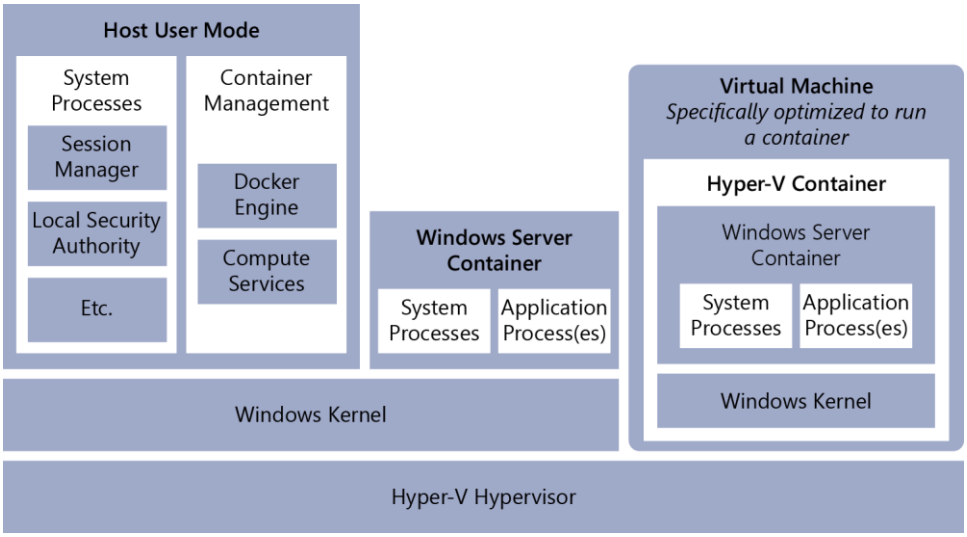

Figure 3-1: Container host deployment scenarios

For example, if an enterprise has a requirement for implementing Windows Server containers and also has applications that have security back-isolation requirements, it can choose to deploy a Windows Server 2016 with Desktop Experience host with Hyper-V turned on, set up nested virtualization on a guest virtual machine (VM) to support deployment of Windows Server Containers with Hyper-V Isolation, and deploy an additional guest VM to be a Windows Server containers host.

This is one of many different deployment scenarios that you could choose; however, each enterprise will need to look at its specific requirements before proceeding.

Choosing which deployment that an enterprise requires will determine at least some of the underlying hardware requirements. For example, if the scenario is one in which you want to run both Windows Server Containers and Windows Server Containers with Hyper-V Isolation, you will need hardware that supports virtualization extension on the processor, and possibly nested virtualization support, depending on how you deploy.

The amount of memory that a machine needs also will be determined by the scenario, although in most cases a container host might be just a VM deployed into an existing virtualization deployment, and thus its memory requirements will not be as large.

If a container host is a VM and a Hyper-V nested container host, you need to allocate at least 4 GB RAM to this machine. A container host VM also requires that you allocate at least two virtual processors to it.

Software

Container hosts are currently supported on the following versions of Windows:

- Windows 10 (Professional or Enterprise)
- Windows Server 2016 (Core, and Desktop Experience)

When Windows Server Containers with Hyper-V Isolation are a requirement, you must install the Hyper-V role on the host before trying to deploy the Windows Server Containers with Hyper-V Isolation.

When only Windows Server Containers are required, you need to install the host operating system (OS) on the C partition only. Windows Server Containers with Hyper-V Isolation do not have this requirement.

Choosing which OS to deploy and the respective version will also depend on the type of containers that are required.

Table 3-1 lists which containers are supported by which container host OS.

Table 3-1: Container software and the supported hosts

Host OS	Windows Server Container	Windows Server Containers with Hyper-V Isolation
Windows Server 2016 with Desktop	Server Core/Nano Server	Server Core/Nano Server
Windows Server 2016 Core	Server Core/Nano Server	Server Core/Nano Server
Windows 10 Pro/Enterprise	Not available	Server Core/Nano Server

Notice, for example, that with a Windows 10 host OS you cannot support Windows Server Containers, or with a Nano Server host without Hyper-V, you can support only a Windows Server Container of type Nano. However, if you have Windows Server Containers with Hyper-V Isolation, you can implement any type of kernel because the operating system is isolated.

> **Note** Except for Nano Server, a traditional windows installation should occur. for more information on installing Windows Server 2016, go to https://technet.microsoft.com/windows-server-docs/get-started/getting-started-with-server-with-desktop-experience.

Deploying a Windows Server 2016 Container host with Desktop Experience

You can deploy a Windows Server 2016 container host with Desktop Experience either as a physical machine or a VM. Either deployment requires that the Windows OS is installed on the C partition. This assumes that the host is not being used for Windows Server Containers with Hyper-V Isolation.

You should perform a default deployment of Windows as a starting point. Next, set up a Windows Server 2016 with Desktop Experience for use as a container host. To do this, you need to install the container feature.

Figure 3-2 demonstrates how you can use the Add Roles and Features Wizard to turn on the container functionality. However, because containers in Windows Server will be managed via Docker, it is recommended that you use the OneGet Windows PowerShell engine. This will turn on the container feature and install the Docker engine and client. For this to work, the host will require Internet access.

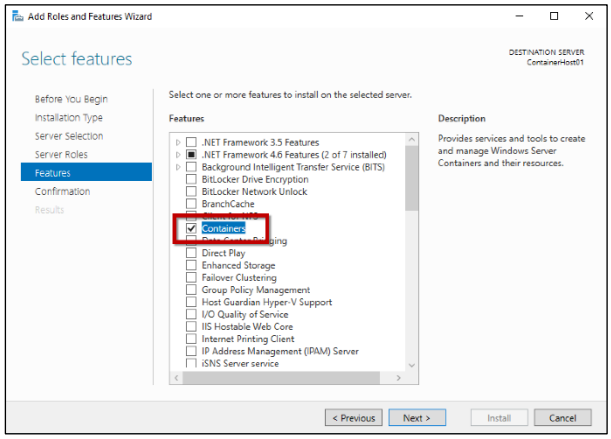

Figure 3-2: Installing the container host feature

From an elevated Windows PowerShell prompt, you can use the following command:

```
Install-Module -Name DockerMSFTProvider -Repository PSGallery
```

If this is a fresh host, you will see a prompt to obtain the NuGet provider, as shown in Figure 3-3.

```
PS C:\Users\Administrator> install-module -Name DockerMSFTProvider  -Repository PSGallery

NuGet provider is required to continue
PowerShellGet requires NuGet provider version '2.8.5.201' or newer to interact with NuGet-based repositories. The NuGet
 provider must be available in 'C:\Program Files\PackageManagement\ProviderAssemblies' or
'C:\Users\Administrator\AppData\Local\PackageManagement\ProviderAssemblies'. You can also install the NuGet provider by
 running 'Install-PackageProvider -Name NuGet -MinimumVersion 2.8.5.201 -Force'. Do you want PowerShellGet to install
and import the NuGet provider now?
[Y] Yes  [N] No  [S] Suspend  [?] Help (default is "Y"):
```

Figure 3-3: Prompting for a NuGet provider

If you want the installation to proceed, confirm by pressing Y. Additionally, if this is an untrusted repository and has not been previously configured, confirm the installation from the untrusted repository. This will download the provider and turn on the containers feature.

Next, install the Docker engine and client by using the following command:

```
Install-Package -Name docker -ProviderName DockerMsftProvider
```

If you have not preconfigured trust for the repository, you will be required to confirm the Docker installation.

This installation step will require you to restart the computer, which you can do by using the following command:

```
Restart-Computer
```

After you have restarted, sign in to the server and perform a full Windows Update to ensure that all components are at the latest version.

Deploying a Windows Server 2016 Core container host

Deploying containers on a Windows Server 2016 Core host follows the same deployment mechanisms as a Windows Server 2016 with Desktop Experience. When you initially sign in to a core host, you need to run "PowerShell" from the command window before running the same sequence.

However, updating Windows Server 2016 Core is slightly different than a traditional host. Windows Server 2016 Core uses the Server Configuration app to update a host. Run the following command to invoke the Server Configuration app:

```
sconfig
```

The Server Configuration app has several options to update a host; choose option 6, as shown in Figure 3-4.

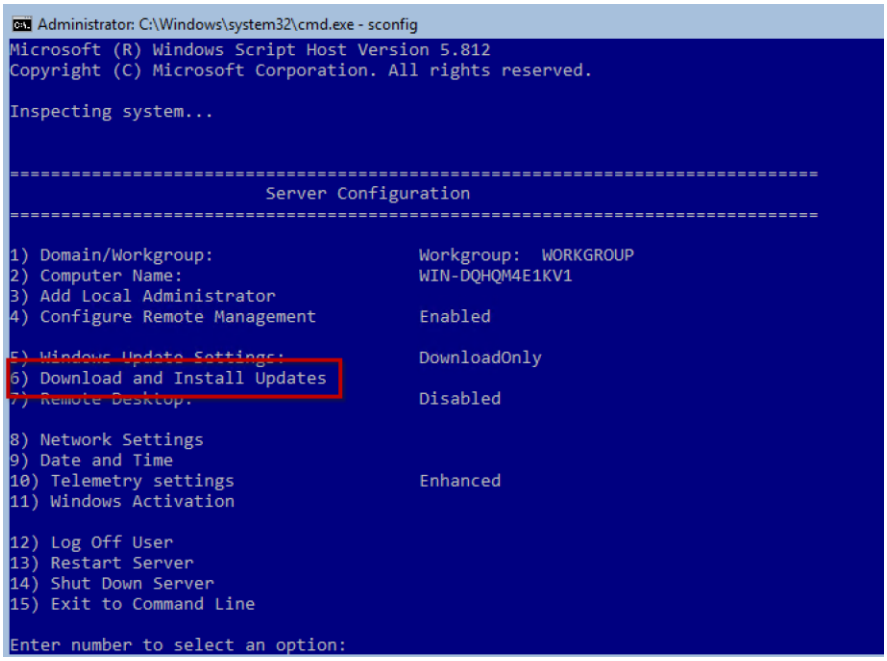

Figure 3-4: The Server Configuration app

Choosing option 6 opens a script host window asking if all updates or just the recommended updates are required, as depicted in Figure 3-5. In this case, choose All.

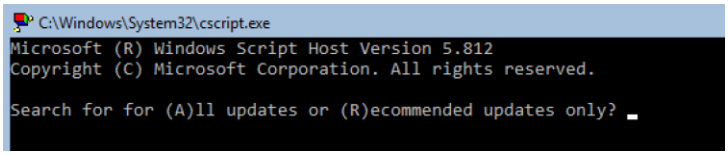

Figure 3-5: Choosing which updates to install on a Windows Server 2016 Core

The system will perform an update and restart, after which it will be ready to use as a container host.

Deploying a Windows 10 container host

As previously mentioned, Windows 10 does not support Windows Server Containers; it supports only Windows Server Containers with Hyper-V Isolation. This requires the underlying hardware to support virtualization extensions so that Hyper-V can be turned on within Windows 10. In addition, the build number must be at least 14393.222. Windows 10 also uses Docker to manage the container engine; however, its installation process is different from that of Windows Server.

Downloading and Installing the Docker Engine will not prepare the Windows 10 host with the required features.

To manually install the necessary features for Windows 10 to be a container host, perform the following:

1. From an elevated Windows PowerShell prompt, run the following command to turn on containers in Windows 10:

   ```
   Enable-WindowsOptionalFeature -Online -FeatureName Container -All
   ```

2. Turn on Hyper-V by using the following command:

   ```
   Enable-WindowsOptionalFeautre -Online -FeatureName Microsoft-Hyper-V -All
   ```

3. After you have turned on the Hyper-V feature, it is recommended that you restart the computer before continuing. Use the following command to do this:

   ```
   Restart-Computer
   ```

   You can download the Docker engine from https://download.docker.com/win/stable/InstallDocker.msi.

4. Run the installer, accepting the defaults to deploy the Docker engine and client.

> **Note** Windows 10 can be a VM; however, the host on which that VM will run requires the ability to turn on nested virtualization. If this is not possible, you cannot set up Windows 10 as a container host on a VM.

