8

Introducing Containers

In this chapter, we cover the following recipes:

* Configuring a container host
* Deploying a hello world sample
* Deploying IIS in a container
* Using a Dockerfile to create and use a container

# Introduction

As a method of Unix virtualization, containers have been around for quite a while. To a large degree, containers serve as an approach for deploying applications popularized by the open source Docker initiative. With Windows 2019, Windows Server supports Docker and Docker containerization integrated with Hyper-V.

Most of the administration you are likely to do with containers in Windows Server 2019 is done not by cmdlets, but by a command-line tool called docker.exe. For those used to PowerShell's object-oriented and task-focused approach, you may find this application hard to use. I daresay you are not alone. The docker.exe application works in PowerShell and you can, of course, use PowerShell to wrap the command.

With containers in Windows Server 2019, you need to download and install a number of components. In the Deploying a hello world sample recipe, you download and make use of OS base images. These require an internet connection.

Containers provide scalability by enabling you to run multiple containers directly on top of Windows Server 2019. This takes up considerably fewer resources than if each container was contained in its own separate virtual machine (VM).

This approach has a theoretical security vulnerability whereby malware enables bad actors to access one container's contents from another. To reduce those risks, you can run containers inside Hyper-V. With Hyper-V containers, the container is run inside a completely virtualized environment that provides additional hardware-level security, albeit at the price of performance. Hyper-V containers are useful in a shared tenant environment, where one container host can run containers belonging to different organizations.

Once you have configured a container host, it's a great idea to test that you can run containers successfully. There are a number of sample containers you can download to test out the basic container functionality (and the use of docker.exe). You use these in the Deploying a hello world sample recipe, as well as look at using containers with Hyper-V virtualization.

With containers, you package applications inside a container, which then makes use of a shared kernel. The (single) shared kernel provides kernel-level features for all the containers deployed on a container host. The container then runs an application making use of the shared kernel.

To deploy containers in Windows Server 2019, you need to provide both a container host (to run the container) and one or more images, which Docker can run as a container. You can also download base operating system images from the Docker repository and build your own images on top.

You build and deploy containers using docker.exe (and in some cases, Hyper-V), as you see in this chapter. In the Deploying a hello world sample recipe, you will explore and download key base images.

An application you can easily containerize is IIS. This is pretty simple, as you can see in the Deploying IIS in a container recipe.

If you are deploying containers, you can use a Dockerfile, a simple text file with build instructions. Docker uses this file to build a customized container image, which you can then deploy. You look at creating and using an image using a Dockerfile in the snappily-named Using a Dockerfile to create and use a container recipe.

This chapter provides only an introduction to containers, images, docker.exe, and Dockerfile files. There is much more to explore with containers. Topics including Docker networking, Docker Swarm, and more are outside the scope of this book. To discover more about containers than we can fit here, look at Packt's book: Learning Windows Server Containers by Srikanth Machiraju. And, for more on the endearingly awful docker.exe application, take a look at Docker on Windows by Elton Stoneman.

For more information on Windows containers, see this link: <https://docs.microsoft.com/virtualization/windowscontainers/about/>.

Note

The recipes in this chapter use the command-line tool docker.exe. For those familiar and comfortable with all of PowerShell's awesomeness, this is going to come as a bit of a shock. docker.exe has no tab completion, all output is minimal text blobs (no objects), parameter names seem random and curious, the online help is not very helpful, and the error reporting is downright atrocious. docker.exe is not all that easy to get to grips with, is less easy to automate than other Windows features, and feels very, very slow even on a well-equipped workstation. Containers as a feature are awesome—if you plan to adopt them, consider spending some time building a good framework and framework tools for your environment. Additionally, using major search engines to discover aspects of containers tends to yield a lot of useful pages, but focused on Linux as a container host and is used in a container.

# Configuring a container host

The first step in containerization is to configure a container host. The container host is a machine (virtual or physical) running Windows Server 2019 with the necessary container-related services and prerequisites installed and running. You can also run containers on Windows 10, but this is outside the scope of this chapter.

In this recipe, you install the components necessary for containers, including loading the Docker components.

## Getting ready

This recipe uses a new Windows Server 2019 system, named CH1. At the start of this chapter and this recipe, CH1 contains only the base-installed Windows features and has not been used for other recipes in this book.