Deploying a Nano Server container host

> **IMPORTANT NOTICE** This section is provided as a legacy reference. This will work only on PRE-RS3 builds of Nano Server. For RS3 builds and beyond, this will not be possible to achieve.

You can get up and running with Nano Server in a variety of ways. The simplest way to deploy Nano Server with the packages required to run a container host and Hyper-V is to use the Windows PowerShell module NanoServerImageGenerator (which is located on the Windows Server 2016 installation media, in the folder NanoServer) to create a VHD file with the necessary roles and services installed.

> **More info** To learn more about creating Nano Server images, go to https://technet.microsoft.com/windows-server-docs/get-started/deploy-nano-server#a-namebkmkonlineainstalling-roles-and-features-online.

For example, the following command will create a new VHD with Nano Server with the Hyper-V and containers packages installed:

```
New-NanoServerImage -Compute -Containers -Mediapath <rootpathtonanoserver> -basepath
<pathtonanoserverfolder> -targetpath <outputpathforvhd> -DeploymentType <Guest/Host> -Edition
<Datacenter/Standard> -AdministratorPassword <password>
```

After you create the VHD, you can create a VM and attach this new VHD to the VM and start the Nano Server. Nano Server itself has no UI, leaving everything that is required to be configured from here via remote Windows PowerShell.

> **Note** VHD is not supported on Generation 2 VMs. If you are deploying Generation 2 VMs, select VHDX for the file extension.

To obtain the IP address of the Nano Server, select Networking, as shown in Figure 3-6.

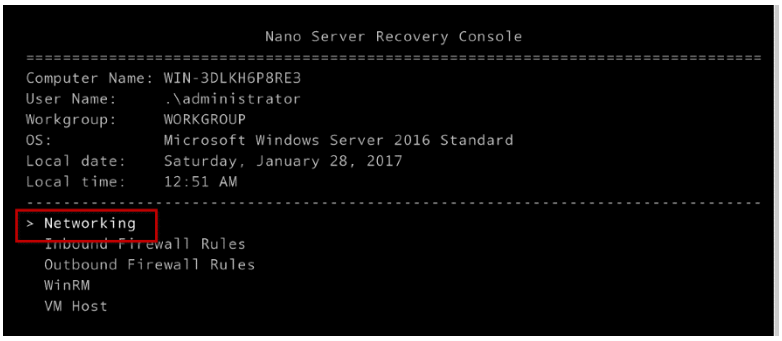

Figure 3-6: Selecting Networking on Nano Server

Next, select the network adapter and record the IP address, as illustrated in Figure 3-7. This will be required to continue the installation via a remote Windows PowerShell session.

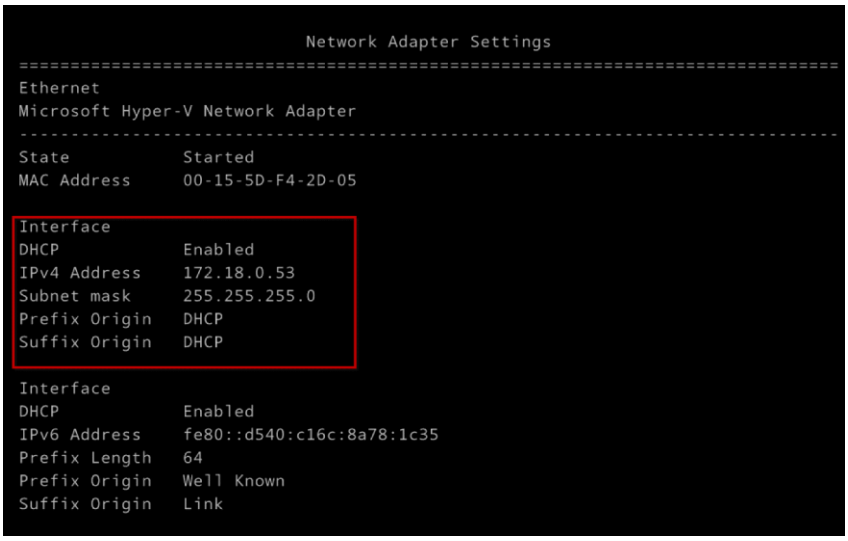

Figure 3-7: Identifying the network configuration

Next, you need to finish the installation of the Docker components on the Nano Server via remote Windows PowerShell. Using the following command, first add the Nano Server IP address to the WSMan trusted hosts:

```
Set-Item WSMan:\localhost\Client\TrustedHosts 172.18.0.53 -Force
```

From here, establish a remote Windows PowerShell session to the Nano Server by using this command:

```
Enter-PSSession -Computer 172.18.0.53 -Credential (Get-Credential)
```

The `(get-credential)` item will prompt for credentials before connecting, in case it is different than the corporate domain credentials. Figure 3-8 shows the session, the steps, and the successful result of connection to a remote Windows PowerShell session.

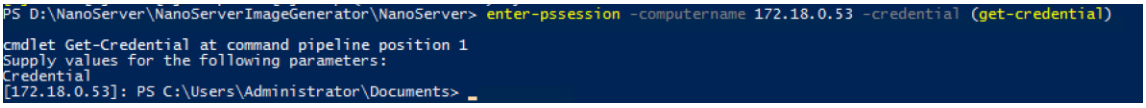

Figure 3-8: Connecting to a remote Windows PowerShell session

Next, you need to carry out a Windows Update on the Nano Server image to ensure that all critical updates are installed. This is a required step to be sure that Nano Server can function properly as a container host.

Using the following command, create a CIM session to the Windows Update API:

```
$updateSession = New-CimInstance -Namespace root/Microsoft/Windows/WindowsUpdate
-ClassName MSFT_WUOperationsSession
```

Now, invoke the update method by using this command:

```
Invoke-CimMethod -InputObject $updateSession -MethodName ApplyApplicableUpdates
```

After the update is complete, restart the computer:

```
Restart-Computer
```

The remaining Docker installation follows the same steps as the other editions of Windows. For the condensed steps, use the following code:

```
Install-Module -Name DockerMSFTProvider -Repository PSGallery -Force
Install-Package -Name docker -ProviderName DockerMsftProvider -Force
Restart-Computer -Force
```

The `-Force` parameter in this example avoids having to acknowledge each part.

When the reboot is finished, you have an operational Nano Server container host—with one exception!

Docker can run two types of sessions for containers: Interactive and Detached. In Nano Server, because it is managed remotely, an interactive container will not work as expected, because it is not able to redirect its terminal output to Windows PowerShell Remote Session. However, you can invoke a detach container, which will run successfully in the background.

To launch a detached container, use the following command:

```
docker run -dt microsoft/nanoserver
```

This invokes a nanoserver container and runs it in the background. If you want to have an interactive container, some additional work is required on the container host and the remote client used for management.

On the container host, you need a new firewall rule to support this. Use the following command from a remote Windows PowerShell session to the Nano Server container host:

```
Netsh advfirewall firewall add rule name="Docker" dir=in action=allow protocol=TCP localport=2375
```

The `localport 2375` is used to make a remote nonsecured connection for the Docker client to the daemon; if you require a secured connection, use `localport 2376`.

> **More info** To read more about creating secure connections and the daemon.json configuration file, go to https://docs.docker.com/engine/security/https/.

Next, you need to create a JSON configuration file (daemon.json), which will instruct the Docker daemon to accept incoming connections on the described port. The following command creates the configuration file in the correct location to be interpreted by the Docker daemon on the next restart:

```
New-Item -Type File C:\ProgramData\docker\config\daemon.json
```

Now, you need to add some configuration lines to the file, the structure of what you need to add is as follows:

```
{
   "hosts" : ["tcp://0.0.0.0:2375","npipe://"]
}
```

Using the `Add-Content` command, inject the configuration into the daemon.json file, as follows:

```
Add-Content "C:\ProgramData\docker\config\daemon.json" '{
"hosts":["tcp://0.0.0.0:2375","npipe://"]}'
```

Finally, restart the Docker service by using this command:

```
Restart-service docker
```

On the client where management of the remote container host will occur, you can use the `-H` parameter to instruct Docker to make a connection to the container host on a specific port. Using the following command, you can invoke a remote interactive container and observe the output:

```
docker -H "tcp://172.18.0.53:2375" run -it microsoft/dotnet-samples:dotnetapp-nanoserver
```

Setting up a Windows Host for Windows Server Containers with Hyper-V Isolation support

Turning on Windows Server Containers with Hyper-V Isolation support for Windows Server 2016 Core and Windows Server 2016 with Desktop Experience is a very simple matter of installing the Hyper-V role. You do this by using the following command:

```
Install-WindowsFeature -Name Hyper-V
```

Ensure that you restart the computer after installing the Hyper-V role. Like Windows 10, if you want to run a Windows Server 2016 Core or a Windows Server 2016 with Desktop Experience as a container host within a VM, the underlying hardware platform must support nested virtualization.

To verify whether the processor(s) supports nested virtualization, run this command:

```
get-wmiobject Win32_Processor |Select SecondLevelAddressTranslationExtensions
```

If the output is `True`, the processor(s) supports nested virtualization, if `False`, the processor does not support nested virtualization.

If the hardware supports nested virtualization, you need to run an additional command against a VM while it is off to set up the extensions into the VM. To do that, use this command:

```
Set-VMProcessor -VMName <vmname> -ExposeVirtualizationExtensions $true
```

Finally, you need to turn on MAC spoofing on the nested VM to allow the "child containers" network access:

```
Get-VMNetworkAdapter -VMName <vmname> | Set-VMNetworkAdapter -MacAddressSpoofing On
```

Deploying a Windows Server 2016 container host in Microsoft Azure

Technically, you could use all of the previously outlined methods to create a VM for any of the desired types of operating systems and upload its VHD (not VHDX; if you have a VHDX, you will need to convert it) to Azure or build a base Windows Server 2016 in Azure and turn on the container components.

In Azure, Microsoft has provided a prebuilt marketplace image with the container functionality turned on. As of this writing, this image currently supports only Windows Server Containers; Azure neither supports nested virtualization nor provides access to the host hypervisor in order to turn on Windows Server Containers with Hyper-V Isolation.

This is not really an issue for getting started with containers today, because developing for a Windows Server Containers with Hyper-V Isolation or a Windows Server Container is essentially the same. However, if for compliance purposes a container image requires the isolation that a hypervisor provides, Azure currently does not support that.

As previously mentioned, Microsoft provides an image for Windows Server 2016 that has container support built in. You can find this image in the Azure Marketplace, as illustrated in Figure 3-9.

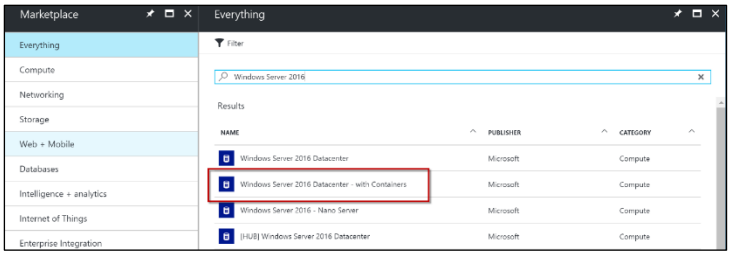

Figure 3-9: Windows Server 2016 with Container image in the Azure Marketplace

Clicking the image and then clicking Create prompts you to navigate the input pages to provide information to create the service. On the first page, Basics (see Figure 3-10), users are asked for basic information.

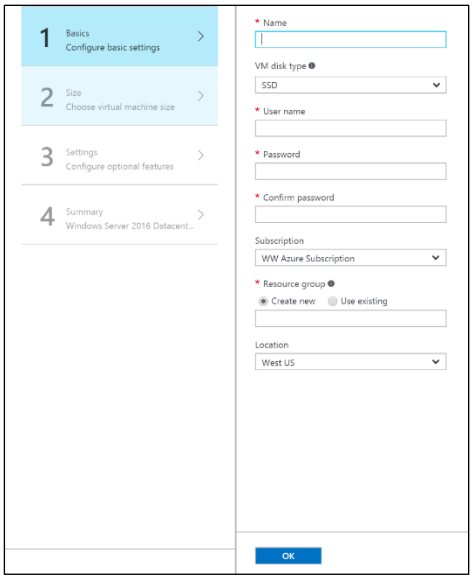

Figure 3-10: Creating a VM basic inputs

Provide the requested information to all of the required fields (highlighted with a red asterisk). You are required to choose the type of VM Disk Type; the choice is between SSD and HDD. Choosing SSD will enforce the use of Premium Azure Storage, whereas choosing HDD will enforce the use of Standard Azure Storage.

More info To learn more about Azure Storage, go to https://docs.microsoft.com/azure/storage/storage-introductionhttps://docs.microsoft.com/azure/storage/storage-introduction.

Keep in mind that users cannot choose the protected names of "root" or "Administrator" for the username of the VM.

When all fields are populated, click OK.

Next, choose a VM size; in this example, choose D1_V2, and then click select.

After choosing the VM size, you need to select the remaining options, as shown in Figure 3-11. By default, all fields will be populated. Some of the fields will be filled-in with resources that already exist in the Azure subscription; however, other fields will attempt to create new resources. For example, the Azure Storage Account will always create a new storage account.

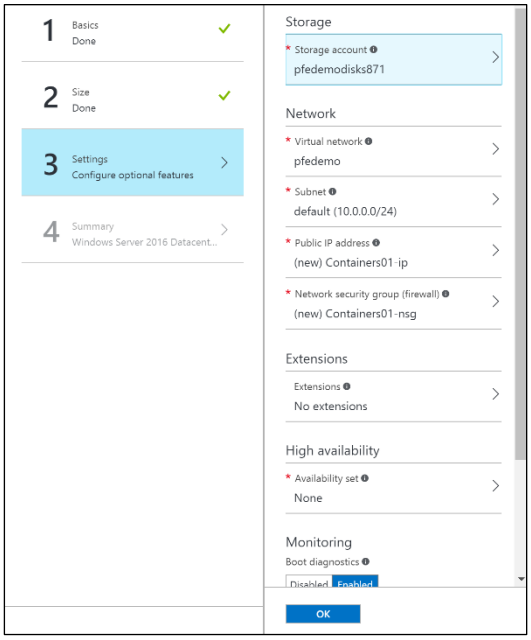

Figure 3-11: Providing a VM's settings

When you've finished configuring all of the options, click OK. The Summary page opens, on which you can confirm the settings. Click OK to deploy the VM with container support.

The build process for the VM takes between 5 and 15 minutes and will become accessible at its private or public address (in this example, we assigned a public IP) via the remote desktop protocol.

The image comes with the Docker tooling and containers preinstalled, so no additional work is required unless a you want to join the domain or install additional software to the container host.

Note You also can utilize Azure Resource Manager templates to deploy your own Windows Server Image or a Base Image and Enable Container Support. Use the examples at https://github.com/Azure/azure-quickstart-templates/ to build a custom template for an enterprise.

Deploying a base container image

When you deploy a container host, initially it has no images. To begin building the layers for an enterprise's application container, you need to deploy base images to the container host.

As of this writing, Microsoft supplies two base images:

- Server Core
- Nano Server

It is possible to download both images to the container host. However, revisit the supportability chart in Table 3-1 to determine which container image can run on which container host. If you inadvertently download an image that is not supported, it simply will not start on the container host.

Using the Docker client, you can invoke a download of the container image from the public repository to the local container host.

To download a Nano Server image, use the following command:

```
docker pull microsoft/iis:nanoserver
```

To download the Server Core image, use this command:

```
docker pull microsoft/windowsservercore
```

After the download is complete, run the following command to verify that the images are downloaded and registered correctly:

```
docker images
```

Figure 3-12 displays the installed images.

```
PS C:\Users\msadmin> docker images
REPOSITORY                      TAG                 IMAGE ID            CREATED             SIZE
microsoft/windowsservercore     latest              4d83c32ad497        2 weeks ago         9.56 GB
microsoft/nanoserver            latest              d9bccb9d4cac        2 weeks ago         925 MB
```

Figure 3-12: Listing the installed images on a container host

Running a sample container

Although you can run the images you downloaded, there is a sample container located in the registry that you can download. When the container is invoked, it will generate an output and close the container. This is a simple way to verify that the container host that you just deployed is operational.

Using the following command, download and run the .NET sample app from the image repository:

```
docker run microsoft/dotnet-samples:dotnetapp-nanoserver
```

This command checks the local repository on the container host. If it detects that the container image is not listed, it will proceed to download the container image and all support layers required to the local container host, as shown in Figure 3-13.

```
PS C:\Users\Administrator> docker run microsoft/dotnet-samples:dotnetapp-nanoserver
Unable to find image 'microsoft/dotnet-samples:dotnetapp-nanoserver' locally
dotnetapp-nanoserver: Pulling from microsoft/dotnet-samples
bce2fbc256ea: Already exists
3ac17e2e6106: Already exists
82d2e10a30fc: Pull complete
4ba860d23251: Pull complete
3ee3c07c71cc: Pull complete
4e10778f40d0: Pull complete
15572b221755: Pull complete
cf4afa5c6989: Pull complete
15a8c52099b5: Pull complete
b583d41d7e4a: Pull complete
Digest: sha256:6227ee7d0b524da91f6f53f6765e2fc964d1a73e27f231e266bf85fb2f9e9618
Status: Downloaded newer image for microsoft/dotnet-samples:dotnetapp-nanoserver
```

Figure 3-13: .NET sample container

Figure 3-14 shows the output after running the container.

Figure 3-14: .NET sample container output

The container image is also installed in the local image repository. You can view it by using the previously shown `docker images` command.

CHAPTER 4

Deep dive: working with containers

In this demonstration-driven chapter, we dive deeper into some of the concepts explored in the previous chapters, including managing containers, automating and orchestrating container deployments, and examining how we can set up existing private cloud deployments to support container deployments.

Docker client cheat sheet

Before diving into this chapter, let's separate the available commands in the Docker client into categories that will help you to locate the command you want, specifically for the action you want.

For each command, you can use the `docker <command> --help` syntax to get detailed information on available commands and how to use them.

Lifecycle

Table 4-1 outlines the commands available for managing the lifecycle of a container.

Table 4-1: Lifecycle commands for the Docker client

Command	Description
`docker create`	Creates a container
`docker rename`	Renames a container
`docker run`	Creates and runs a container
`Docker rm`	Removes a container
`docker update`	Updates a container resource limits

Starting and stopping a container

Table 4-2 lists the commands available for starting and stopping a container.

Table 4-2: Start and stop commands for the Docker client

Command	Description
`docker start`	Starts a container
`docker stop`	Stops a container
`docker restart`	Stops and starts a container
`docker pause`	Pauses a running container
`docker unpause`	Unpause a running container
`docker wait`	Blocks until running container stops
`docker kill`	Sends a SIGKILL to a container
`docker attach`	Connects to a running container

Container resource constraints

Table 4-3 presents the commands available to limit resources.

Table 4-3: Container resource constraints commands

Command	Description
`docker run --ti --c 512 <containername>`	Sets the container to 50% usage of the available CPU cores The value 512 specifies 50%, whereas changing the value to 1024 specifies 100%
`docker run –ti -cpuset-cpus=0,1,2`	Sets the container to use a specific number of cores
`docker run -it -m 300M <container>`	Sets the container to have a memory limit

Container information

Table 4-4 outlines the commands available to show information around and in a container.

Table 4-4: Container information commands

Command	Description
`docker ps`	Shows the running containers
`docker logs`	Gets logs from a container
`docker inspect`	Looks at all the information on a container
`docker events`	Gets event information from a container
`docker port`	Shows the public-facing port of a container
`docker top`	Shows running processes in a container
`docker stats`	Shows the resource usage statistics for a container
`docker diff`	Shows changed files in the containers file systems

Images

Table 4-5 lists the commands available for image management.

Table 4-5: Container image commands

Command	Description
`docker images`	Shows all the images on the container host
`docker build`	Create an image from a Dockerfile
`docker commit`	Creates an image from a container
`docker rmi`	Remove an image from a container host
`docker history`	Shows all the history of image
`docker tag`	Tags an image to a local host or registry
`docker search`	Search the Docker Hub for an image

Network

Table 4-6 shows the commands available for networks.

Table 4-6: Network commands

Command	Description
`docker network create`	Creates a network for a container
`docker network rm`	Removes a network
`docker network ls`	Lists all networks
`docker network inspect`	Display all info in relation to the network
`docker network connect`	Connects a container to a network
`docker network disconnect`	Disconnects a container from a network

Managing container deployments

With our examination of the most common commands available in the Docker client complete, we now dive into some practical examples that use these commands to build on top of the basic examples of pulling an image from a repository and running a container that we demonstrated in Chapter 3.

Listing installed images

When you first install a container host, no images are installed by default except in the case of the Microsoft Azure Marketplace image. The Azure Marketplace image installs both the Windows Server Core Image and the NanoServer image.

To list all of the images installed on a container host, use the following command:

```
docker image
```

Figure 4-1 shows the output of the Docker `image` command and the images installed.

Figure 4-1: Container images output

The output includes essential information about the container, which can also help you to understand the dependency layers of other containers, as well.

Searching for an image from a repository

As just mentioned, images are not installed by default, thus you need to download them from an image repository. By default, the `docker search` command looks at Docker Hub for the stored container images and download from there.

To identify a container image that you want to use, type the following command:

```
docker search <keyword>
```

Figure 4-2 depicts the output of a search for NanoServer.

```
PS C:\Users\Administrator> docker search nanoserver
NAME                                DESCRIPTION                                     STARS     OFFICIAL   AUTOMATED
microsoft/nanoserver                Windows Server 2016 Nano Server base OS im...   115
nanoserver/iis                      Nano Server + IIS. Update on january 28, 2...   8
nanoserver/iis-mysql-php-wordpress  Nano Server Container + IIS + MySQL + PHP ...   3
adikurniawan94/nanoserver                                                           0
lgarcia4593/nanoserver-iis          Windows 2016 NanoServer running IIS             0
tirexx/nanoserver                   Nano Server base OS image for Windows cont...   0
jovanstojanov/nanoserver                                                            0
radarvector/nanoserver                                                              0
egitmen/nanoserver                                                                  0
codedbeard/nanoserver-redis         Redis running on microsoft/nanoserver           0
laoshancun/nanoserver                                                               0
sixeyed/nanoserver                                                                  0
brycem/nanoserver                                                                   0
djd4352/nanoserver                                                                  0
shichiyou/hello-world-nanoserver    hello world nanoserver version.                 0
gavineke/nodejs-nanoserver                                                          0
sixeyed/nanoserver-helloworld       Basic sample container using Windows Nano ...   0
stefanscherer/nanoserverapiscan     NanoServerApiScan tool as a Docker image        0
zyzzvn/nanoserver                                                                   0
nanoserver/wamp                     Nano Server + Apache + MySQL + PHP. WAMP S...   0
instrutordocker/nanoserver-dexter-wamp                                              0
nanoserver/apache24                 Nano Server + Apache 2.4. Updated on 07 Ja...   0
myfinder/aspnet-docker-nanoserver                                                   0
nanoserver/java                                                                     0
nanoserver/dotnetcore                                                               0
PS C:\Users\Administrator> _
```

Figure 4-2: Search output

The key part is to identify the name. Some examples you can experiment with are Windows, WindowsServer, Nano, and SQL; these will give you different images that have been stored in the repository, and then we can use the Docker `pull` command to download the correct image to the repository.

Pulling images from a repository

After you have identified which image you require, you can then use the following command to download it from the repository:

```
docker pull <imagename>
```

In most cases, the image will have dependencies, the Docker client will determine if the dependencies are in place on the local container host and download them, as well, if necessary.