## How to do it...

1. Install the nuget provider:

Install-PackageProvider -Name nuget -Force

1. Install the Docker provider:

$IHT1 = @{

Name = 'DockerMSFTProvider'

Repository = 'PSGallery'

Force = $True

}

Install-Module @IHT1

1. Install the latest version of the Docker package. This also enables the containers feature in Windows Server:

$IHT2 = @{

Name = 'Docker'

ProviderName = 'DockerMSFTProvider'

Force = $True

}

Install-Package @IHT2

1. Ensure Hyper-V and the Hyper-V management tools are installed:

Install-WindowsFeature -Name Hyper-V -IncludeManagementTools |

Out-Null

1. Remove Windows Defender as it can interfere with Docker:

Remove-WindowsFeature -Name Windows-Defender |

Out-Null

1. Restart the computer to enable Docker and Windows Containers:

Restart-Computer

1. Check that the Windows Containers and Hyper-V features are installed on CH1:

Get-WindowsFeature -Name Containers, Hyper-v

1. Next, check the Docker service:

Start-Service -Name Docker

Get-Service -Name Docker

1. Check the Docker version information:

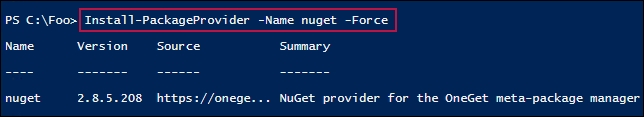
docker version

1. Display the Docker configuration information:

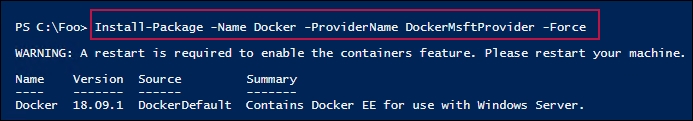
docker info

## How it works...

In step 1, you add the nuget package provider. The output of that step looks like this:

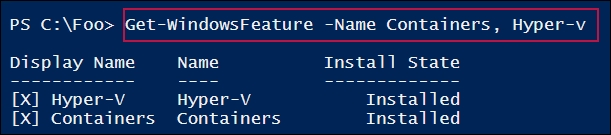


In step 2, which produces no output, you install the DockerMSFTProvider provider. Next, in step 3, you install the latest version of the Docker packages. This step produces the following output:

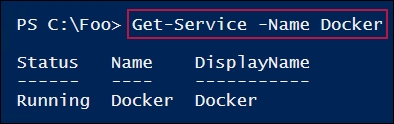


In step 4, you install the Hyper-V feature and related management tools. In step 5, you remove Windows Defender, which can interfere with Docker. In step 6, you reboot the system to complete the installation of the core container features. These three steps produce no output.

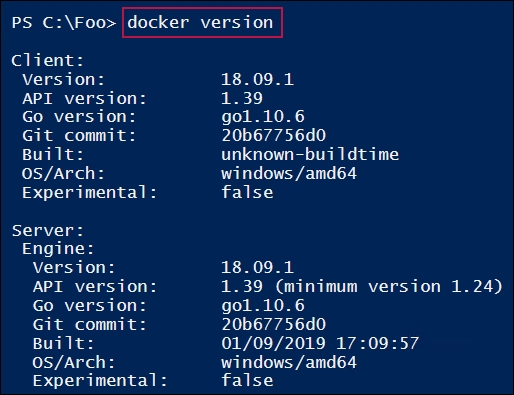
After CH1 has restarted, in step 7, you check on the containers and Hyper-V Windows features, which looks like this:



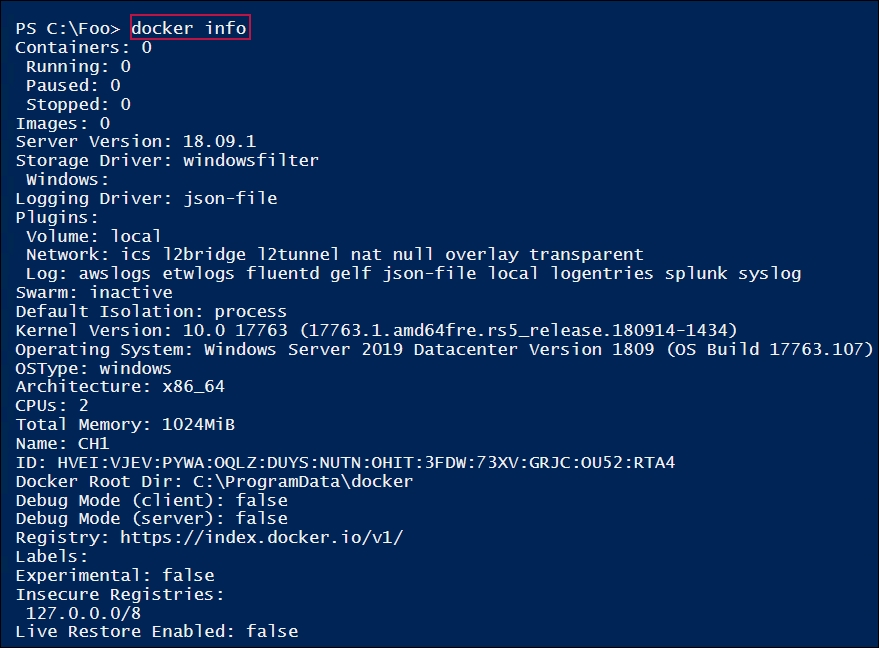
In step 8, you check to ensure that the Docker service is running, which looks like this:



In step 9, you use the docker.exe command to check the Docker client and server version details, which looks like this:



In step 10, you view the Docker configuration details, which looks like this:



## There's more...

In step 1, you download and install the NuGet provider. This enables you to get the Docker provider and the Docker application package from the PowerShell Gallery. For more details on NuGet, see: <https://docs.microsoft.com/nuget/what-is-nuget>.