Figure 4-3 shows a sample output for "pulling" the microsoft/iis:nanoserver image from the repository.

```
PS C:\Users\msadmin> docker pull nanoserver/iis
Using default tag: latest
latest: Pulling from nanoserver/iis
bce2fbc256ea: Already exists
3ac17e2e6106: Already exists
2b8149153c0f: Pull complete
Digest: sha256:0096df968a33cb4c5cdce935101ec5890b44ad177046b2f16976bf7fe0b8814b
Status: Downloaded newer image for nanoserver/iis:latest
```

Figure 4-3: Pulling an image from the Docker registry

When you download new images, run the Docker `image` command as previously shown to view the installed images.

Starting and stopping containers

Now that you have some container images downloaded, you need to start at least one of them so that you can begin customizing it later and creating your own images for use.

The first task you need to do on any host is to run a Docker image. I say this very specifically because, technically, there are no containers yet. This will help you to choose the command that you want to use.

Docker client offers three commands for starting and stopping containers. The first, `docker run`, makes it possible for you to start an image and create a container runtime so that you can use it and

interact with it. the second command, `docker start`, starts a container that you have stored or created from an image. Finally, `docker stop`, as its name implies, stops the running container.

When you use the `docker run` command, you need to consider *how* you want the container to run. For example, do you want it to run in the background and do its job, or do you need to interact with it? This will determine the initial runtime option you select; for example, `-detach` or `-d` for detached mode, or `-interactive` or `-i` for interactive mode.

> **Note** Be careful when you start a container in interactive mode because the process you are starting within the container needs to support this!

Thus, to start a container in detached mode from a base image, the `docker run` command as follows:

```
docker run -d <imagename>
```

If you replace <*imagename*> with the name of the image listed in Figure 4-3, it will invoke a container based on the microsoft/iis:nanoserver image.

You can verify that it is running by using the `docker ps` command, as shown in Figure 4-4.

```
PS C:\Users\msadmin> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
       NAMES
ed756d464e85        microsoft/iis       "C:\\ServiceMonitor..."  About a minute ago  Up 59 seconds       80/tcp
       happy_pike
c974ba768f75        microsoft/iis       "C:\\ServiceMonitor..."  2 minutes ago       Up 2 minutes        80/tcp
       fervent_haibt
95ade33e3ea5        microsoft/iis       "C:\\ServiceMonitor..."  6 minutes ago       Up 6 minutes        80/tcp
       vigilant_haibt
2ce34cc8f5ce        microsoft/iis       "C:\\ServiceMonitor..."  11 minutes ago      Up 6 minutes        80/tcp
       nostalgic_bhabha
PS C:\Users\msadmin>
```

Figure 4-4: Showing running containers

You can use the container ID shown in Figure 4-4 to stop a running container. At this point, we can't stop a container by its name, because we have not committed any changes and created a custom container yet.

To stop a running container by its container ID, use the following syntax:

```
docker stop <containerID>
```

This stops the container, which you can verify by using the `docker ps -a` command, which lists all of the containers and their states. Figure 4-5 depicts the output showing stopped containers.

```
PS C:\Users\msadmin> docker ps -a
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
        PORTS               NAMES
ed756d464e85        microsoft/iis       "C:\\ServiceMonitor..."  12 minutes ago      Exited (1067) 2 seconds
 ago                        happy_pike
c974ba768f75        microsoft/iis       "C:\\ServiceMonitor..."  13 minutes ago      Up 13 minutes
        80/tcp              fervent_haibt
95ade33e3ea5        microsoft/iis       "C:\\ServiceMonitor..."  18 minutes ago      Up 18 minutes
        80/tcp              vigilant_haibt
2ce34cc8f5ce        microsoft/iis       "C:\\ServiceMonitor..."  22 minutes ago      Up 17 minutes
        80/tcp              nostalgic_bhabha
```

Figure 4-5: Showing all containers

You can start the same container by using the `docker start` command, as follows:

```
docker start <containerID>
```

As of this writing, the Docker client offers a pause and un-pause option, which currently are not working for Windows Server Containers.

Finally, if you need to completely stop a container forcibly and the `docker stop` command is not working, you can utilize the `docker kill` command to instantly stop the container from running. Here's the code to do that:

```
docker kill <containerID>
```

Running commands within a container

With a base container up and running on your container host, you will likely want to perform customizations or carry out actions within the container, such as install a windows role. There are two main options from which you can choose to do this: `docker attach` or `docker exec`.

If you want to view the status of the container, use `docker attach`, as follows:

```
docker attach <containerID>
```

Using this method, you also can attach to a container by invoking it using something like `cmd` or `powershell`, which gives you the ability to interact with the container. If you don't want to interact with the container, you must instead use the `docker exec` command, as follows:

```
docker exec <containerID> powershell
```

This opens a Windows PowerShell session directly within the container, which you then can customize or interact with it.

Committing changes to an image

As we have previously mentioned, containers are immutable and stateless. This essentially means that when you pull a container image from a repository and start it, it will run from the stored build on the repository. You can then deploy changes to the container, but because of the containers nature, if you do not commit those changes, as soon as the container restarts, you will lose all of your changes.

To avoid losing changes that you want to keep, use the `docker commit` command, as follows:

```
docker commit <containerID> <newcustomimagename>
```

> **Note** The <*newcustomimagename*> property in the preceding snippet must be lowercase.

Figure 4-6 shows committing a container image, and then that image appearing in the image store for use.

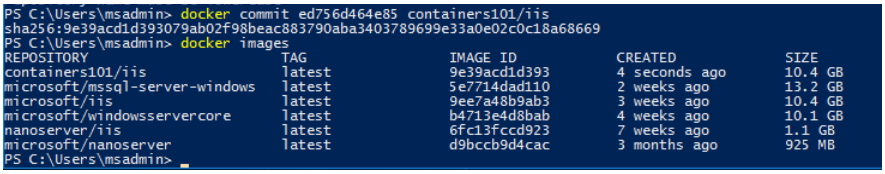

Figure 4-6: Committing a container and displaying the repository

When you gain more experience with containers, you will use Dockerfiles to build out your container and commit from there versus manually performing the changes and then committing them.

Deleting containers

If you need to delete a container image, use the `docker rmi` command, as follows:

```
docker rmi <containername>
```

If you have a container started or previously created from that base image, you might need to include the `-f` or `-force` option to remove the image from the repository.

Container resources restrictions

If you need to restrict resources for a container, you can so across three items: CPU, memory, and storage.

CPU

To restrict the CPU when running a container image, use the `-cpus` parameter. Here's a simple example that shows a container being restricted to 90 percent of the host CPU capacity:

```
docker run --cpus=".9" Microsoft/iis
```

To verify this, you can use the `docker inspect` command, as follows:

```
docker inspect <containerID>
```

The output shows a complete list of configuration items from the container, as depicted in Figure 4-7, but in this case, our focus should be on the `NanoCpus` property in the JSON output.

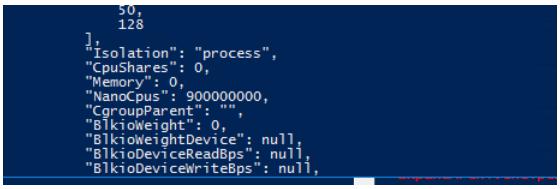

Figure 4-7: Highlighting the CPU resource restriction

Memory

To restrict the memory of a container, use the `-memory` or `-m` parameter with the `docker run` command, as shown here:

```
docker run -m="50" microsoft/iis
```

In this example, we're limiting memory to 50 MB, which, again, you can verify by using the `docker inspect` command. In the corresponding output, the memory property appears just above the NanoCpus, as shown in Figure 4-7.

Storage

To restrict the storage a container, use the `--storage-opt` parameter with the subset option of size to restrict the containers file system.

```
docker run --storage-opt size=70G microsoft/iis
```

Run the `docker inspect` command to verify the restriction (look for the StorageOpt property).

Understanding container operations

In the previous sections, we introduced two commands, `docker ps` and `docker inspect`, which show you the running and stopped containers on your host as well as their assigned configuration.

Host information

To review information on the container host, you can use the `docker info` command, as shown in the following snippet:

```
docker info
```

Figure 4-8 illustrates the output, which contains detailed information about the host.

```
PS C:\Users\msadmin> docker info
Containers: 30
 Running: 6
 Paused: 0
 Stopped: 24
Images: 6
Server Version: 17.03.0-ce
Storage Driver: windowsfilter
 Windows:
Logging Driver: json-file
Plugins:
 Volume: local
 Network: l2bridge l2tunnel nat null overlay transparent
Swarm: inactive
Default Isolation: process
Kernel Version: 10.0 14393 (14393.953.amd64fre.rs1_release_inmarket.170303-1614)
Operating System: Windows Server 2016 Datacenter
OSType: windows
Architecture: x86_64
CPUs: 1
Total Memory: 3.5 GiB
Name: Containers01
ID: H7D3:4N4S:SP7U:FUR4:PJ6B:ONQY:7LJU:WPI4:W7TQ:2J5B:ACT2:27YW
Docker Root Dir: C:\ProgramData\docker
Debug Mode (client): false
Debug Mode (server): false
Username: johnmccabe2014
Registry: https://index.docker.io/v1/
Experimental: false
Insecure Registries:
 127.0.0.0/8
Live Restore Enabled: false
PS C:\Users\msadmin>
```

Figure 4-8: The output from running the `docker info` command

Viewing container information

There might be times when you need to understand a bit more about what the container is consuming at that instant from an outside perspective. To view this information, use the `docker stats` command with the `-a` option, as demonstrated here:

```
docker stats -a
```

This command provides an insight into exactly how the container is consuming CPU, memory, and network, as shown in Figure 4-9. Alternatively, we can show only the running containers by omitting the `-a` option, as shown here:

```
docker stats
```

```
CONTAINER           CPU %               PRIV WORKING SET    NET I/O             BLOCK I/O
224355062ed3        0.00%               52.59 MiB           108 kB / 165 kB     193 MB / 20 MB
1957059358b5        0.00%               52 MiB              115 kB / 172 kB     195 MB / 29 MB
b8457c53fc08        0.00%               47.14 MiB           199 kB / 183 kB     195 MB / 23.5 MB
00b64a98f5bd        0.00%               46.2 MiB            211 kB / 187 kB     195 MB / 27.2 MB
c2afc6cde95d        0.00%               0 B                 0 B / 0 B           0 B / 0 B
ed756d464e85        0.00%               0 B                 0 B / 0 B           0 B / 0 B
c974ba768f75        0.00%               44.84 MiB           339 kB / 235 kB     193 MB / 28.1 MB
95ade33e3ea5        0.00%               45.36 MiB           361 kB / 242 kB     208 MB / 28.2 MB
2ce34cc8f5ce        0.00%               44.34 MiB           339 kB / 244 kB     42 MB / 15.5 MB
ba3b37ea6d0e        0.00%               0 B                 0 B / 0 B           0 B / 0 B
7595ef4873fd        0.00%               0 B                 0 B / 0 B           0 B / 0 B
97f8ec8167da        0.00%               0 B                 0 B / 0 B           0 B / 0 B
4e22ffd333a6        0.00%               0 B                 0 B / 0 B           0 B / 0 B
ac6513ca3be1        0.00%               0 B                 0 B / 0 B           0 B / 0 B
d64b53044337        0.00%               0 B                 0 B / 0 B           0 B / 0 B
9bb72e81d1ef        0.00%               0 B                 0 B / 0 B           0 B / 0 B
53494fdb0cd3        0.00%               0 B                 0 B / 0 B           0 B / 0 B
65c21c4ececb        0.00%               0 B                 0 B / 0 B           0 B / 0 B
10d1d2bc951f        0.00%               0 B                 0 B / 0 B           0 B / 0 B
4094fcc65a6e        0.00%               0 B                 0 B / 0 B           0 B / 0 B
98447eed0821        0.00%               0 B                 0 B / 0 B           0 B / 0 B
c9fa0846c74f        0.00%               0 B                 0 B / 0 B           0 B / 0 B
94b56240c2d6        0.00%               0 B                 0 B / 0 B           0 B / 0 B
899959babe06        0.00%               0 B                 0 B / 0 B           0 B / 0 B
880108ba4134        0.00%               0 B                 0 B / 0 B           0 B / 0 B
ae0d4ee1b802        0.00%               0 B                 0 B / 0 B           0 B / 0 B
3c2136f10aa9        0.00%               0 B                 0 B / 0 B           0 B / 0 B
457ab01633a9        0.00%               0 B                 0 B / 0 B           0 B / 0 B
a2122c634d4a        0.00%               0 B                 0 B / 0 B           0 B / 0 B
57ca5ad379db        0.00%               0 B                 0 B / 0 B           0 B / 0 B
```

Figure 4-9: The output from running the `docker stats` command

To view the status of a specific container, include the container's ID in the `docker stats` command, as shown in the following:

```
docker stats <containerID>
```

Now, if you need to see inside the container (without running a `docker exec` or `attach` command), use the `docker top` command, as follows:

```
docker top <containerID>
```

Figure 4-10 illustrates that the output shows the top running processes within a container.

```
PS C:\Users\msadmin> docker top 1957059358b5
Name                PID                 CPU                 Private Working Set
smss.exe            10756               00:00:00.078        225.3 kB
csrss.exe           10840               00:00:00.359        888.8 kB
wininit.exe         10508               00:00:00.109        700.4 kB
services.exe        5964                00:00:00.171        1.593 MB
lsass.exe           9476                00:00:00.171        3.17 MB
svchost.exe         10368               00:00:00.062        1.995 MB
svchost.exe         10872               00:00:00.156        1.638 MB
svchost.exe         7796                00:00:00.062        2.015 MB
svchost.exe         10644               00:00:00.203        4.141 MB
svchost.exe         11000               00:00:04.265        10.59 MB
svchost.exe         11136               00:00:00.140        2.765 MB
svchost.exe         7648                00:00:01.531        3.99 MB
svchost.exe         10680               00:00:00.078        3.195 MB
svchost.exe         11068               00:00:00.000        823.3 kB
svchost.exe         10704               00:00:00.468        3.99 MB
CExecSvc.exe        10280               00:00:00.062        688.1 kB
svchost.exe         10800               00:00:00.062        3.416 MB
ServiceMonitor.exe  7192                00:00:00.000        385 kB
msdtc.exe           8632                00:00:00.046        1.741 MB
PS C:\Users\msadmin>
```

Figure 4-10: The output after running the `docker top` command for a container

Configuring networking

Chapter 1 discusses networking for containers in considerable depth. But, in this section, we dive a little deeper into configuring a network on a container host.

By default, when you deploy a container host, it will automatically deploy a default Network Address Translation (NAT) network, unless you have modified the dameon.json configuration file located at c:\programdata\docker\config\deamon.json.

Listing networks

We can look at what networks are created on a host by using the following command:

```
docker network ls
```

Figure 4-11 presents the output. Note that because we have not yet implemented any other networks, it displays the default `nat` network.

```
PS C:\Users\msadmin> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
a82b710cb0ba        nat                 nat                 local
3508e5902243        none                null                local
PS C:\Users\msadmin>
```

Figure 4-11: Listing the Docker networks

Viewing network information

If you want to retrieve more specific information in relation to a deployed network, you can use the command `docker network inspect` command, as follows:

```
docker network inspect <networkname>
```

Figure 4-12 depicts the output, which is in JSON format and shows the configuration of the network as well as the containers that are attached.

Figure 4-12: Network configuration output after running the `docker network inspect` command

Creating networks

You can create additional networks as needed for your container host, but depending on the type of network you want, you'll need to choose from among different deployment options. The two main options we will create here are another NAT network and a transparent network.

To create the new network, use the `docker network create` command along with the `-d` *<networktype>* parameter, which defines the network driver you want to use. For a NAT network, you use `-d nat`, and for a transparent network, use `-d transparent`.

Here's the full syntax to create a NAT network:

```
docker network create -d nat --subnet=10.0.0.0/24 --gateway=10.0.0.1 prodnat
```

This creates a new NAT network to which containers can attach.

To create a transparent network, use the following syntax:

```
docker network create -d transparent prodtransparentnet
```

This creates a transparent network and allows containers to attach and be addressable via the network.

For circumstances in which you need to map the transparent network to the physical network, you also can use the `--subnet` and the `--gateway` parameters, as follows:

```
docker network create -d transparent - -subnet=10.0.0.0/24 - -gateway=10.0.0.1 prodtransparentnet
```

This will not interfere with DHCP on the network; it is just aligning the transparent network with the physical network.

> **Note** It is important to understand that the port mappings (which we discuss in a moment) do not apply to transparent networking.

As mentioned, containers by default attach to the NAT network (if not already overridden). For cases in which the NAT network was not created or you require your container to attach to a different network, you can use the `--network` parameter when invoking a container to attach to the different network. For example, if you want your containers to attach to your new `prodtransparentnet`, you would invoke the container by using the following syntax:

```
docker run -d --network=prodtransparentnet <image>
```

If you do not have a DHCP service on the production network, you'll need to statically assign an IP address to the container. To do this, when you invoke or stop the container, use the `--ip` parameter, as demonstrated here:

```
docker run -d --network=prodtransparentnet --ip=10.0.0.100 <image>
```

This invokes the container with the static IP address of 10.0.0.100.

> **Note** If this is a virtualized container host, you need to turn on Mac Spoofing on the virtual machines Network Interface Card (NIC). You also need to ensure that this is an actual free IP address on the network before allocating it.

Removing networks

To remove old networks, you can use the `docker network rm` command, as follows:

```
docker network rm <netname>
```

Port mapping

Containers in NAT mode are isolated unless you present their endpoint via a port mapping so that it becomes accessible to the outside world. For example, if you run an Internet Information Services (IIS) container image, you will not be able to access the web page until you expose the port.

For a container host for which NAT is involved, you might need to map an outside port to the inside port of the application, especially if you have multiples of the same service using the same port.

A simple example is if you have two IIS containers, both using port 80 for their application, you cannot expose both applications on port 80; thus, on container 1, you will use the external port of 8080 and map it to the internal container on port 80, and for container 2, you will use the external port of 8081 and map it to the internal container on port 80. External clients will access the application via port 8080 or 8081, depending on the IIS application they choose to access.

You must set up port mappings as you are creating a container, or you must stop the container and perform the mapping by using the `-p` parameter, as shown in the following:

```
docker run -it -p <extPort>/<IntPort> <containerimage> <cmdtorun>
```

Also, if you do not specify an external port, Docker will automatically create a dynamic mapping for use, as shown in Figure 4-13.

```
PS C:\Users\msadmin> docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
          NAMES
5fdc3defa353        microsoft/iis       "C:\\ServiceMonitor..."  25 seconds ago      Up 18 seconds       0.0.0.0:40711
->80/tcp   pensive_goodall
0faf53897ec2        microsoft/iis       "C:\\ServiceMonitor..."  46 seconds ago      Up 39 seconds       0.0.0.0:54967
->80/tcp   inspiring_heyrovsky
e360b28666d4        microsoft/iis       "C:\\ServiceMonitor..."  About a minute ago  Up About a minute   0.0.0.0:9908-
>80/tcp    compassionate_darwin
```

Figure 4-13: Docker containers with dynamic port mapping

Binding networks to a specific host adapter

In some cases, when you need to have multiple NICs on a host and you want to map different containers to different networks, you might want to associate the network with a different physical NIC on the container host.

To do this, first run the following command to find the network adapter name to which you want to bind:

```
Get-NetAdapter
```

Using the adapter name, you then can create a new Docker network bound to that specific adapter by using the `-o` parameter, as shown in the following example:

```
docker network create -d transparent -o com.docker.network.windowsshim.interface=<InterfaceName>
<NetName>
```

It you specify multiple network adapter names in the form of "adapter1","adapter2","adapter3", you can create a teamed network for Docker to use.

Virtual LANs

If the container network needs to be on a specific virtual LAN (VLAN), you can use the `-o` parameter again to tag the Docker network to a specified VLAN, as shown here:

```
docker network create -d transparent -o com.docker.network.windowsshim.vlanid=<vlandid> <netname>
```

This creates a network attached to a specific VLAN for traffic routing.

> **More info** To learn more, go to https://docs.microsoft.com/virtualization/windowscontainers/manage-containers/container-networking.

Dockerfiles

To this point, we have seen how to pull an image and run it on a container host, and then utilize the Docker `attach` or `exec` commands to customize the container and then finally commit the changes to a new container so that we can reuse it. However, this is not a very efficient process.

Moving into the world of DevOps, we need to think about automating the process of building out the containers we need across the range of environments on which we might run them.

With Dockerfiles, you can store your container images as code. This becomes important because this makes it possible for you to describe your container image in a particular way with all of its dependencies, and then call the Dockerfiles to build your application.

Take, for example, a scenario we will cover later in this chapter that involves patching a container. Although there are different approaches, the most efficient way of engaging this process is to build Dockerfiles for our container images so that we can swap out the dependencies to the latest version easily and then rebuild our application container, end to end.

Even if you take an application lifecycle, being able to fully describe the deployment process in a Dockerfile gives you the ability to simplify your build cycle.

Basic instructions

There are some basic instructions that you will need to understand before you can actually create a Dockerfile. These instructions perform some action, be it marking a dependent image, running a piece of code, executing a command, and so on.

The escape character for Docker is the backslash (\). This, obviously can be very problematic in Windows because we use that character quite extensively when working with directories, for example. As Table 4-1 illustrates, we need to escape the backslash in a directory name. For example, normally a path to the temp directory might be c:\temp\. But, if you're referencing that directory Dockerfiles, you need to escape the backslash; thus, it becomes c:\\temp\\.

To make life a bit easier, though, you can specify to use the back-tic (`) character instead of the backslash as the escape character by adding the following at the beginning of the Dockerfile:

```
Escape = `
```

Later in this chapter, we show some examples of how this becomes useful.

More info The list of commands in Table 4-1 is not exhaustive. To see all of the available commands for a Dockerfile, go to https://docs.microsoft.com/virtualization/windowscontainers/manage-containers/container-networking.

Table 4-1: Commands for Dockerfile

Instruction	Syntax	Example	Description
FROM	FROM *<image>*	From nanoserver/iis	The FROM instruction sets the base container image from which the container will be derived.
RUN	RUN ["*<exe>*", "*<param1>*", "*<param2>*"] or RUN *<command>*	RUN ["powershell","New-Item", "C:\\Appfiles"] or RUN powershell New-Item c:\Appfiles	The RUN command runs the command inside the container. In the first example, the command runs explicitly; the second example essentially uses the shell of the container (i.e., cmd.com) to run the command
COPY	COPY ["*<Source>*", "*<Destination>*"]	COPY ["App.zip", "C:\\App files\\"	The COPY command duplicates files into the container.

ADD	ADD ["*<Source>*", "*<Destination>*"]	ADD ["app.zip","c:\\app files\\"] or ADD [http://downloadserver/app.zip, "c:\\app files\\"]	The ADD command is similar to the COPY command except that it also can add from remote locations
WORKDIR	WORKDIR *<Path>*	WORKDIR c:\\windows	WORKDIR sets a working directory for other commands
CMD	CMD ["*<exe>*","*<param>*"] or CMD *<command>*	CMD ["C:\\Apache\\bin\\httpd.exe", "-w"] or CMD c:\\Apache\\Bin\\httpd.exe -w	The CMD instruction sets the default command that you want to run when deploying a container instance.

Note A coding best practice is for each command to be all uppercase.

Creating a Dockerfile

Now that you understand the basics of a Dockerfile, let's focus on creating a few samples to build upon. The goal from the beginning is to create a highly optimized build process so that you can deploy your container rapidly to any environment. However, you need to create a basic file and show how we can optimize it with simple changes.

Note Before you begin, you need to sign up for a Docker Account at https://cloud.docker.com/.

A basic Dockerfile

In this example, we will create a simple Dockerfile, which will use the base image of microsoft/iis, and set up a simple web page. To do this, we will use two commands: FROM and RUN.