# Deploying a hello world sample

Once you have a container host configured, you need to ensure that your environment has been configured successfully and can utilize containers. A really simple way to check that all is well on your container host is by downloading and running an application. Running this containerized application successfully shows that you have containers and Docker set up.

There are a few simple applications you can utilize that test the container environment on CH1 (which you set up in the Configuring a container host recipe). You use a few of them in this recipe.

Before you can run a container, you must acquire a container image. There are several ways to obtain images as you see in this chapter. Docker maintains an online registry that contains a variety of container images for you to leverage. Using the docker command, you can search and download images either to use directly or to use as the basis of a custom-built container (for example, as you can see in the Using a Dockerfile to create and use a container recipe). In this recipe, you download several images, run them as containers, and then clean up your environment.

This recipe demonstrates using the Docker registry to obtain images, and then using those images locally. This recipe looks at some basic container management tasks and shows some methods to automate the docker.exe command.

## Getting ready

This recipe needs a configured container host, CH1. You set up this host in the Configuring a container host recipe.

## How to do it...

1. Find any hello-world containers at the Docker registry:

docker search hello-world

1. Pull the official Docker hello-world container image:

docker pull hello-world

1. List the images on CH1:

docker image ls

1. Run the hello-world container image:

docker run hello-world

1. Look for Microsoft images in the Docker registry:

docker search microsoft

1. Get the nanoserver base image:

docker image pull mcr.microsoft.com/windows/nanoserver:1809

1. Run the nanoserver base image:

docker run mcr.microsoft.com/windows/nanoserver:1809

1. Check the images available now on CH1:

docker image ls

1. Inspect the first image:

$Images = docker image ls

$Image = (($Images[1]).Split(' ')|where {$\_.Length -ge 1})[2]

docker inspect $image | ConvertFrom-Json

1. Get another (older) image and try to run it:

docker image pull microsoft/nanoserver | Out-Null

docker run microsoft/nanoserver

1. Now, run the image with Hyper-V isolation:

docker run --isolation=hyperv microsoft/nanoserver

1. Examine the differences in runtimes with Hyper-V:

# Run the container with no isolation

$S1 = Get-Date

docker run hello-world | Out-Null

$E1 = Get-Date

$T1 = ($E1-$S1).TotalMilliseconds

# Run the same container with isolation

$S2 = Get-Date

docker run --isolation=hyperv hello-world | Out-Null

$E2 = get-date

$T2 = ($E2-$S2).TotalMilliseconds

# display the difference

"Without isolation, took : $T1 milliseconds"

"With isolation, took : $T2 milliseconds"

1. Remove all container images:

docker rmi $(docker images -q) -f | out-Null

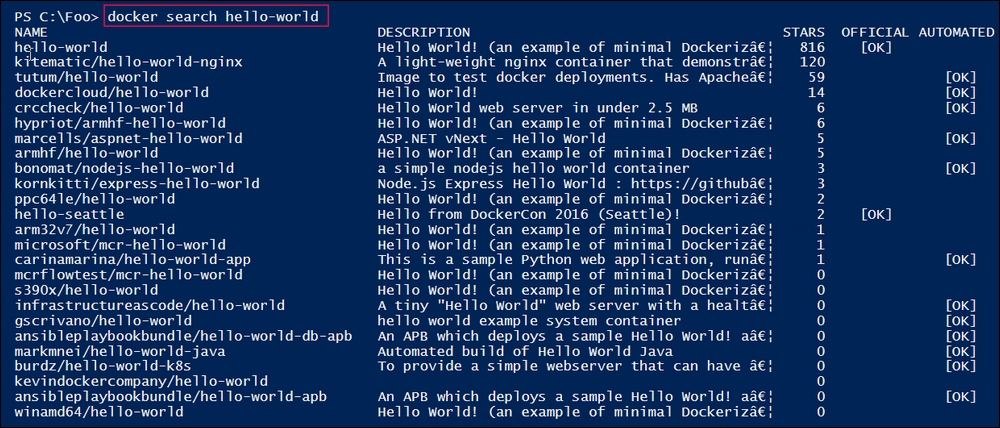
1. View the remaining images and verify that the containers have been removed:

docker image ls

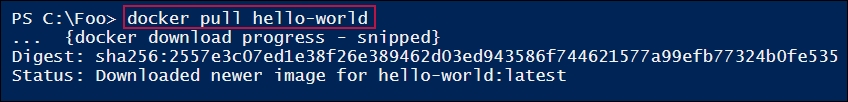
docker container ls

## How it works...

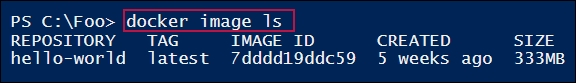
In step 1, you search the Docker registry for hello-world images, like this:



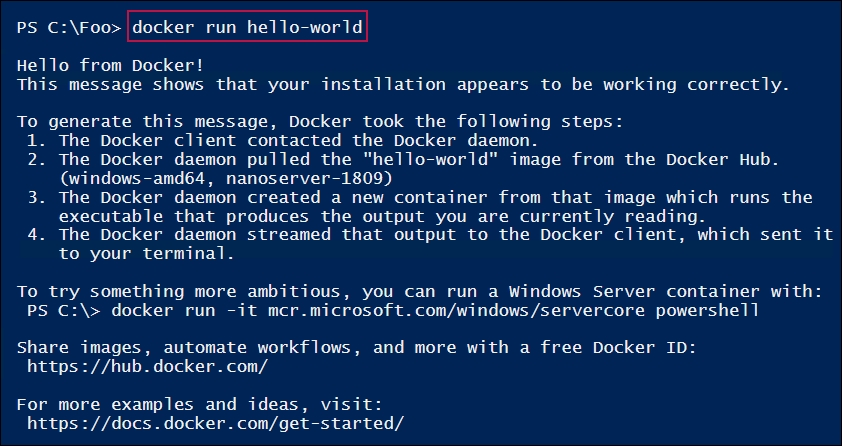
In step 2, you pull the official Docker hello-world image and download it to your container host, which looks like this:



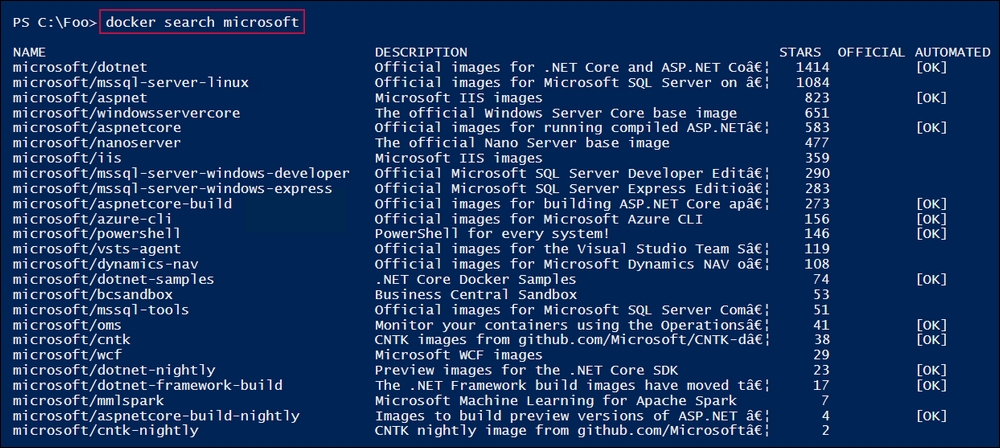
In step 3, you use the docker command to list the images on CH1, which looks like this:



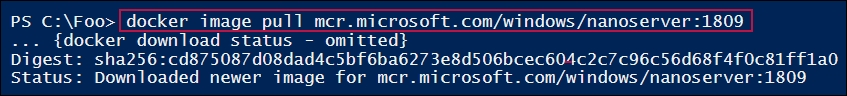
In step 4, you run the Docker hello-world image, which produces the following output:



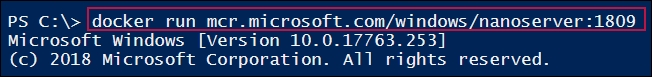
In step 5, your search for Microsoft images in the online Docker registry, which produces output like this:



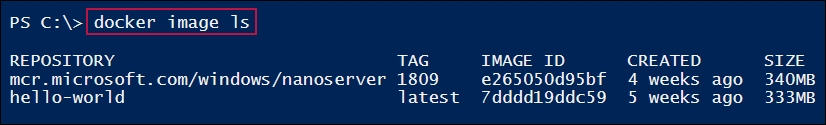
In step 6, you download another base image from the Docker registry with output like this:



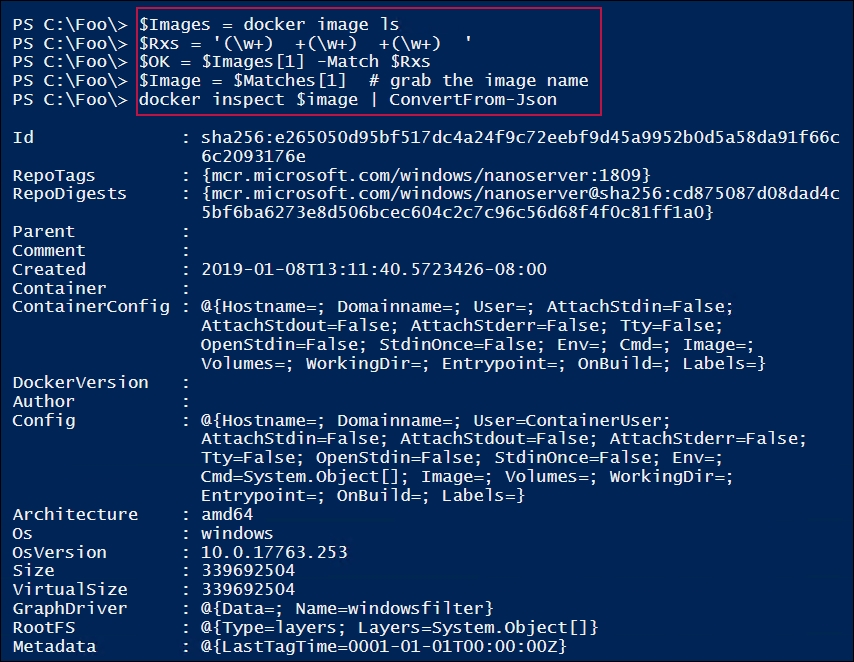
In step 7, you run the nanoserver:1809 image, which looks like this:



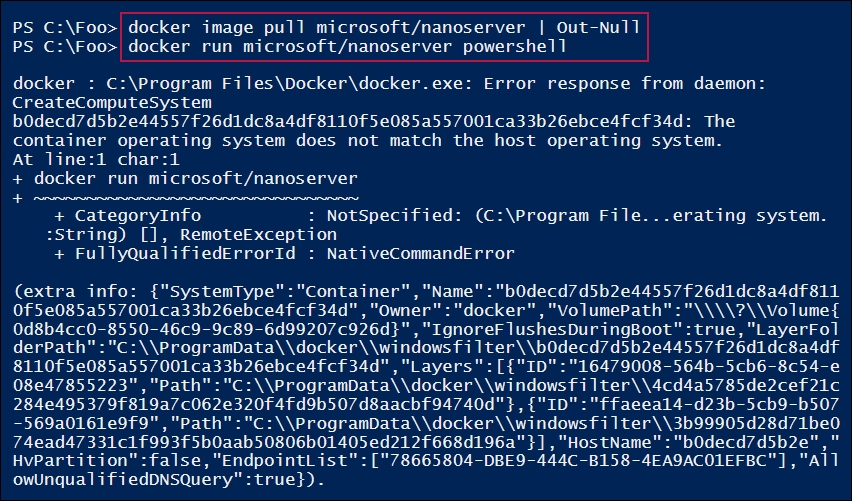
In step 8, you look at the images you have downloaded so far, which now looks like this:



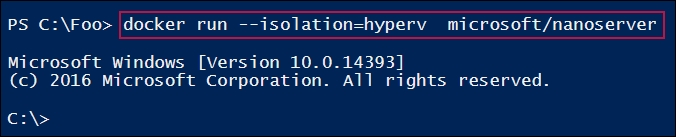
In step 9, you use the docker inspect command to get more information about the nanoserver image, which looks like this:



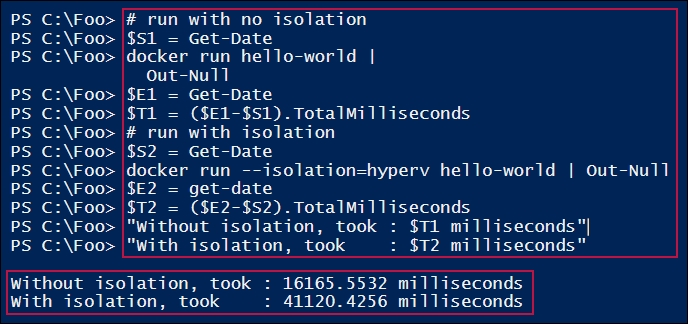
In step 10, you download and run a container that needs isolation, which generates a Docker error, as follows:



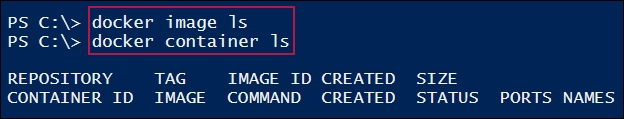
In step 11, you run this same image, but this time using Hyper-V isolation. The results are more like what you might expect, as shown in the following screenshot:



In step 12, you examine the performance impact of using Hyper-V isolation, which looks like this:



In step 13, you remove all the container images on CH1, which produces no output. In step 14, you list the current images and containers on CH1, which looks like this:



## There's more...

In step 4, you run the hello-world container you downloaded in step 2. This container prints out some text, then exits. This is a great demonstration that your container host is up, running, and able to host containers.