1. Create a new Dockerfile by using the following Windows PowerShell command:

   ```
   New-Item -type file c:\<yourdirectory>\DockerFile
   ```

2. Open the new Dockerfile using Notepad or your favorite text editor.

3. Type the following at the top of the file, being sure that you use the base image microsoft/iis:

   ```
   FROM microsoft/iis
   ```

4. Type the following on line 2 to create a new web page:

   ```
   RUN echo "MSFT Containers 101" > c:\inetpub\wwwroot\index.html
   ```

   Save your dockerfile.

5. Using the Docker `build` command, create a new image:

   ```
   docker build -t <dockerusername>/<imagename> c:\temp
   ```

   This browses the directory for your Dockerfile and begins the build process, as show in Figure 4-14.

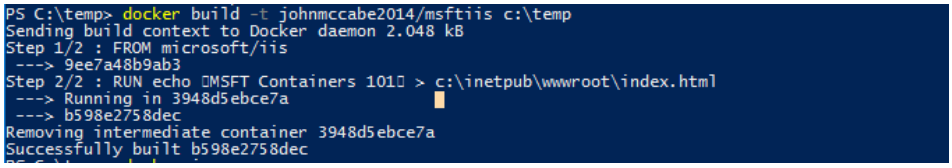

Figure 4-14: Using the `docker build` command

You can, of course, run this container and do a dynamic or static port mapping and view the web page if you want to test utilizing the commands `docker run` and the parameter `-p` described earlier in this chapter.

In this scenario, however, we want to demonstrate pushing the new build to the Docker hub repo.

We present a more complex Dockerfile example in this chapter in just a moment as well as in Chapter 5.

Pushing the image to the repository

You also can push this to the Docker repository by using the `docker push` command so that other members of your team can utilize this new container. To push your image, type the following:

```
docker push <dockerusername>/<imagename>
```

If you haven't done so already, you will be prompted to authenticate by using your Docker credentials for which you signed up earlier.

A complex Dockerfile example

In this example, we want to take a deeper look at and demonstrate some of the more complex mechanisms that you can use to deploy an application framework into a container image.

1. On your container host, using Windows PowerShell, create a new directory called ApacheDeploy, as follows:

   ```
   New-item -type directory c:\Appfiles
   ```

2. Create a new Dockerfile:

   ```
   New-Item -type file c:\Appfiles\Dockerfile
   ```

3. Create a new PS1 file called ApacheInstall.ps1 by using the following commands:

   ```
   New-item -type file c:\appfiles\ApacheInstall.ps1
   ```

4. Create a new PS1 file called VCRedistInstall.ps1, as follows:

   ```
   New-Item -type file c:\appfiles\VCRedistInstall.ps1
   ```

5. Open the Dockerfile in Notepad, copy the following code, and save it:

   ```
   FROM microsoft/windowsservercore
   ADD ApacheInstall.ps1 /windows/temp/ApacheInstall.ps1
   ADD VCRedistInstall.ps1 /windows/temp/VCRedistInstall.ps1
   RUN powershell.exe -executionpolicy bypass c:\windows\temp\ApacheInstall.ps1
   RUN powershell.exe -executionpolicy bypass c:\windows\temp\VCRedistInstall.ps1
   WORKDIR /Apache24/bin
   CMD /Apache24/bin/httpd.exe -w
   ```

6. In Notepad, open the ApacheInstall.ps1 file and copy the following code:

   ```
   Invoke-WebRequest -Method Get -Uri http://www.apachelounge.com/download/VC14/binaries/
       httpd-2.4.25-win64-VC14.zip -OutFile c:\apache.zip
   Expand-Archive -Path c:\apache.zip -DestinationPath c:\
   Remove-Item c:\apache.zip -Force
   ```

7. Save and close the file, and then, in Notepad, open VCRedistInstall.ps1 and copy the following code:

```
Invoke-WebRequest -Method Get -Uri "https://download.microsoft.com/download/9/3/F/93FCF1E7-E6A4-478B-96E7-D4B285925B00/vc_redist.x64.exe" -OutFile c:\vc_redist.x64.exe
start-Process c:\vc_redist.x64.exe -ArgumentList '/quiet' -Wait
Remove-Item c:\vc_redist.x64.exe -Force
```

8. Save and close the file.

   This Dockerfile copies the Windows PowerShell scripts into the container, runs the scripts which will install the required software for Apache to operate.

9. Using the `build` command, create the new image:

```
Docker build -t <reponame>/<containername> c:\apachebuild
```

Figure 4-15 shows the output of the build process for this more complex Dockerfile.

```
C:\AppFiles>Docker build -t johnmccabe2014/apachedemo c:\appfiles
Sending build context to Docker daemon 4.096 kB
Step 1/7 : FROM microsoft/windowsservercore
 ---> b4713e4d8bab
Step 2/7 : ADD ApacheInstall.ps1 /windows/temp/ApacheInstall.ps1
 ---> 11a0c42790e1
Removing intermediate container abf9881bf29b
Step 3/7 : ADD VCRedistInstall.ps1 /windows/temp/VCRedistInstall.ps1
 ---> 4461e3720a88
Removing intermediate container 9c63df376924
Step 4/7 : RUN powershell.exe -executionpolicy bypass c:\windows\temp\ApacheInstall.ps1
 ---> Running in 3b8997426abe
 ---> 41a9e3889461
Removing intermediate container 3b8997426abe
Step 5/7 : RUN powershell.exe -executionpolicy bypass c:\windows\temp\VCRedistInstall.ps1
 ---> Running in 421915f9c081
 ---> f5586b2940c6
Removing intermediate container 421915f9c081
Step 6/7 : WORKDIR /Apache24/bin
 ---> e9646160a436
Removing intermediate container de800c6f8b82
Step 7/7 : CMD /Apache24/bin/httpd.exe -w
 ---> Running in d6c1ef5e7904
 ---> 9ae825785365
Removing intermediate container d6c1ef5e7904
Successfully built 9ae825785365
```

Figure 4-15: A complex Dockerfile build

You now can run this image or deploy an application to Apache.

Docker Swarm

Achieving high availability (HA) on a Windows platform traditionally is accomplished by load balancing or failover clustering. Docker does not support the built-in Windows HA mechanisms but has a mode that will help you to achieve HA across your Docker environment and provide orchestration capabilities. This feature is called Swarm mode.

> **Note** Swarm mode requires that you meet the following prerequisites before proceeding:
>
> - Windows 10 Creative Update or Greater
> - Windows 2016 with April 2017 Cumulative update
> - Docker Engine v1.13 or later

There are two roles in Swarm mode: a manager node and worker nodes. Every node begins with a manager node, which is also responsible for initializing the Swarm, controlling the worker nodes, and maintaining the overall desired state of the applications running on the Swarm. There can be multiple manager and worker nodes in a Swarm.

> **Note** Before creating a Docker Swarm, consider the use of availability sets for the worker nodes if you plan on using a virtualization technology or public cloud. This is to ensure that not all worker nodes and manager nodes end up on the same host.

Initializing a Swarm cluster

In our next example, we will create a Swarm with three machines: one for the manager node, and two worker nodes.

Initializing the manager node

To begin, use the following command to initialize the Swarm mode:

```
docker swarm init --advertise-addr=<HostIPAddr> --listen-addr=<HostIPAddr>:2377
```

This initializes the cluster and outputs the Docker Swarm keys to all worker nodes to join.

Joining an additional manager or worker node

If you close the window to quickly before you record the keys, you can use the following commands to retrieve the keys for the worker and manager:

For a worker node:

```
docker swarm join-token worker
```

For a manager node:

```
docker swarm join-token manager
```

Figure 4-16 displays the output for the worker node.

```
PS C:\Users\msadmin> docker swarm join-token worker
To add a worker to this swarm, run the following command:

    docker swarm join \
    --token SWMTKN-1-0efljm2d804f2kxzgqn5ryhypi4h4uipr5ttbwve9w5dnqpvlw-es7r8uj002wcgcpb3x8gqos8q \
    10.0.0.9:2377
```

Figure 4-16: Retrieving a Docker Swarm join token for a worker node

Copy the output to the worker node to join it to the Swarm cluster. You also can do the same for the manager node.

Figure 4-17 shows the output when a worker node joins a swarm.

```
PS C:\Users\msadmin>   docker swarm join --token SWMTKN-1-0efljm2d804f2kxzgqn5ryhypi4h4uipr5ttbwve9w5dnqpvlw-es7r8uj002wcgcpb3x8gqos8q 10.0.0.9:2377
This node joined a swarm as a worker.
```

Figure 4-17: A worker node joining a Swarm

Viewing your cluster nodes

If you need to understand what makes up your Docker Swarm cluster, you can use the Docker **node** command to display the information, as follows:

```
docker node ls
```

Figure 4-18 displays the results of running the node command, which, in this case, is three nodes with one being the leader.

```
PS C:\Users\msadmin> docker node ls
ID                           HOSTNAME      STATUS  AVAILABILITY  MANAGER STATUS
683uydg6xi6cjthcs35hqo2yp *  Containers01  Ready   Active        Leader
6gwh3uyb2nfo2a2nmxk4jdzgf    Containers02  Ready   Active
zgbvtctorcozfsz10f1ousfdd    Containers03  Ready   Active
PS C:\Users\msadmin>
```

Figure 4-18: Output of `docker node` command

Swarm networking

Now that you are in a cluster, one of the questions that generally arises is how to network across the hosts. How do you span your NAT network and allow machines to communicate or, more important, do you allow this?

Swarm mode introduces a network driver called Overlay (Windows Server 2016 April 2017 CU and Windows 10 Creators Update sets up this feature) The Overlay network is based on VXLAN Technology and you can use it to span multiple container hosts. Each overlay network is assigned its own private IP subnet.

Creating an Overlay network

To create an Overlay network, use the `docker network create` command again, but this time with `-d overlay` as the option:

```
docker network create -d overlay <netname>
```

Using the `network` command again, you can list out the new networks after creation and their mode type, as shown in Figure 4-19.

```
PS C:\Users\msadmin> docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
p2wcxp2jszkq        OverLayNet          overlay             swarm
d68b2ed3396f        TransNet            transparent         local
xmxmgqyoomcv        ingress             overlay             swarm
6f186f2c68fd        nat                 nat                 local
429fe09944f8        none                null                local
PS C:\Users\msadmin>
```

Figure 4-19: Output of Docker networks and their types

Deploying services

In Swarm mode, the terminology changes slightly from standard "images" and "containers" to "services." Essentially, a service is a container, except now on a Docker Swarm cluster.

Deploy a simple service

We can deploy one of the previous containers we downloaded or created earlier to the swarm cluster as follows:

```
docker service create microsoft/iis
```

This deploys our container to the Swarm, which you can verify by using the `docker service` command to browse what services are deployed, as follows:

```
docker service list
```

Figure 4-20 presents the output.

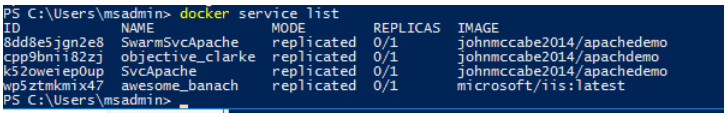

Figure 4-20: Output of the `docker service list` command

If you want to give the service a manageable name in the Swarm cluster instead of the one randomly generated for it, you can include the `--name` parameter.

You also can bind the service to a specific overlay network by using the `--network=<networkname>` parameter. The command for creating a service would thus be as follows:

```
docker service create --name=<servicename> --network=<overlaynetname> <containerimage>
```

Scale, load balancing, and port exposure

When you want to increase the availability of your service on a Swarm cluster, one of the first things you must do is scale the service. You can do this by using the `docker service scale` command, as follows:

```
docker service scale <servicename>=<replicainstancesrequired>
```

You can run this command using the information in the Name field for the `servicename` (see Figure 4-20) and choose a number (i.e., 1, 2, 3, etc.) for the `replicaInstancesrequired`, you can scale the service. After you run the `scale` command, use the `docker service list` command again to view the replica changes happening on your service.

When scaling as just described, load-balancing across the services come into question with regard to how we achieve this. Whereas Docker Swarm has support for a few different options, with Windows Containers, only two are currently supported: DNS Round Robin, and external load-balancing using published ports.

To turn on DNS Round Robin, select the `--endpoint-mode dnsrr` parameter on the `docker service create` command, as follows:

```
docker service create --name=<servicename> --endpoint-mode dnsrr --network=<overlaynetname>
<containerimage>
```

Finally, if you want to allow clients into the service or expose the service endpoints so that we can externally load balance, we can use the `--publish` command on `service create` to achieve this:

```
docker service create --name=<servicename> --publish mode=host,target=<containerport> --
network=<overlaynetname> <containerimage>
```

Mixed mode clusters

Docker Swarm can manage mixed node clusters so that you can have Linux and Windows nodes managed by the same Swarm manager. However, to ensure that you don't try to deploy a service on a Linux node that is a Windows service, or vice versa, you must ensure that you add a label to our nodes so that during deployment you can set deployment constraints that are based on the label.

To add a label to a cluster node, use the `docker node` command, as shown here:

```
docker node update --label-add os=<windows/linux> <NodeName>
```

In the preceding example, `--label-add` is followed by a label of our designation called `os`, which, in this example, should have a value of `windows` or `linux`; however, a label also could be something like `dept=finance`, and so on.

Now, when deploying a service to a Swarm cluster, you can utilize the `--constraint` parameter and designate the node labs, as shown in the following example:

```
Docker service create --name=TestSVC --endpoint-mode dnsrr --network overlaynet --constraint
'node.labels.os=windows' microsoft/iis
```

Alternatively, `'node1.labels.os'` could equal Linux depending on the service type you are deploying.

Docker compose

You can use Docker compose to build multicontainer applications to produce an end-to-end service consisting of databases, queues, frontends, and more.

The first thing you need to do is download and install docker-compose.exe. From an elevated Windows Powershell session, use the following script:

```
$uri = 'https://github.com/docker/compose/releases/download/1.9.0/docker-compose-Windows-
x86_64.exe'
Invoke-WebRequest -Uri $uri -UseBasicParsing -Outfile $Env:ProgramFiles\docker\docker-compose.exe
```

Next, you need to create a Docker compose file, which you'll use to help define services, networks, and volumes. The compose file is a YAML (YAML Ain't Markup Language) file with an extension of .yml or .yaml.

The docker compose file shown in Figure 4-21 is taken from GitHub. The compose file comprises a .yml file called docker-compose and two subdirectories called db and web. Inside the db and web folder, you can view previously created Dockerfiles and artifacts that are needed for the deployment of the application.

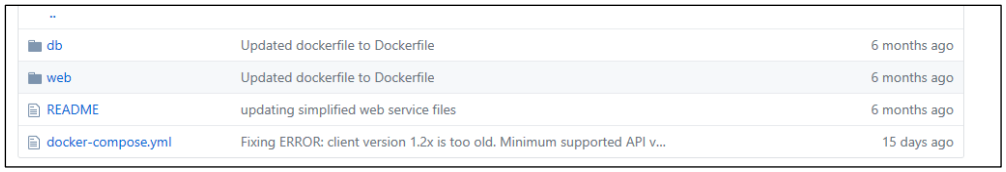

Figure 4-21: GitHub example layout

Now, let's examine the docker-compose.yml file:

```
version: '2.1'
services:
 web:
  build: ./web
  ports:
   - "80:80"
  depends_on:
   - db
  tty:
    true
 db:
  build: ./db
  expose:
   - "1433"
  tty:
    true
networks:
 default:
  external:
   name: "nat"
```

The compose file begins by defining the service version and then defines the services you want to deploy. In this case, as with Docker Swarm, the service refers to the containers that make up the service. The file shows two services defined: web and db.

If we examine the web section, you see first the `build` statement, which is referencing the directory ./web, which during deployment will go and look into the directory and use the references in the Dockerfile to build out the web part of the service. It also exposes a port during creation for the

service. Another interesting item, the `depends_on` field, instructs docker to wait until the db service has been created before building the web end.

Now, we can use the `docker-compose build` and `docker-compose up` commands from within the downloaded sample to run the container. `docker-compose build` reads the compose file and begins to build out the service and the support images, whereas `docker-compose up` brings the service alive.

More info To read more about `docker-compose`, go to https://docs.docker.com/compose/.

Azure Container Service

Azure Container Service (ACS) is a public cloud–based service that facilitates the quick creation of a container service where you can migrate your existing containerized applications to rapidly utilizing industry-standard tooling like Docker, DC/OS, Kubernetes, and so on.

ACS utilizes the Docker container format to ensure portability among private or public cloud deployments of containers.

In this section, we want you to focus on the Docker sections of the procedure. We provide a high-level guide to the public documentation in order to create an Azure container service and deploy applications to it. The guides also cover DC/OS and Kubernetes.

Deploying ACS

To deploy ACS, go to https://docs.microsoft.com/azure/container-service/container-service-deployment for the complete up-to-date procedure. There, you find the step-by-step guidance required, including the prerequisites for the service and also the steps to create it via an ARM template.

Connecting with an ACS cluster

To connect and manage an ACS cluster, go to https://docs.microsoft.com/azure/container-service/container-service-connect to view the full procedure.

Deploying apps to an ACS solution by using Docker Swarm

To deploy your applications manually or via docker-compose, go to https://docs.microsoft.com/azure/container-service/container-service-docker-swarm.

Docker Swarm continuous integration

To understand how you can bring your ACS Docker Swarm and integrate it into your continuous integration services go to https://docs.microsoft.com/azure/container-service/container-service-docker-swarm-setup-ci-cd.

Service Fabric and containers

Service Fabric is a distributed systems platform that makes it easy to package, deploy, and manage scalable and reliable microservices. Service Fabric provides cluster management and orchestration capabilities to ensure service reliability of the applications being deployed.

When approaching containers, we often begin to divide up traditional applications into more pieces, which will begin to look like a microservices approach.

Service Fabric utilizes Docker under the hood to provide lifecycle management for its containers. It supports two modes of operation today: Guest and Service Fabric Inside a Container.

Guest container

The Guest container part of this operates much like an application being deployed to a Service Fabric deployment: you create a service manifest, which references the container image you require and, utilizing Docker, it deploys the container on the Service Fabric as required.

Here is a sample manifest file:

```
<ServiceManifest Name="DemoServiceTypePkg" Version="1.0">
    <ServiceTypes>
        <StatelessServiceType ServiceTypeName="DemoServiceType" ... >
        </StatelessServiceType>
    </ServiceTypes>
    <CodePackage Name="CodePkg" Version="1.0">
        <EntryPoint>
            <ContainerHost>
               <ImageName>Microsoft/iis</ImageName>
               <Commands></Commands>
            </ContainerHost>
        </EntryPoint>
    </CodePackage>
    . . .
</ServiceManifest>
```

Service Fabric services inside a container

It is possible to create a Service Fabric cluster and then deploy a container Service Fabric within the cluster to support your workloads, as necessary. It will operate as a normal Service Fabric cluster, only containerized.

Deploy Windows Containers on Service Fabric

A detailed discussion on how to deploy a container to a Service Fabric is beyond the scope of this book. If you would like to learn more, go to https://docs.microsoft.com/azure/service-fabric/service-fabric-deploy-container. There, you can find several examples that demonstrate how to utilize Service Fabric on Azure.

Is this book useful?
Did it meet your expectations?
Is there room for improvement?

Let us know at http://aka.ms/tellpress

Your feedback goes directly to the staff at Microsoft Press, and we read every one of your responses. Thanks in advance!

Microsoft

CHAPTER 5

Deep dive: containerizing your application

In this chapter, we walk you through a deep dive on how you should approach the journey to containerize your applications. Included in this chapter, we discuss the tooling available today to assist you, and we set up a scenario in which we move a traditional application to a containerized environment.

Methodology

Unfortunately, there is no silver bullet when it comes to containerizing your application. Over time, applications have been created in many ways. Developers might have selected a preference for a programming language or chosen to implement their applications in ways that might not suit containerization. There are a significant number of factors that ultimately might stack up against an application for initial containerization. It is also important to note that while some applications might not be good initial candidates for containerization, this simply highlights that the system needs to be modernized, which could lead to code refactoring or a new development.

Although containerizing does provide a lot of benefits, the overall mindset with which you should approach this topic is one of *application modernization*. Application modernization is a complex journey by which you configure your applications to be more cloud aware. This does include containerization, but it is not necessarily the end of the journey. Containerization can provide the first steps to help an enterprise separate an application into a microservices-like architecture, and then eventually to a full microservice-based system like Service fabric.

Legacy application considerations

Selecting a legacy application does not need to be a complex task. Plus, there are some basic considerations that can help you to determine whether a candidate application is a good one—at least initially! In this section, we describe some of these considerations.

Source code—what programming language

If the original source code is old and written in something like COBOL, Fortran, or other legacy languages, this will almost certainly lead to a code refactoring, which can become expensive. This is not necessarily a bad thing because all applications need to be updated at some point. However, an organization might find itself unable to refactor the code. We will, however, stress that applications written in a modern development language will lend themselves to containerization easier than older languages.

Application type

Is this application a mainframe application, or is it an *N*-tier architecture with database, worker, and web roles? If it is a mainframe application, this will not be a good candidate to containerize and will require refactoring. A traditional *N*-tier or single-server application at least initially looks like a more appropriate candidate to containerize.

User interface

If an application has a server-side user interface that is required to run for the application to function, this also will not be a good candidate to migrate as a container. However, if the server-side interface application could be decoupled from the main service running the server, it becomes a more viable candidate for containerization.

Containers are inherently designed to run almost as headless servers; for example, a windows service like IIS that listens on port 80 but does not contain any state except for the web pages it serves. IIS is a single service. In our example, we would have a single IIS and then a single web site serving one purpose like a home page. Another example would be that in an *N*-tier architecture, we could have our web and worker roles in two separate containers and users accessing the application via a website, which would connect to the web role and subsequently through the worker role to the database virtual machine (VM) or container.

State

Containers do not maintain state, if you reboot a container, whatever existed within the container at that time is destroyed. When selecting an application, this is an important item to consider because state does not only mean where does it store my data; it also means where does it store any transient information it requires to function.

For example, if you have an application that writes a transaction log on the local filesystem and another service of that application reads that transaction log and verifies the information but a container is restarted, what happens to the transaction log? Also consider whether the configuration files might change? If you need them to persist after a container restart, are you sure you have the latest ones in your container image?

Applications being selected for containerization need to be able to externalize the state of the application, be it data or configuration files or log files. Containers offer methods to do this in simple ways by allowing mounting volumes from part of the host system into the container so that we can maintain state. Additionally, we also can connect to external systems for the data or the configuration.

For example, we can configure a container for a IIS and configure IIS to point to a share for a configuration file, which defines how it is supposed to be configured. After it retrieves the configuration file on initialization, the website container will know which server to connect for its data store. The logs for the website are logged out to the external volume provided by the host, and we can collect the data from there and process it in other tools.

In many cases, enterprises might want to examine how they can essentially provide data as a service for their applications and create those interfaces as necessary. This will facilitate easier decoupling of the applications and support moving toward application modernization.

Multiservice single box

If we have an application that has multiple services on a single box, we need to examine the possibility of splitting these services. The developers might have written the application to communicate only across the local system and not over TCP connections between systems. If the application has been written to be on only a single system, code refactoring will be required to allow you to break apart the layers of the application.

High availability

If the application was never designed for high availability (HA), we must consider the potential impact of containerizing it. Although this is not a technical blocker, because you can simply invoke one container, it is more the operational process you need to have in place to ensure that the application is not scaled! It might be beneficial for the enterprise to understand how the application might be designed or refactored to support HA.

Identity

Containers themselves do not technically have any identity. They also are not domain joined. This again is not technically a blocker, but an enterprise needs to understand how its applications authenticate and how moving to containers will affect this. We can provide a group-managed service account that can be used to allow the application to connect into domain-based systems, but can the application support this method? Refactoring of the authentication and authorization methods might be required.

Monitoring and auditing

All applications in an enterprise eventually are subject to monitoring and auditing. Because containers are not domain joined and are stateless, we need to consider how we will retrieve information about the application running and information on the health of the container itself. This again is not a technical blocker in most cases, but it is a point that you need to review, especially in industries that have defined policies from something like SOX or PCI DSS.