In step 7, you download the nanoserver image. By using the :1809 tag in the image name, you ensure that docker.exe downloads an image which corresponds to the kernel in Windows Server 2019 (and Windows 10 1809).

In step 9, you use the docker image command to get all the images on the container host (CH1). Docker.exe returns the images on the system as an array of strings (that is, not objects!). The first ($Images[0]) entry is a string of the line of column headers. The next two entries in the $Images array relate to the nanoserver and hello-world images, respectively. You create the $Image value by using a regular expression to pull the Docker image name from the string returned by docker. Those more experienced in the dark arts of regular expressions could no doubt improve this step.

In step 10, you download and attempt to run a container whose built-in base OS is a different version to the OS running on the container host (CH1). This is to be expected, and you have two alternatives. The first is to use a more up-to-date container image—one that matches yours (or create an updated image as shown in the Using a Dockerfile to create and use a container recipe). The other alternative is to use Hyper-V isolation, which works fine, as you saw in step 11.

Using isolation, however, has a performance implication. The approach does provide added security, which may be appropriate in a share-hosting environment. There is, however, a significant startup performance hit. The good news is that the overhead of running a container using the isolation provided by Hyper-V is not huge once the container is up and running.

In step 13, you start up a detached container. In doing so, you tell Docker to map the local host's port 80 to port 80 in the container. So, when you browse to http://CH1, you see the standard IIS startup screen, which comes from IIS running in the container. We look at doing a bit more with IIS in the Deploying IIS in a container and Using a Dockerfile to create and use a container recipes.

## See also

For more information around getting started with Docker, see this link: <https://docs.docker.com/get-started/>.

# Deploying IIS in a container

In the Deploying a hello-world sample recipe, you downloaded and ran multiple container images. One of those images was the microsoft/iis image. This image contains IIS with only the default website set up and working. When you run this image, IIS is loaded in the container, and with port mapping in place, you can easily see the website in the container, even though IIS is not loaded on the container host.

For this recipe to work as written, the container has to have the same base OS image as your container host. This recipe, therefore, assumes that the host you are using for the recipes in this chapter is 1809 (for both Windows 10 and Server 2019). If you run this recipe on an older OS, for example, 1709 or Server 2016, Docker would not run the container and you get an error, so ensure that the kernel versions on the container host and the container itself are the same (or use Hyper-V isolation).

## Getting ready

This recipe uses the CH1 host, which you configured in the Configuring a container host recipe.

## How to do it...

1. Create the C:\Reskitapp folder:

$EA = @{ErrorAction='SilentlyContinue'}

New-Item -Path C:\ReskitApp -ItemType Directory @EA

1. Create a web page:

$Fn = 'C:\Reskitapp\Index.htm'

$Index = @"

<!DOCTYPE html>

<html><head><title>

ReskitApp Container Application</title></head>

<body><p><center><b>

HOME PAGE FOR RESKITAPP APPLICATION</b></p>

Running in a container in Windows Server 2019<p>

</center><br><hr></body></html>

"@

$Index | Out-File -FilePath $Fn

1. Get a server core with a server core image (with IIS loaded) from the Docker registry:

docker pull mcr.microsoft.com/windows/servercore/iis |

Out-Null

1. Run the image as a container named rkwebc:

$image = 'mcr.microsoft.com/windows/servercore/iis'

docker run -d -p 80:80 --name rkwebc "$image"

1. Copy our file into the container:

Set-Location -Path C:\Reskitapp

docker cp .\index.htm rkwebc:c:\inetpub\wwwroot\index.htm

1. View the page:

Start-Process "Http://CH1.Reskit.Org/Index.htm"

1. Clean up:

docker rm rkwebc -f | Out-Null

docker image rm mcr.microsoft.com/windows/servercore/iis |

Out-Null

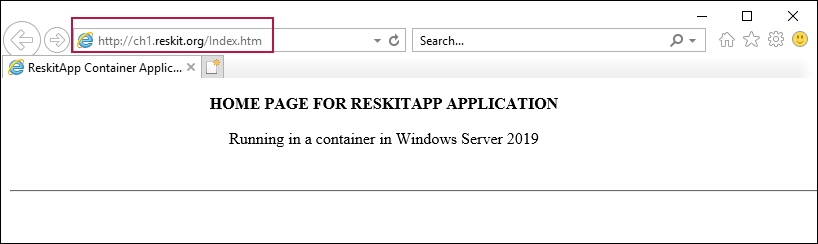
## How it works...

In step 1, you create a folder on CH1 to hold a web page. In step 2, you create a very simple home HTML page. In step 3, you download a server core image that contains IIS. These first three steps produce no output.

In step 4, you run the image as a container named rkwebc and bridge the local port 80 to the container's port 80, which looks like this:

How it works...

In step 5, you copy the web page HTML file from CH1 into the container, which produces no output. In step 6, you view the container's web page, which looks like this:



Finally, in step 7, you forcibly stop the container and remove the container image, which produces no output.

## There's more...

This recipe creates a new web page (in step 1 and step 2) on the CH1 host, then copies that file into the running container (step 5). When you run the container, you use port forwarding to instruct Docker to forward port 80 on the container host to port 80 in the container. This means that although you do not have IIS loaded on CH1, it is loaded, active, and runs a website inside the container. In this recipe, you are making use of the existing network address/name (that is, of the CH1 host) to access the container's website. You can see another method to push data into a container in the Using a Dockerfile to create and use a container recipe.

In step 5, you use the docker cp command to copy files from the container host into the container. In this recipe, you only add (and in step 6, view) a single page to the existing default website that was loaded by installing IIS. You can use the docker exec command to create a new website inside the container and run that, much like you did in the recipes in the IIS chapter. You could also copy all the files and other resources that are necessary for a rich website, set up SSL, and make use of host headers to support multiple containers.

In this recipe, you forwarded traffic inbound to port 80 on the container host to port 80 in the container. This is a very simple way to use containers and container networking. You could also create a Docker network and give your container unique IP settings. For more on Docker networking, see the following: <http://rafalgolarz.com/blog/2017/04/10/networking_golang_app_with_docker_containers/> and <https://docs.docker.com/v17.09/engine/userguide/networking/>. You can, as ever, use your search engine to discover more about containers and networking. One thing to keep in mind as you search is that much of the search results relate to running containers on Linux, where the networking stack is quite different, and differently managed.

## See also

This recipe uses the docker cp command to copy a file into the container. There are other ways to transfer information between your container and other hosts in your environment. See <https://markheath.net/post/transfer-files-docker-windows-containers> to take a look at some methods you can use to transfer data into and out of your containers.

# Using a Dockerfile to create and use a container

Containers can be used in a variety of ways. In most cases, you are going to want to build your own custom images, complete with an operating system, OS features, and applications. A great way to build your image is to use a Dockerfile containing the instructions for building a new image, and then use the docker build command to create a customized container you can then run.

## Getting ready

In this recipe, you use the container host, CH1, that you set up in the Configuring a container host recipe.

## How to do it...

1. Create a folder and Set-Location to the folder on CH1:

$SitePath = 'C:\RKWebContainer'

$NIHT = @{

Path = $SitePath

ItemType = 'Directory'

ErrorAction = 'SilentlyContinue'

}

New-Item @NIHT | Out-Null

Set-Location -Path $NIHT.Path

1. Create a script to run in the container to create a new site in the container:

$SB = {

# 2.1 create folder in the container

$SitePath = 'C:\RKWebContainer'

$NIHT = @{

Path = $SitePath

ItemType = 'Directory'

ErrorAction = 'SilentlyContinue'

}

New-Item @NIHT | Out-Null

Set-Location -Path $NIHT.Path

# 2.2 Create a page for the site

$PAGE = @'

<!DOCTYPE html>

<html>

<head><title>Main Page for RKWeb.Reskit.Org</title></head>

<body><p><center><b>

HOME PAGE FOR RKWEBr.RESKIT.ORG</b></p>

Containers and PowerShell Rock!

</center/</body></html>

'@

$PAGE | Out-File $SitePath\Index.html | Out-Null

# 2.3 Create a new web site in the container that uses Host headers

$WSHT = @{

PhysicalPath = $SitePath

Name = 'RKWeb'

HostHeader = 'RKWeb.Reskit.Org'

}

New-Website @WSHT

} # End of $SB script block

1. Save the script block to the file:

$SB | Out-File $SitePath\Config.ps1

1. Create and test a new A record for our soon-to-be containerized site:

Invoke-Command -Computer DC1.Reskit.Org -ScriptBlock {

$DNSHT = @{

ZoneName = 'Reskit.Org'

Name = 'RKWeb'

IpAddress = '10.10.10.221'

}

Add-DnsServerResourceRecordA @DNSHT

}

Resolve-DnsName -Name Rkweb.Reskit.Org

1. Create a Dockerfile that contains build instructions:

$DF = @"

FROM mcr.microsoft.com/windows/servercore:1809

LABEL Description="RKWEB Container" Vendor="PS Partnership" Version="1.0.0.42"

RUN powershell -Command Add-WindowsFeature Web-Server

WORKDIR C:\\RKWebContainer

COPY Config.ps1 Config.ps1

RUN powershell -Command .\config.ps1

"@

$DF | Out-File -FilePath .\Dockerfile -Encoding ASCII

1. Build the image:

docker build -t rkwebc .