Some approaches to ensuring compliance to policies are as follows:

- Using a Docker Rest API to collect metrics and information on the container
- Running Docker Stats to retrieve container metrics
- Utilizing Microsoft OMS with an agent deployed to the host to obtain container metrics
- Writing a service inside the container to output to the stdout stream
- Using `Docker exec` or `Docker cp` to copy the log files out of the container and parse them elsewhere

Although this is not a definitive list of everything you need to understand and address before selecting an application for containerization, it will provide some of the necessary high-level items to help select a suitable application initially.

Moving the application

In this section of the methodology, there are two main approaches after you have selected your application: lift-and-shift, and microservices

Lift-and-shift

For most applications that meet the criteria we explained earlier in this chapter, a lift-and-shift into a container for other applications is a possible approach to delving into containerization. Even though you probably will need to significantly debug the application in a container and provide "fixes" to get it to work, it is possible to stuff a single app with dependent services inside and container and run it. However, it is considered a dirty approach and will end up costing more time in the long run.

By comparison, you can containerize applications like ASP.NET, .NET Apps, web apps and so on relatively easily. Figure 5-1 shows how we can take a three-tier application like the ones mentioned, including SQL, and bring them to a containerized world.

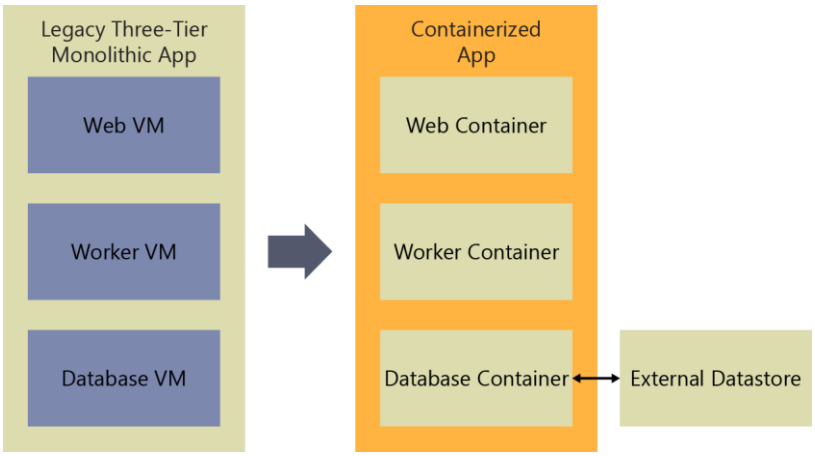

Figure 5-1: Containerizing a three-tier monolithic app

In Chapter 4, we looked extensively at containers and Dockerfiles. We can build a container with all of the dependencies and import the application into the container during deployment. Using those techniques, we can easily migrate an application to a container and be up and running quickly.

Microservices-based approach

While lift-and-shift will get you to containers quickly, we must assess if that is truly the appropriate approach from the perspective of long-term strategy. Although refactoring the code of your application to a full microservices platform might not be feasible, taking steps to split the application so that it represents functional components of a microservices architecture might be possible. These components essentially will be a microservices-based approach.

Figure 5-2 shows you the structure of a simple web application and how we can potentially split it into containers. Although this approach will require some refactoring and reconfiguring the application to ensure that each component can be redirected to talk to another container, it does at least begin the journey toward application modernization with a microservices-based approach.

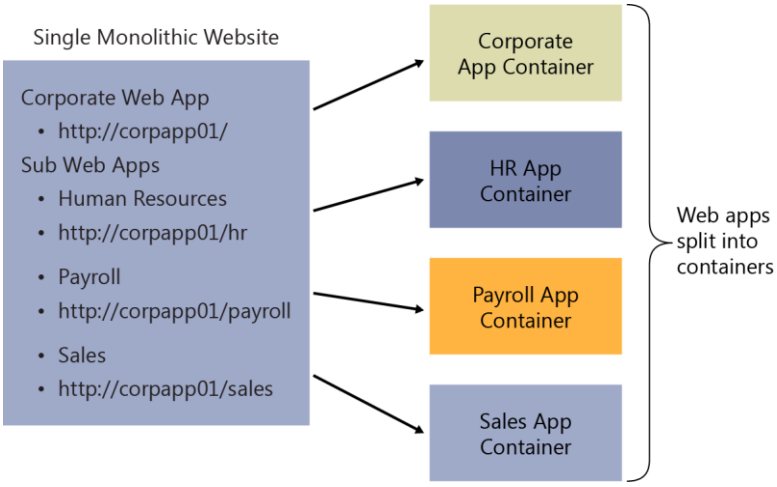

Figure 5-2: Splitting a web app into individual containers

We can dive further into the microservices approach if we then focus on a single application and look at its individual components. Take, for example, the Sales application. In that app, we might have a customer management app, a quoting application, a point-of-sales system app, and so on. We can even have additional functions that are in each app, which we could split away and share if we think about the overall architecture. However, if we just think in terms of the sales app itself, we show how we can further break it down into containers servicing the specific functions in the sales app itself.

Figure 5-3 visualizes this approach and dividing one step further in which we take a subapplication from the sales app and potentially containerize it further, ultimately ending up with a microservices-based approach using containers.

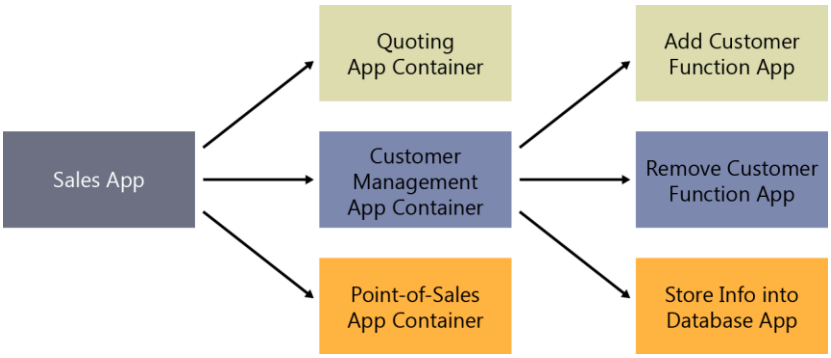

Figure 5-3: Breaking down a web app into individual containers and further diving the applications function into containers

Although this is not a true microservices architecture and is indeed a very simple example, this should provide as a framework for the thought process involved to get you to split an application into the necessary parts.

Tools

In this section, we take a brief walkthrough of some of the available tools, many of which are more than likely tools that are in use in your environment today. These tools can start you and serve you on your journey into containers.

As we know from Chapter 2, we support most modern languages across Linux and Windows allowing almost any application today to exist in a container ecosystem, if designed correctly.

Just for awareness, you can, with little or no tooling, apart from the Docker engine and client, migrate applications into containers. In Chapter 4, we discovered Dockerfiles and the ability to install applications and dependencies in a variety of different ways. We used Notepad to create the Dockerfiles. We then used the docker tools to deploy the applications to our container host.

There are additional tools beyond Docker and Notepad, and we will discuss some of them here. These tools will give you a more integrated experience so that you can develop and deploy seamlessly.

Microsoft Visual Studio 2017

Docker support is now natively built in to Visual Studio 2017, providing a seamless experience from design, build, debug, and deploy to local container host or Microsoft Azure Container Services. It will also support .NET FX on Windows Server Containers and .NET Core on Windows Server (Nano) and Linux Containers.

> **More info** To learn more about Visual Studio 2017 and the native Docker support, go to https://blogs.msdn.microsoft.com/webdev/2016/11/16/new-docker-tools-for-visual-studio/.

Visual Studio 2015—Visual Studio Tools for Containers

Container support is via this extension to Visual Studio 2015. With this toolset, you can build, debug, and deploy apps locally or Azure-hosted containers. Developers also gain multiproject debugging for single and multicontainer scenarios.

> **More info** To read more about the Visual Studio Tools for Containers for Visual Studio 2015, go to https://docs.microsoft.com/dotnet/articles/core/docker/visual-studio-tools-for-docker.

Docker for Azure

Get started building, assembling, and shipping containerized applications on Azure. The native Azure application provides an integrated, easy-to-deploy Docker environment, optimized to use the underlying Azure Infrastructure as a Service (IaaS) platform. This includes Docker Datacenter. Docker Datacenter uses prebuilt cloud templates to develop and run containerized apps directly in the Azure cloud. Docker Datacenter delivers efficiency of computing and operations resources through Docker-supported container management and orchestration.

> **More info** To see more about Docker for Azure, go to https://docs.docker.com/docker-for-azure/.

Azure Container Service

Start building, assembling, and shipping applications in Azure—no additional software installation required. This native Azure app provides an integrated, easy-to-deploy environment that uses the underlying Azure IaaS and a modern Docker platform to deploy portable apps. Standard Docker tooling and API support are included.

> **More info** For more information on Azure Container Service please the following link https://docs.microsoft.com/en-us/azure/container-service/.

.NET Core tools

These tools provide a seamless experience for Windows, Linux, and Mac OS developers. Optimized for high-scale, high-performance microservices, these tools make building containerized .NET apps a breeze.

> **More info** To learn more about the available .NET Core tools, go to https://hub.docker.com/r/microsoft/dotnet/.

Image2Docker

Point this Windows PowerShell module at a virtual hard drive image, scan for common Windows components, and suggest a Docker le. The tool supports VHD, VHDK, and WIM, with a conversion tool for VMDK.

> **More info** For more information regarding Image2Docker for Windows, go to https://github.com/docker/communitytools-image2docker-win.

Examples

Now you know the methodology for approaching how to containerize our applications as well as the considerations for doing so.

When writing this book, we decided to not include a definitive example showing you the process; rather we decided to go with examples that are updated on a regular cadence by the blog owners. This makes it possible for us to give you the latest information available even faster updating this ebook! However, you will find plenty of examples throughout this book on top of which of all these items are built.

Migrating ASP.NET MVC applications to Windows Containers

https://docs.microsoft.com/aspnet/mvc/overview/deployment/docker-aspnetmvc

Running console applications in containers

https://docs.microsoft.com/dotnet/articles/framework/docker/console

Convert ASP.NET Web Services to Docker with Image2Docker

https://blog.docker.com/2016/12/convert-asp-net-web-servers-docker-image2docker/

Running SQL Server + ASP.NET Core in a container on Linux in Azure Container Services

https://blogs.msdn.microsoft.com/maheshkshirsagar/2017/02/14/running-sql-server-asp-net-core-in-a-container-on-linux-in-azure-container-service-on-docker-swarm-part-1/

Using Visual Studio to automatically generate a CI/CD pipeline to deploy ASP.NET Core web apps with Docker to Azure

https://www.visualstudio.com/docs/build/apps/aspnet/aspnetcore-docker-to-azure

About the authors

John McCabe works for Microsoft as a senior premier field engineer. In this role, he has worked with the largest customers around the world, supporting and implementing cutting-edge solutions on Microsoft Technologies. In this role, he is responsible for developing core services for the Enterprise Services Teams. John has been a contributing author to several books, including *Mastering Windows Server 2012 R2* from Sybex, *Mastering Lync 2013* from Sybex, and *Introducing Microsoft System Center 2012* from Microsoft Press. John has spoken at many conferences around Europe, including TechEd and TechReady. Prior to joining Microsoft, John was an MVP in Unified Communications with 15 years of consulting experience across many different technologies such as networking, security, and architecture.

Michael Friis is a product manager at Docker where he works on Docker for Amazon Web Services and Azure. He also focuses on integrating Docker with Microsoft technology. Previously he was at Heroku, and, before that, AppHarbor, a .NET platform as a service.

From technical overviews to drilldowns on special topics, get *free* ebooks from Microsoft Press at:

www.microsoftvirtualacademy.com/ebooks

Download your free ebooks in PDF, EPUB, and/or Mobi for Kindle formats.

Look for other great resources at Microsoft Virtual Academy, where you can learn new skills and help advance your career with free Microsoft training delivered by experts.

Microsoft Press