1. Run the image:

docker run -d -t --name rkwebc -p 80:80 rkwebc | Out-Null

1. Check that the rkwebc container is running:

docker ps

1. Get the page using Invoke-WebRequest:

Invoke-WebRequest -UseBasicParsing HTTP://RKweb.Reskit.Org

1. View the page using a browser:

Start-Process 'HTTP://RKWeb.Reskit.Org'

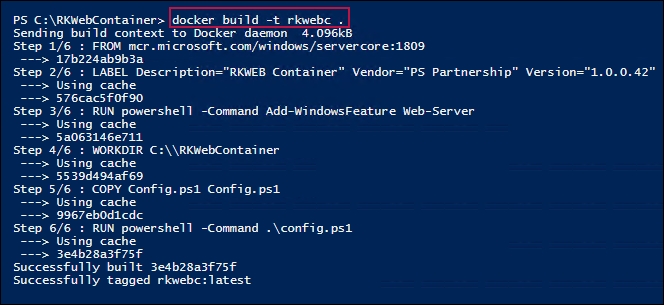
1. Clean up forcibly:

docker container rm rkwebc -f | Out-Null

## How it works...

In step 1, you create a folder, C:\RKWebContainer in CH1, to all the files needed in this recipe. In step 2, you create a script block that you later use to create a website in the container. In step 3, you save this script block to a file on CH1. In step 4, you set up a new DNS A record for RKweb.Reskit.Org that points to the same IP address as the CH1 host. In step 5, you create a file, Dockerfile, that contains the Docker image-building instructions. These steps produce no output.

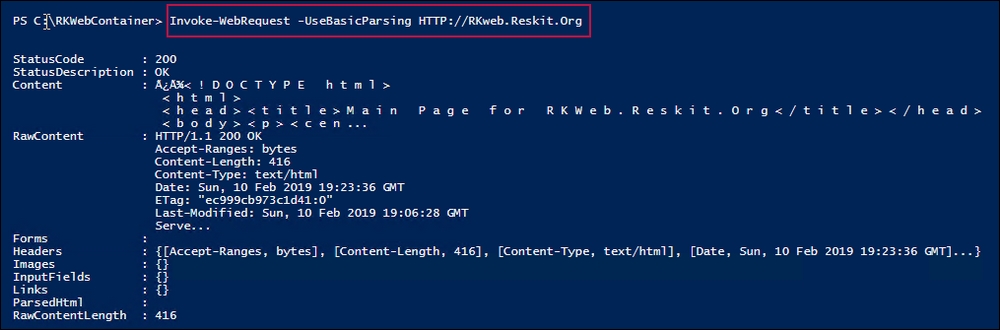
In step 6, you use the docker build command to build a customized image, rkwebc, on CH1. The output looks like this:



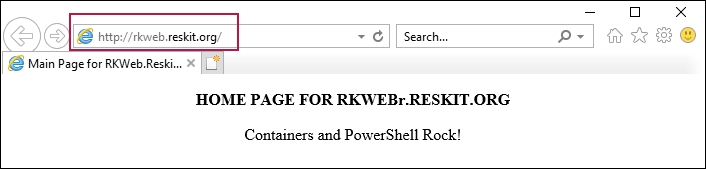
With your image created, in step 7, you run the image as a container, which produces no output. In step 8, you use the docker ps command to view the rkwebc container running on CH1, which looks like this:

How it works...

In step 9, you use Invoke-WebRequest to view the newly created website. The output looks like this:



In Step 10, you view the site in the browser, which looks like:



## There's more...

In this recipe, you use a base container image that you have to download from the Docker registry (mcr.microsoft.com/windows/servercore:1809). Then, you build a container that has the web server feature added and in which you can run the Config.ps1 file to configure the container to run your website. For more information on Dockerfiles, see this link: <https://docs.docker.com/engine/reference/builder/>.

## See also

In this recipe, you build and run a Dockerfile to create an image. In the Dockerfile, you use the RUN instruction to run PowerShell twice during the building of the image. To optimize the container, you could run the two commands (to add the Windows feature and to run the Config.Ps1 script) as a single Docker image. For some tips on how to build container images, see <https://cloud.google.com/blog/products/gcp/7-best-practices-for-building-containers>.